

DATABASE MANAGEMENT SYSTEM

R25 CSE – DATA SCIENCE

UNIT – I

Data: Data refers to a collection of raw facts, unprocessed facts. It can be in various formats such as text, numbers, images, videos, audio, or any other type of digital or analog representation.

Ex: 25, Suresh, Hyderabad, etc

Information: Information refers to the processed data, and presented in a meaningful way to convey a specific message

Ex: the age of Suresh is 25

Database: A database is an organized collection of related data stored and accessed electronically through the use of a database management system.

Small databases can be stored on a file system, while large databases are hosted on computer clusters or cloud storage.

Meta Data: Metadata represents data about data. Metadata enriches the data with information, which makes it easier to discover, use and manage. In other words, metadata provides information about the characteristics, attributes, and properties of data.

Definition of DBMS

DBMS: DBMS Stands for **Database Management System**. It refers to a System / Software / Collection of programs that enables the creation, organization, and management of databases. A DBMS provides a structured and efficient way to store, retrieve, update, and manipulate large volumes of data.

Functionalities of DBMS:

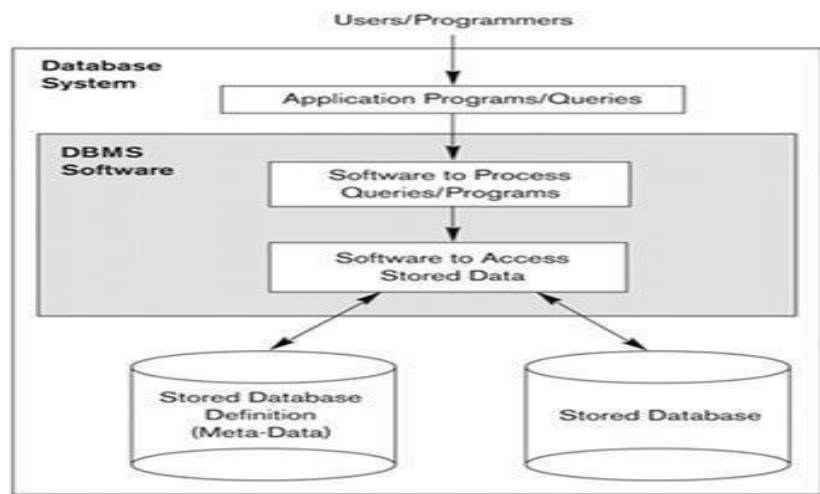
- **Define:** Specifying the data type, structures and constraints for the data to be stored.
- **Construct:** Process of storing data on some storage medium.
- **Manipulate:** Querying the database to retrieve specific data, updating database, and generating reports.
- **Share:** Allows multiple users and programs to access the database concurrently.
- **Protection:** Protection of the database from unauthorized access or from hardware and software failures and also maintenance of the database for long period of the time.

Properties of DBMS:

1. A database **represents** some aspects of the **real world** (mini world)
2. A database is a **logically coherent** collection of data with some inherent meaning.
3. A database is **designed, built** and **populated** with data for a **specific purpose**.

Database Environment:

A database system environment refers to the surroundings and components in which a database system operates. It includes various elements that interact with the database system to facilitate data storage, retrieval, management, and processing.

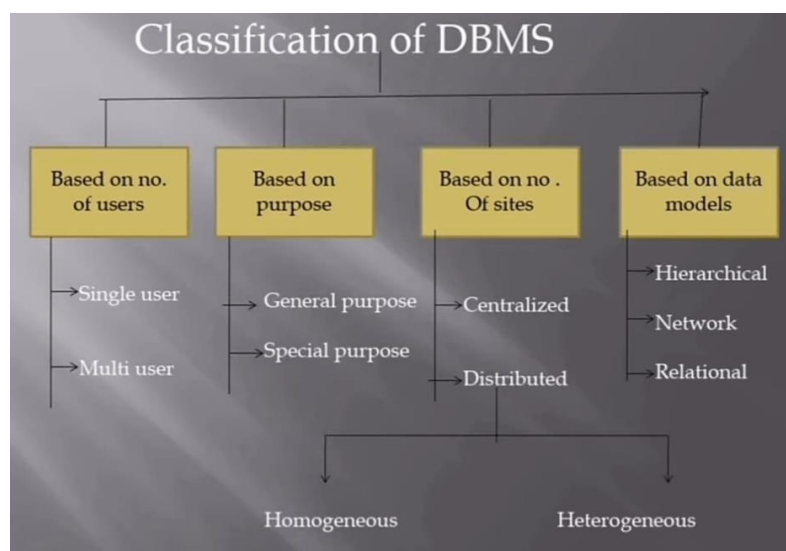


Here are some key components of a database system environment:

- Hardware
- Operating System
- Database Management System (DBMS)
- Database
- Application Programs
- Users
- Network
- Backup and Recovery Systems
- Security Infrastructure
- Data Integration and ETL Tools

Classification of DBMS

Database Management Systems (DBMS) can be classified into several categories based on different criteria. Here are some common classifications of DBMS:



Historical Perspective:

1. In the early 1960s, the first general-purpose DBMS, designed by **Charles Bachman**, was called the Integrated Data Store (IDS) which was based on network data model
2. In the late 1960s, IBM developed the Information Management System (IMS) DBMS. Which was based on hierarchical data model
3. In 1970, relational data model was developed by **Dr. E.F. Codd**. Representing data in the form of relations (tables)
4. In 1976, Peter Chan developed Entity Relationship(ER) model widely used in database design
5. In 1980, SQL (Structured Query Language) became standard. IBM launched DB2.
6. In 1990, Oracle, IBM DB2, and PostgreSQL introduced ORDBMS capabilities. Now various companies are developing and maintaining the different DBMS software.

Vendors	Products
Oracle Corporation	Oracle Database MySQL Oracle NoSQL Database Oracle TimesTen In-Memory Database Oracle Berkeley DB
Microsoft Corporation	Microsoft SQL Server Azure SQL Database Azure Cosmos DB Access (Desktop Database System)
IBM Corporation	IBM Db2 IBM Informix IBM Cloudant (NoSQL DB) IBM IMS (Hierarchical DB)
Amazon Web Services (AWS)	Amazon Aurora Amazon DynamoDB Amazon Redshift
SAP SE	SAP HANA SAP ASE (formerly Sybase ASE) SAP IQ (formerly Sybase IQ)

Database / DBMS Applications: Database Management Systems (DBMS) are widely used to build various types of database applications. Here are some common types of applications that make use of DBMS:

1. Universities
2. Banking
3. Railway Reservation System
4. Social Media Sites
5. Library Management System
6. E-commerce Websites
7. Medical
8. Accounting and Finance
9. Industries
10. Airline Reservation System
11. Telecommunication
12. Manufacturing
13. Human Resource Management
14. Broadcasting
15. Insurance

File based system versus a DBMS:

Basis	DBMS Approach	File System Approach
Meaning	DBMS is a collection of data. In DBMS, the user is not required to write the procedures.	The file system is a collection of data. In this system, the user has to write the procedures for managing the database.
Sharing of data	Due to the centralized approach, data sharing is easy.	Data is distributed in many files, and it may be of different formats, so it isn't easy to share data.
Data Abstraction	DBMS gives an abstract view of data that hides the details.	The file system provides the detail of the data representation and storage of data.
Security and Protection	DBMS provides a good protection mechanism.	It isn't easy to protect a file under the file system.
Cost	The database system is expensive to design.	The file system approach is cheaper to design.
Data Redundancy and Inconsistency	Due to the centralization of the database, the problems of data redundancy and inconsistency are controlled.	In this, the files and application programs are created by different programmers so that there exists a lot of duplication of data which may lead to inconsistency.
Structure	The database structure is complex to design.	The file system approach has a simple structure.
Data Independence	In this system, Data Independence exists,	In the File system approach, there exists no Data Independence.
Integrity Constraints	Integrity Constraints are easy to apply.	Integrity Constraints are difficult to implement in file system.
Flexibility	The changes are more easily with a database approach.	less flexibility
Examples	Oracle, SQL Server, Sybase etc.	Cobol, C++ etc.

What is DBA? What are the roles and responsibilities of DBA?

DBA stands for Database Administrator. A person who has central control over the database system, DBA may be a single or group of persons who are responsible for implementing the database system within an Organization. DBA is only the guardian of the organization but not the owner.

Roles and responsibilities of DBA

1. Responsible for Design, Implementation and maintenance of database
2. Software installation and maintenance
3. Data Extraction, Transformation and loading
4. Specialized in data handling
5. Database backup & recovery
6. Security
7. Authentication
8. Performance monitoring
9. Data storage
10. Integrity constraint

Advantage and disadvantage of DBMS:

Advantages of DBMS:

1. **Data Centralization:** DBMS allows for the centralization of data, meaning that data is stored in a single location and can be accessed by multiple users and applications. This reduces data redundancy and inconsistency.
2. **Data Consistency and Integrity:** DBMS enforces integrity constraints and rules on the data to ensure consistency.
3. **Data Security:** DBMS offers security features to protect sensitive data. It allows for user authentication, authorization, and access control, ensuring that only authorized users can access and modify the data.
4. **Data Sharing and Collaboration:** DBMS enables concurrent access to the database by multiple users and applications.
5. **Data Independence:** This allows for modifications in the physical storage without affecting the logical view, providing data independence.

Disadvantages of DBMS:

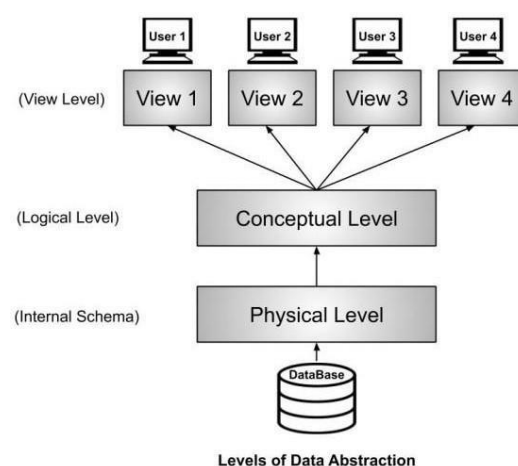
1. **Complexity:** Implementing and managing a DBMS can be complex and requires specialized knowledge and expertise.
2. **Cost:** Acquiring and maintaining a DBMS can be expensive.
3. **Performance Overhead:** DBMS introduces overhead in terms of processing and resource utilization.
4. **Single Point of Failure:** If the DBMS experiences a failure or downtime, it can disrupt access to the entire database, affecting all applications and users dependent on it.
5. **Learning Curve:** Users and developers need to learn the specific query languages and programming interfaces supported by the DBMS.

Data Abstraction and its phases (levels):

Data Abstraction is a process of hiding unwanted or irrelevant details from the end user. Data abstraction allows users to interact with the database without needing to understand the underlying complexities. It provides a conceptual framework that separates the logical view of the data from its physical storage. This separation enables users to focus on the essential aspects of the data and its manipulation, rather than being concerned with the technical details.

There are typically three levels of data abstraction in a DBMS:

1. **External Level (also known as the View Level):** This level represents the individual user's view of the database. It defines how each user or application sees and interacts with the data. Users can define their own customized views of the data by specifying the desired data fields, their relationships, and the operations they want to perform. The external level provides a simplified and tailored view of the database for different users, which enhances data security and privacy.
2. **Conceptual Level (also known as the Logical Level):** This level represents the overall logical structure and organization of the entire database. It focuses on the relationships between different data elements and provides a global view of the data. The conceptual level defines the entities, attributes, and their relationships using a data model (such as the Entity-Relationship Model or the Relational Model). It acts as an intermediary between the external and internal levels and ensures data independence, where changes in the physical storage or organization of data do not affect the external views.
3. **Internal Level (also known as the Physical Level):** This level represents the physical storage and implementation details of the database on the underlying hardware. It describes how the data is stored, indexed, and accessed by the DBMS. The internal level deals with low-level details such as data storage structures, file organization, and access methods. It optimizes data storage and retrieval for efficient performance and supports the logical and external levels by providing the necessary data manipulation capabilities.



Data Independence in DBMS:

Data independence is the ability to modify the scheme without affecting the programs and the application to be rewritten. Data is separated from the programs, so that the changes made to the data will not affect the program execution and the application.

We know the main purpose of the three levels of data abstraction is to achieve data independence. If the database changes and expands over time, it is very important that the changes in one level should not affect the data at other levels of the database. This would save time and cost required when changing the database.

There are two levels of data independence based on three levels of abstraction. These are as follows –

- Physical Data Independence
- Logical Data Independence

Physical Data Independence:

Physical Data Independence means changing the physical level without affecting the logical level or conceptual level. Using this property, we can change the storage device of the database without affecting the logical schema.

The changes in the physical level may include changes using the following –

- A new storage device like magnetic tape, hard disk, etc
- A new data structure for storage
- A different data access method or using an alternative files organization technique.
- Changing the location of the database.

Logical Data Independence:

Logical view of data is the user view of the data. It presents data in the form that can be accessed by the end user.

Codd's Rule of Logical Data Independence says that users should be able to manipulate the Logical View of data without any information of its physical storage. Software or the computer program is used to manipulate the logical view of the data.

The changes in the logical level may include –

- Change the data definition.
- Adding, deleting, or updating any new attribute, entity or relationship in the database.

Data Models:

Data models describe how a database's logical structure is represented. In a database management system, data models are essential for introducing abstraction. Data models specify how data is linked to one another, as well as how it is handled and stored within the system.

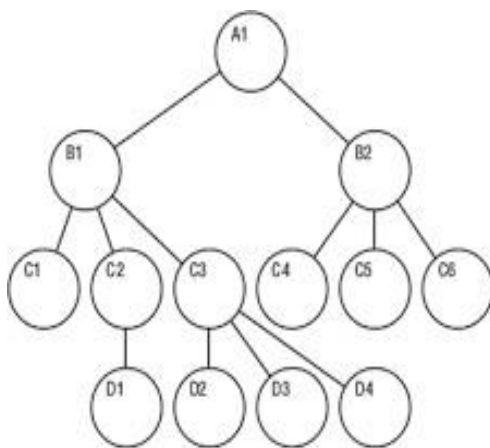
A data model is a conceptual representation of how data is organized, stored, and accessed within a database. It defines the structure, constraints, and relationships of the data.

There are several types of data models commonly used in DBMS

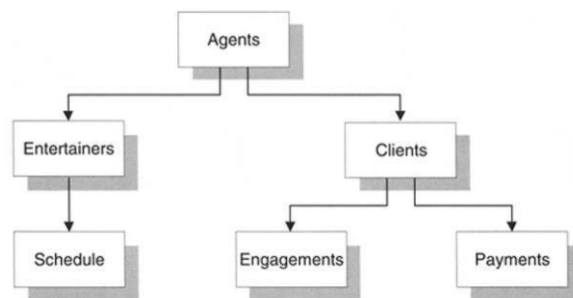
1. Hierarchical Data Model
2. Network Data Model
3. Entity-Relationship (ER) Model
4. Relational Data Model
5. Object-Oriented Data Model

1. Hierarchical Data Model:

The hierarchical model organizes data in a tree-like structure, where each record has a parent-child relationship. It consists of nodes connected by parent-child relationships, forming a hierarchy.



Hierarchical Database model



Advantages of the Hierarchical Model:

1. **Simplicity:** The hierarchical model is relatively simple to understand and implement.
2. **Efficiency:** Retrieving data using the hierarchical model can be efficient, especially when navigating through parent-child relationships.
3. **Data Integrity:** The hierarchical model enforces data integrity by maintaining strict parent-child relationships. It ensures that each child record has a single parent, preventing anomalies and inconsistencies in the data.

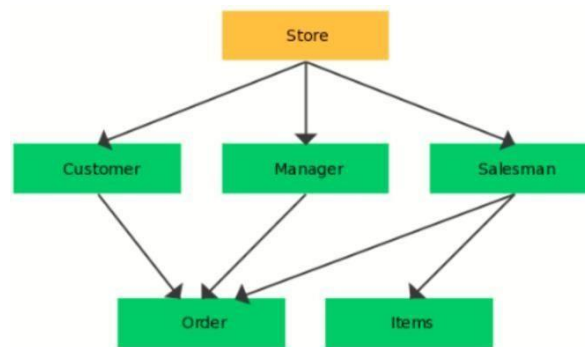
Disadvantages of the Hierarchical Model:

1. **Lack of Flexibility:** Drawbacks of the hierarchical model is its lack of flexibility. The strict one-to-many parent-child relationship can be limiting when dealing with complex or evolving data structures.
2. **Data Redundancy:** In the hierarchical model, data redundancy can occur since the same data might need to be duplicated across multiple parent-child relationships

2. Network Data Model:

Data Organization in the network model is just like a graph rather than a tree, a child record can have any number of parent records

This was the most widely used model before the relational database model was introduced



Advantages of Network Model :

1. This model is very simple and easy to design like the hierarchical data model.
2. This model is capable of handling multiple types of relationships which can help in modeling real-life applications, for example, 1: 1, 1: M, M: N relationships.
3. In this model, we can access the data easily, and also there is a chance that the application can access the owner's and the member's records within a set.

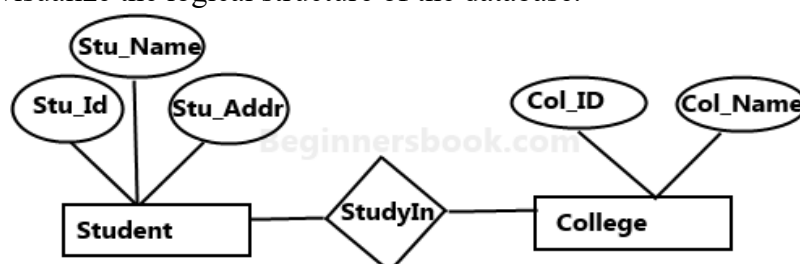
Disadvantages of Network Model:

1. The schema or the structure of this database is very complex in nature as all the records are maintained by the use of pointers.
2. There's an existence of operational anomalies as there is a use of pointers for navigation which further leads to complex implementation.
3. The design or the structure of this model is not user-friendly.

3. Entity-Relationship (ER) Model

The ER diagram was proposed by Peter Chen in 1976 as a visual tool to represent the ER model. ER model in DBMS is the high-level data model. It stands for the Entity- relationship model and is used to represent a logical view of the system from a data perspective. It works around real-world entities and the associations among them.

ER model makes use of **ER diagrams**, which are the diagrams sketched to design a database. ER diagrams are built on **three** basic concepts: **entities**, **attributes**, and **relationships** between them. An ER diagram in DBMS defines **entities**, **associated attributes**, and **relationships between entities**. This helps visualize the logical structure of the database.



Sample E-R Diagram

Advantages of the ER Model:

1. **Simplicity:** The ER model uses a simple and intuitive graphical representation, which makes it easy to understand and communicate the database structure to stakeholders.
2. **Clear representation of relationships:** The ER model provides a clear depiction of the relationships between entities, such as one-to-one, one-to-many, and many-to-many relationships. This helps in understanding the data dependencies and designing the database accordingly.
3. **Database design:** The ER model assists in the logical design phase of database development. It helps in identifying entities, attributes, and relationships, which serve as a foundation for creating a well-structured database schema.

Disadvantages of the ER Model:

1. **Lack of support for implementation details:** The ER model is a high-level conceptual model and does not provide details about implementation aspects such as data types, indexing, or physical storage considerations.
2. **Limited support for behavior modeling:** The ER model primarily focuses on the static structure of the database and does not explicitly capture the dynamic behavior or operations performed on the data. It lacks constructs to represent actions, triggers, or procedures.

4. Relational data model

The relational data model is a conceptual model used to organize and structure data in a database. It was introduced by Edgar F. Codd in the early 1970s and has since become the most widely used database model

The relational model uses a collection of tables to represent data with columns and rows. Tables are also known as relations.

Relational Data Model

Table:

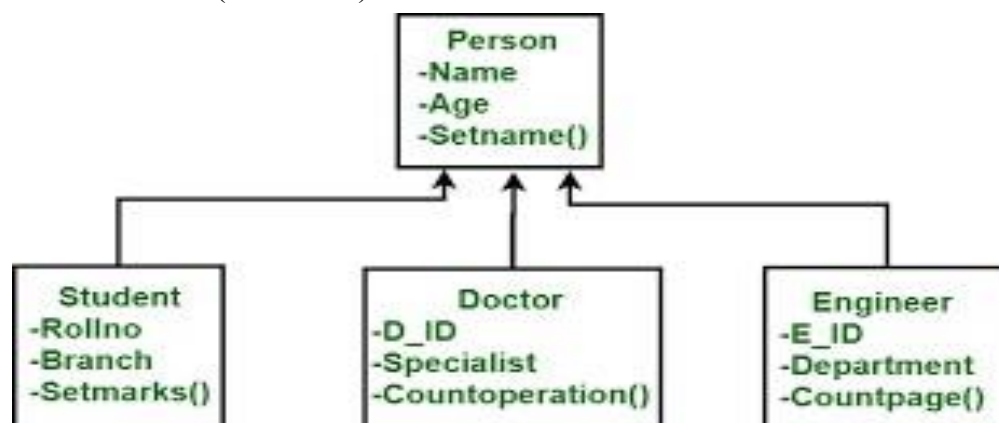
Column 1	Column 2	Column 3	Column 4
Roll No.	Name	Age	Gpa
1	Aryan	21	3
2	Sachin	25	4
3	Prince	20	2.5
4	Anuj	21	3.5

Row 1
Row 2
Row 3
Row 4

5. Object-oriented data model:

The object-oriented data model is a data model that represents data in the form of objects. Objects are instances of classes and represent real-world entities.

Both data and the data relationships are stored into a single structure that's known as an object in the object-oriented data model (or OODM).

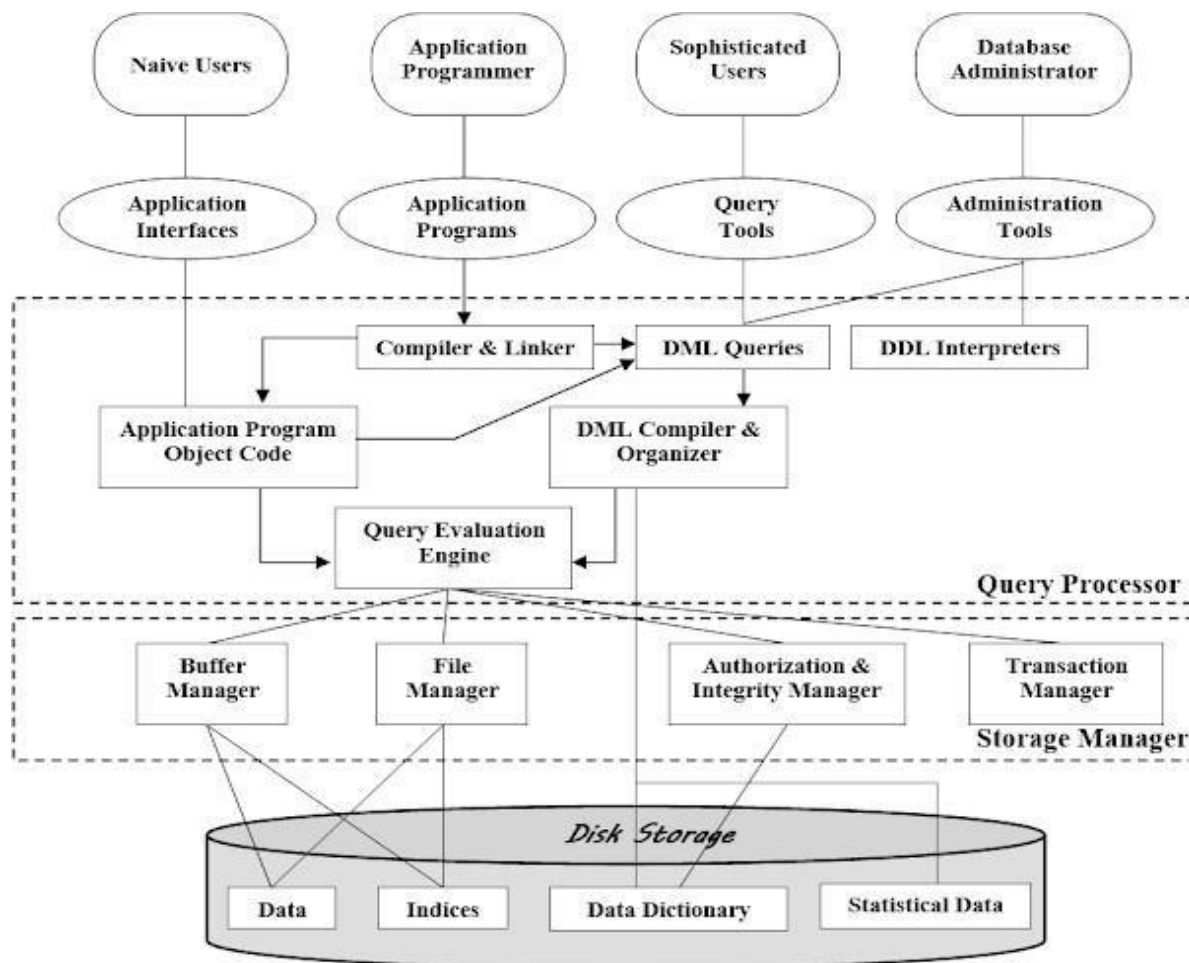


STRUCTURE OF DBMS

DBMS (Database Management System) acts as an interface between the user and the database. The user requests the DBMS to perform various operations (insert, delete, update and retrieval) on the database. The components of DBMS perform these requested operations on the database and provide necessary data to the users. The various components of DBMS are shown below:

Four Parts that make up the Database System are:

- Users
- Query Processor
- Storage Manager
- Disk Storage



Users:

Database Users: Users are differentiated by the way they expect to interact with the system

Application programmers:

- Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

Sophisticated users:

- Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language.
- They submit each such query to a query processor, whose function is to break down DML statements into instructions that the storage manager understands.

Naïve users:

- Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.
- For example, a bank teller who needs to transfer Rs. 50/- from account A to account B invokes a program called transfer. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

Database Administrator:

- Database administrator has full control on database. They may be a group of persons or a single person. They use Data Definition Language to pass on schemas and mapping constraints on database, which further passed to DDL processor.

Query Processor:

The query processor will accept query from user and solves it by accessing the database.

Parts of Query processor:

- **DDL interpreter**
This will interpret DDL statements and fetch the definitions in the data dictionary.
- **DML compiler**
 - a. This will translate DML statements in a query language into low level instructions that the query evaluation engine understands.
 - b. A query can usually be translated into any of a number of alternative evaluation plans for same query result DML compiler will select best plan for query optimization.
- **Query evaluation engine**
This engine will execute low-level instructions generated by the DML compiler on DBMS.

The storage manager components include:

- **Authorization and integrity manager:** Checks for integrity constraints and authority of users to access data.
- **Transaction manager:** Ensures that the database remains in a consistent state although there are system failures.
- **File manager:** Manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager:** It is responsible for retrieving data from disk storage into main memory. It enables the database to handle data sizes that are much larger than the size of main memory.

Disk Storage: The Disk Storage in the Structure of DBMS represents the space where data is stored. It has the following components:

- **Files:** These are responsible for storing the data.
- **Data Dictionary:** It is the repository that maintains the information of the database object and maintains the metadata.
- **Indices:** These are the keys that are used for faster retrieval of data.

Database Design: Database design is the organization of data according to a database model. Properly designed databases are easy to maintain, improves data consistency. The database design process can be divided into six steps.

1. Requirement analysis
2. Conceptual database design
3. Logical database design
4. Schema refinement
5. Physical database design
6. Application and security design

1. Requirement analysis

- ❖ It is necessary to understand what data need to be stored in the database, what applications must be built, what are all those operations that are frequently used by the system.
- ❖ The requirement analysis is an informal process and it requires proper communication with user groups.
- ❖ There are several methods for organizing and presenting information gathered in this step. Some automated tools can also be used for this purpose.

2. Conceptual database design

- ❖ The information gathered, is used to develop a high-level description of the data to be stored in the database
- ❖ This is a steps in which E-R Model i.e. Entity Relationship model is built.
- ❖ The goal of this design is to create a simple description of data that matches with the requirements of users.

3. Logical database design

- ❖ This is a step in which ER model in converted to relational database schema, sometimes called as the logical schema in the relational data model.

4. Schema refinement

- ❖ In this step, relational database schema is analyzed to identify the potential problems and to refine it.
- ❖ The schema refinement can be done with the help of normalizing and restructuring the relations.

5. Physical database design

- ❖ The design of database is refined further.
- ❖ This step may simply involve building indexes on tables and clustering tables, redesign of parts of the database schema obtained from the earlier design steps.

6. Application and security design

- ❖ Using design methodologies like UML(Unified Modeling Language) try to address the complete software design of the database can be accomplished.
- ❖ The role of each entity in every process must be reflected in the application task.
- ❖ For each role, there must be the provision for accessing and prohibiting some part of database.
- ❖ Thus some access rules must be enforced on the application(which is accessing the database) to protect the security features.

Entity: An entity refers to a real-world object or concept about which we want to collect and store the data that is being represented in the database. It may be a living thing or nonliving thing.

Example: STUDENT, EMPLOYEE, PRODUCT, CUSTOMER, ORDER, etc.

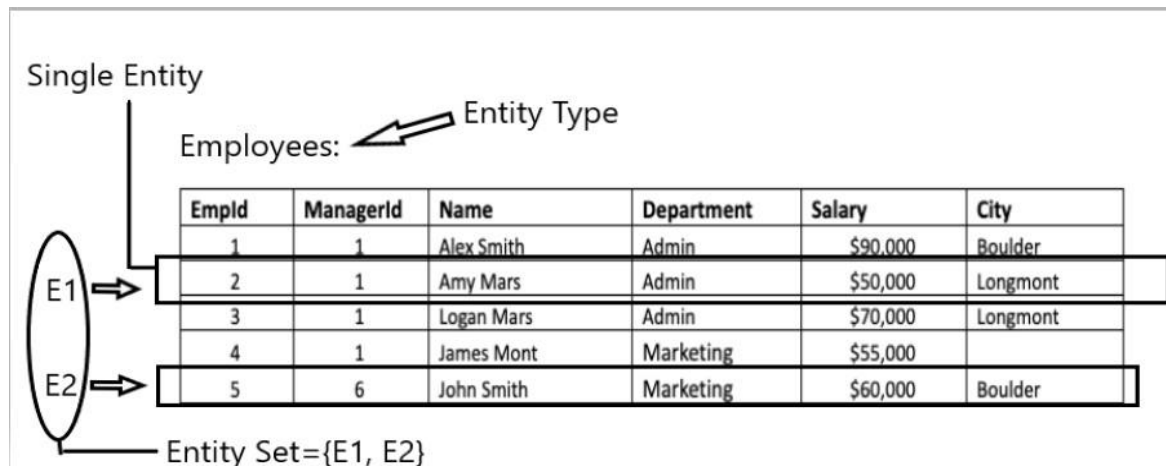
✚ In ERD's entities are represented using a rectangular box

✚ Entity name should be a noun in singular form

✚ Entity names are represented in all capital letters

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

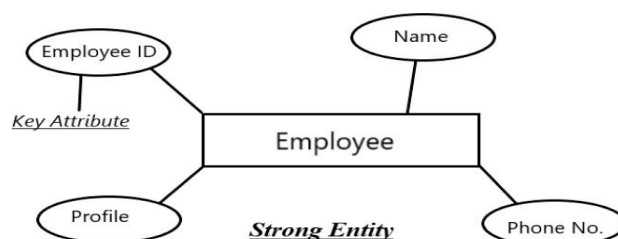
Entity set: The group of related entities of an entity type which share all common attributes is called entity set



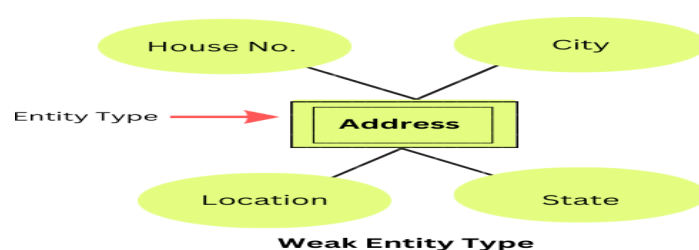
It can be classified into two types

1. Strong Entity
2. Weak Entity

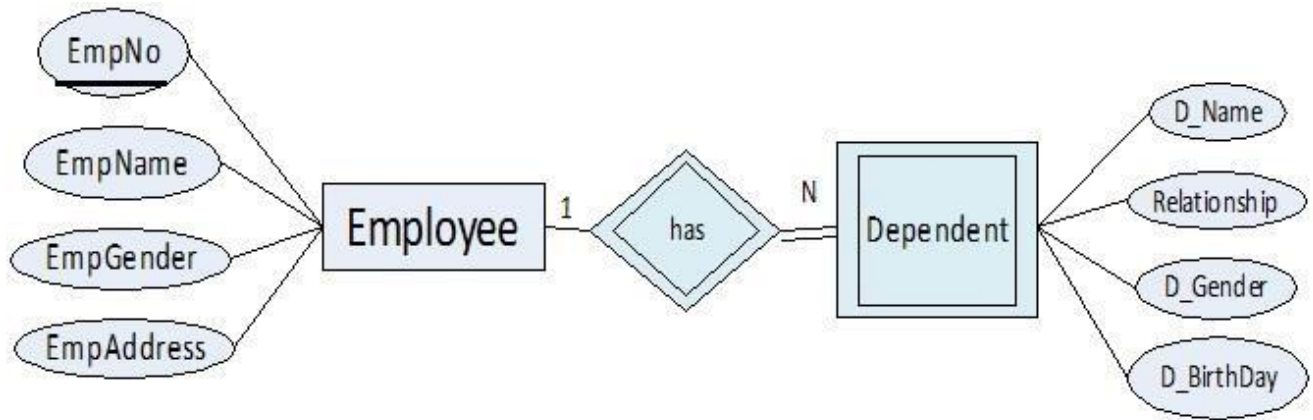
1. Strong Entity: A strong entity is a standalone entity that does not require support from any other entity. A strong entity is an entity type that has a key attribute (primary key), and it is represented by a single rectangle in the ER diagram.



2. Weak Entity: Contrary to a strong entity, a weak entity is dependent and cannot exist independently. It requires the support of a strong entity to validate its existence. Weak entity type doesn't have a key attribute and it is represented by a double rectangle in the ER diagram.



Example:



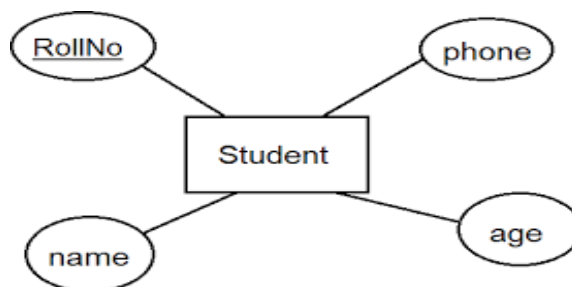
In the above example **Employee** is a strong entity and **Dependent** is a weak entity

Attribute: An attribute is a property, characteristic or feature of an entity. An entity may contain any number of attributes.

Example: Student has attributes like name, age, roll number, and many more.

✚ In an Entity-Relation model, attributes are represented in an oval or ellipse shape.

✚ Attribute names are represented with initial capital letter and remaining small



Types of attributes:

There are several types of attributes in DBMS, each with its unique characteristics and applications. Following are some of the types of attributes-

1. Key Attributes
2. Simple Attributes
3. Composite Attributes
4. Single-valued attributes
5. Multi-valued attributes
6. Stored attribute
7. Derived attribute

1. Key Attributes:

- Attributes that define the originality of the entity,
- Key attributes are those attributes that can uniquely identify the entity in the entity set.
- In a student database, for example, the roll number can be the key property that uniquely identifies each student.
- The key attribute is represented using an oval with the attribute name **underlined** () inside the oval.

2. Simple Attributes:

- An attribute which can not be divided to yield additional attributes/component is called simple attributes (or) An attribute that cannot be further subdivided into components is a simple attribute
- For example, in a student table, the branch attribute cannot be further divided.

3. Composite Attributes:

- An attribute which can be divided to yield additional new attributes/component is called Composite attributes (or) An attribute that can be split into new attributes/component is called a composite attribute
- The address can be further split into house number, street number, city, state, country, and pin code, the name can also be split into first name middle name, and last name.

4. Single-valued attributes:

- Those attributes which can have exactly one value are known as single valued attributes. They contain singular values, so more than one value is not allowed.
- Example: The age of a student. Another example is gender because one person can have only one gender.

5. Multi-valued attributes:

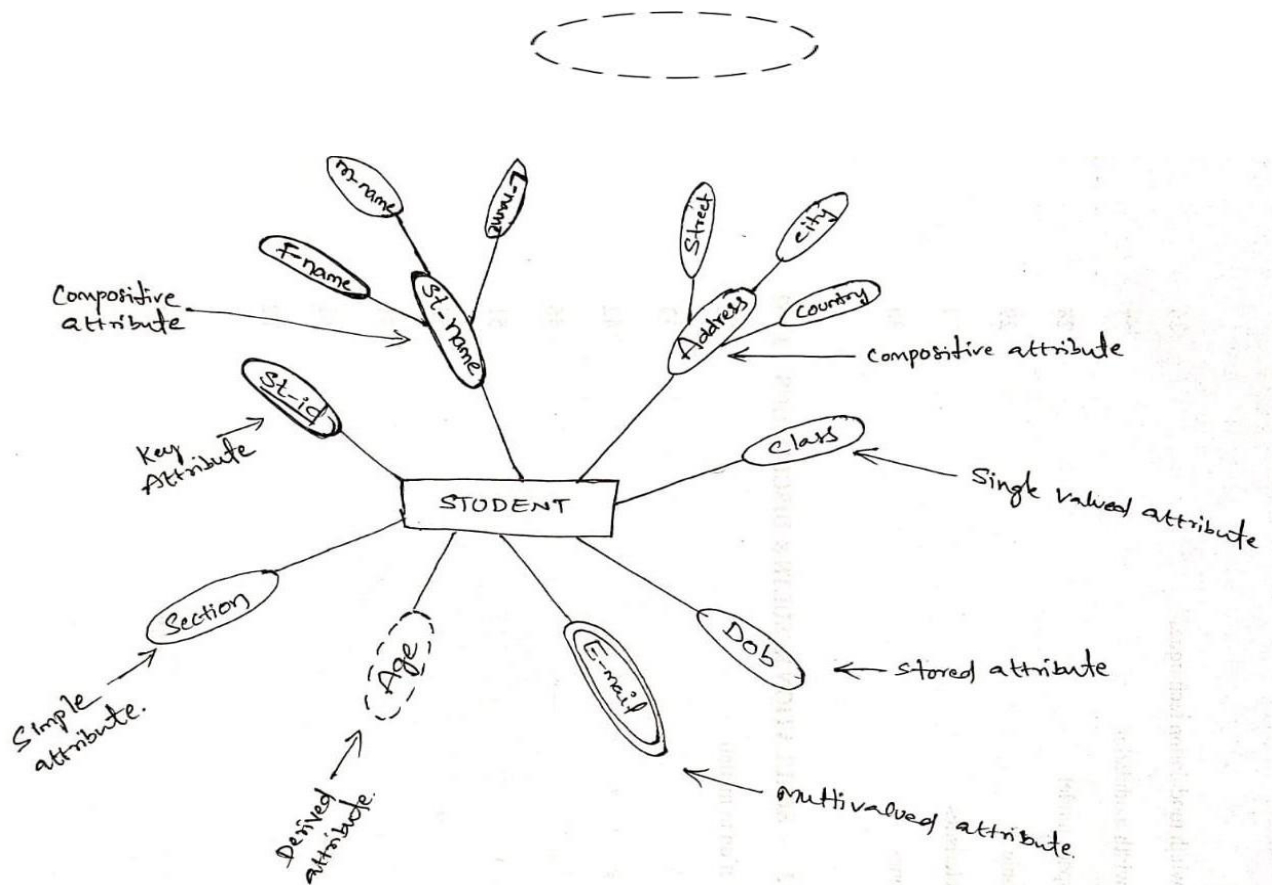
- Those attributes which can have more than one entry or which contain more than one value are called multi valued attributes.
- In the Entity Relationship (ER) diagram, we represent the multi valued attribute by **double oval** representation. 
- For example, one person can have more than one phone number, so that it would be a multi valued attribute. Another example is the hobbies of a person because one can have more than one hobby.

6. Stored attribute:

- Stored attribute is an attribute which supplies value for another attribute is called stored attribute
- Stored attributes represent data that remains constant and fixed for an entity instance. They provide the foundation for deriving derived attributes.
- Example: DOB(Date of birth) is the stored attribute. We can find the Age (attribute value) by using DOB (attribute value) i.e. DOB supplies values to Age, so DOB is stored attribute.

7. Derived attribute

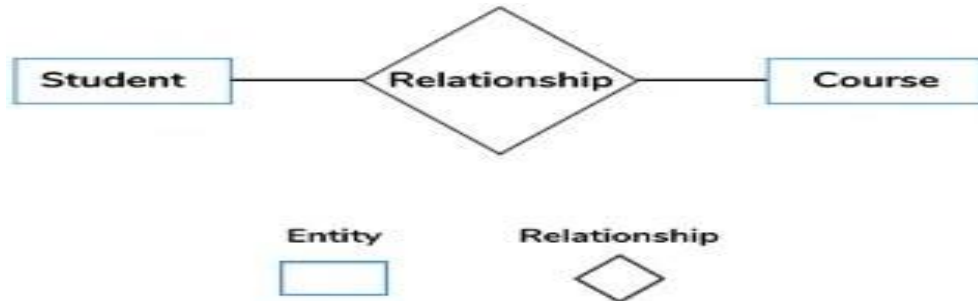
- Derived attributes are those attributes which can be derived from other attribute(s).
- Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database.
- For example, age of the student should not be saved directly in the database, instead it can be derived from Date of Birth.
- In the ER diagram, the **dashed oval** represents the derived attribute



Relationship:

A relationship is the meaningful association among the instances of two or more entity types / same entity type. Relationships define how data in different tables are connected or related to each other.

- In ERD relationships are represented using Rhombus /Diamond shape.
- Relationship name should be a verb.



Relationship Set:

The group of (or) the no. of relationships participated in a relationship type is called relationship set.
(Or)

A collection of various relationships that belongs to the same relationship type is called a relationship set.

Degree of Relationship:

A degree of relationship represents the number of entity types that are associated with a relationship.

Cardinality of relationship:

Cardinality of a relationship can be defined as the no. of entities of an entity type having the association with no. of entities of the another entity type participated in the relationship

(Or)

The cardinality in DBMS refers to the number of entities to which another entity can be linked via a given relation set. It shows the number of times instances of one thing are connected to another.

Relationship types:

In database management systems (DBMS), there are several types of relationships that can exist between entities in a database. Here are the main types of relationships

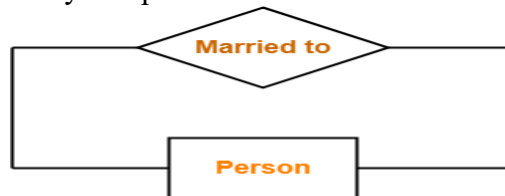
- ❖ Based on the degree relationships are classified into four types
 1. Unary Relationship
 2. Binary Relationship
 3. Ternary Relationship
 4. n-ary Relationship
- ❖ Based on the cardinality relationships are classified into four types
 1. One - to - One Relationship
 2. One - to - Many Relationship
 3. Many - to - One Relationship
 4. Many - to - Many Relationship

Based on the degree relationships are classified into four types:

1. **Unary Relationship:** A unary relationship, also called recursive, is one in which a relationship exists between occurrences of the same entity set. In this relationship, the primary and foreign keys are the same, but they represent two entities with different roles.

The degree of relationship is 1

Ex: One person is married to only one person



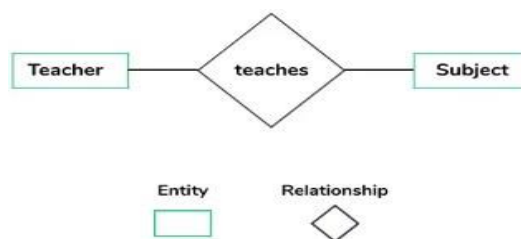
Unary Relationship Set

2. **Binary Relationship:**

A Binary Relationship is the relationship between two different Entities.

The degree of relationship is 2

Ex:

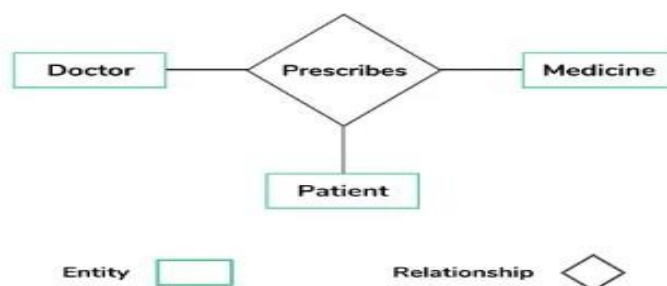


3. **Ternary Relationship:**

A Ternary Relationship is the relationship between three different Entities.

The degree of relationship is 3

Ex:

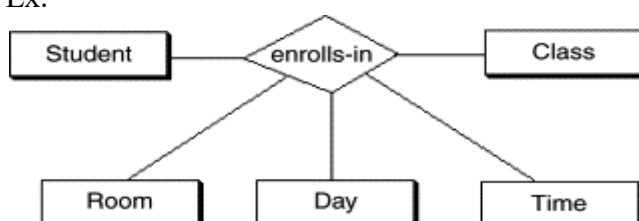


4. **N-ary Relationship:**

A n-ary Relationship is the relationship between 'n' no.of different Entities.

The degree of relationship is n

Ex:



The degree of the above n-ary relationship is 5

Based on the cardinality relationships are classified into four types

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

It is used to create a relationship between two tables in which a single row of the first table can only be related to one and only one records of a second table. Similarly, the row of a second table can also be related to anyone row of the first table.

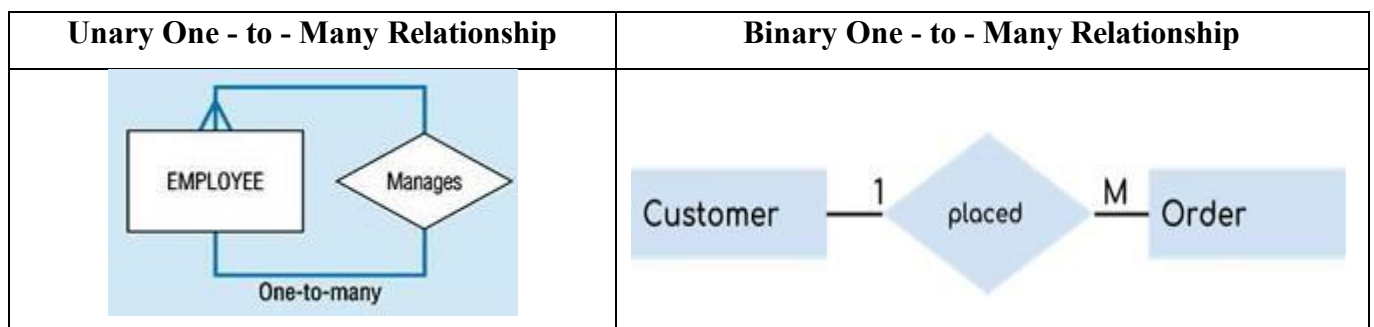
Ex:



2. One - to - Many Relationship(1:M):

It is used to create a relationship between two tables. Any single rows of the first table can be related to one or more rows of the second tables, but the rows of second tables can only relate to the only row in the first table.

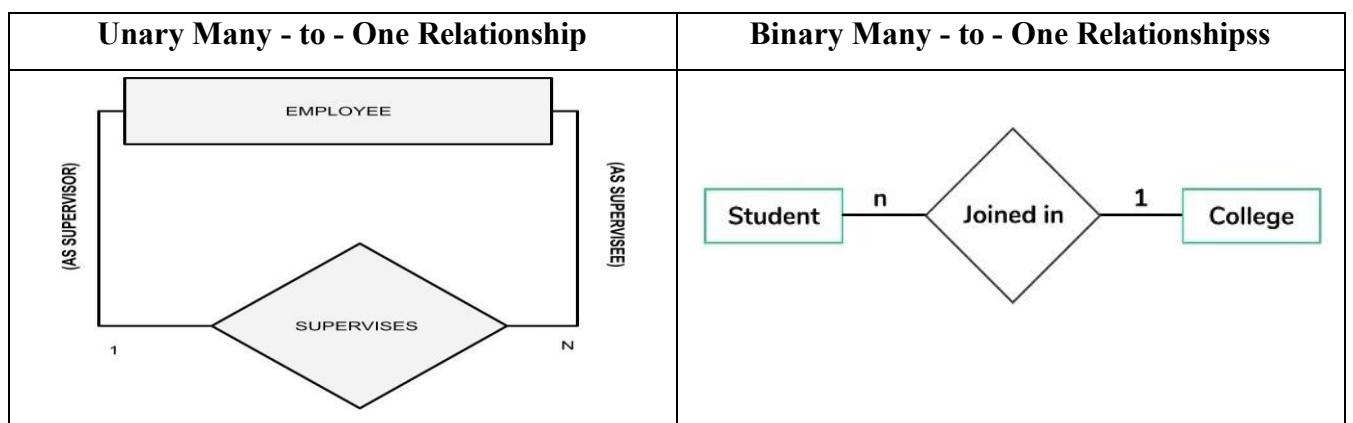
Ex:



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

It is reverse of one to many Relationship, Any single rows of the second table can be related to one or more rows of the first tables, but the rows of first tables can only relate to the only row in the second table.

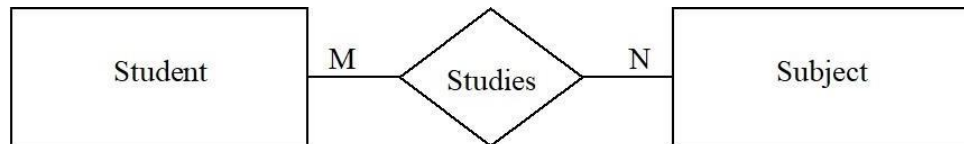
Ex:



4. Many - to - Many Relationship (M:N):

A many-to-many relationship exists between the tables if a single record of the first table is related to one or more records of the second table and a single record in the second table is related to one or more records of the first table.

Ex:



Relationship Participation:

Participation Constraints in database management refer to rules that determine the minimum and maximum participation of entities or relationships in a given relationship set.

There are two types of participation constraints

1. Total Participation
2. Partial Participation

1. Total Participation:

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

2. Partial Participation:

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



- Double line between the entity set “Student” and relationship set “Enrolled in” signifies total participation. It specifies that each student must be enrolled in at least one course
- Single line between the entity set “Course” and relationship set “Enrolled in” signifies partial participation. It specifies that there might exist some courses for which no enrollments are made.

Additional Features of the ER Model:

"Enhanced Entity-Relationship / Extended Entity-Relationship " (EER) data modeling, an extension of the traditional ER model. EER models include additional constructs (Features) to represent more complex relationships and constraints.

EER model is a conceptual (or semantic) data model, capable of describing the data requirements for a new information system in a direct and easy to understand graphical notation.

There are several additional features or concepts that can enhance its expressiveness and usefulness in database design

1. Super type or Subtype / Super class or Subclass
2. Generalization and Specialization
3. Category or Union
4. Aggregation

1. Super type or Subtype / Super class or Subclass:

A supertype is a generalized entity that represents a group of one or more related entities. It contains common attributes and relationships that are shared among its subtypes.

Or

Supertype is an entity type that has got relationship (parent to child relationship) with one or more subtypes and it contains attributes that are common to its subtypes..

In a supertype/subtype entity structure: The top-level entity is referred to as the parent entity or the supertype entity. Each lower-level entity is referred to as a child entity or a subtype entity.

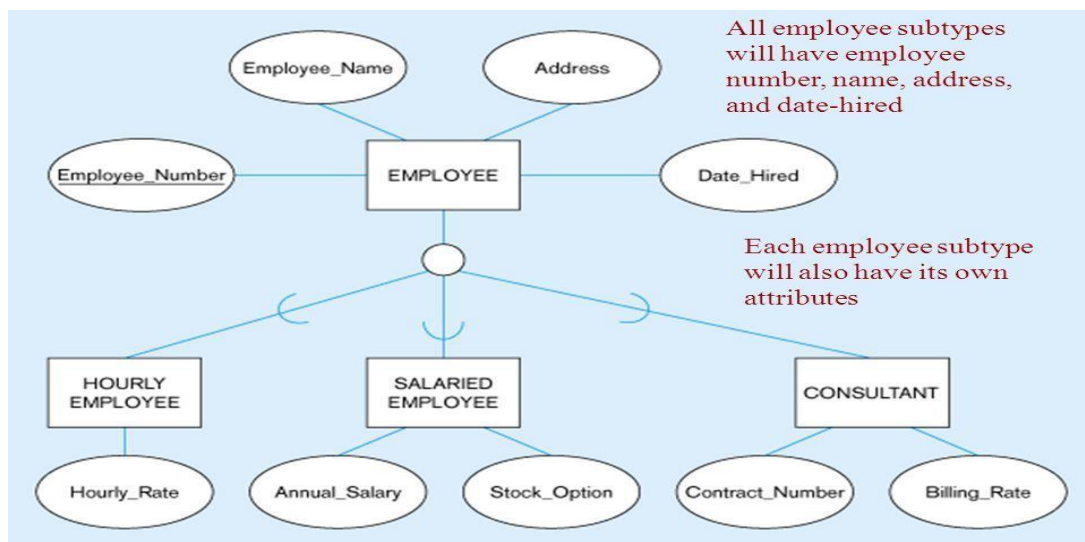


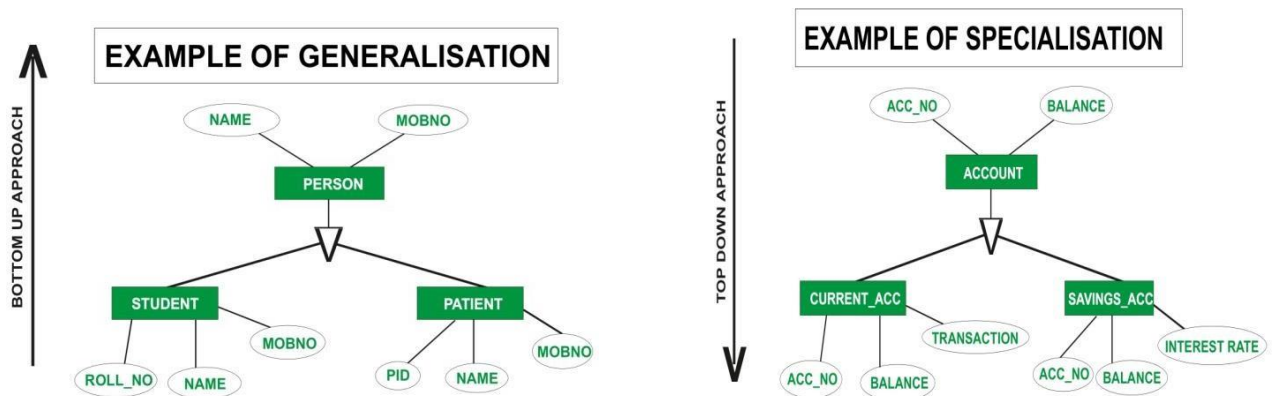
Figure – Employee supertype with three subtypes

- Super type and subtypes are connected through the connectors by using symbols (circle/triangle)
- On connecting line 'U' shape symbol indicates the subtype is subset of super type

2. Generalization and Specialization:

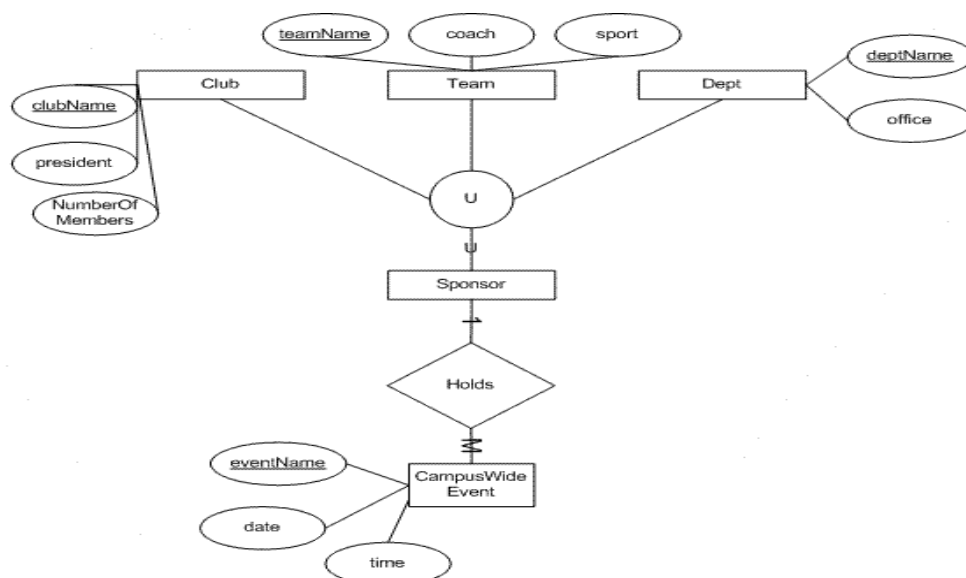
Generalization is the process of defining a general entity type from a set of more specialized entity types. It involves identifying common characteristics among multiple entity types and abstracting them into a higher-level entity type.

Specialization is the opposite of generalization. It involves creating one or more specialized entity types from a more general entity type. Specialization allows you to define specific characteristics or attributes for certain subsets of entities that are distinct from the general entity type.



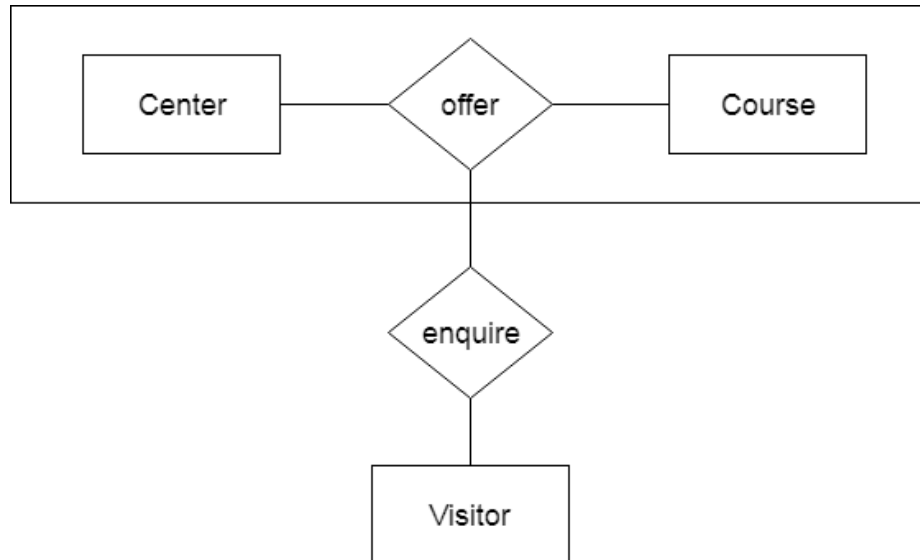
3. Category or Union:

EER model allows for the creation of a union type, which is a combination of two or more entity types. The union type can have attributes and relationships that are common to all the entity types that make up the union.



4. Aggregation:

Aggregation in DBMS (Database Management System) is the process of combining two or more entities to form a more meaningful new entity. When the entities do not make sense on their own, the aggregation process is used



DATABASE MANAGEMENT SYSTEM

R25 CSE – DATA SCIENCE

UNIT – II

INTRODUCTION TO THE RELATIONAL MODEL:

E.F. Codd proposed the relational Model to model data in the form of relations or tables. After designing the conceptual model of the Database using ER diagram, we need to convert the conceptual model into a relational model which can be implemented using any RDBMS language like Oracle SQL, MySQL, etc.

The relational model represents how data is stored in Relational Databases. A relational database consists of a collection of tables, each of which is assigned a unique name. Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE, and AGE shown in the table.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI		18

BASIC TERMINOLOGIES IN RELATIONAL MODEL:

- 1. Attribute:** An attribute refers to a specific characteristic or property of an entity or a relation (table). Attributes are the columns or fields within a table, representing the various types of information associated with the entities described by the table. Each attribute has a name and a data type, defining the kind of information it can hold (e.g., string, integer, date).

For example, in a table describing employees, attributes could include:

EmployeeID, FirstName, LastName, DateOfBirth.. etc

STUDENT

ROLL_NO	NAME	ADDRESS	PHONE	AGE
---------	------	---------	-------	-----

- 2. Tuple:** Each row in the relation is known as a tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18
---	-----	-------	------------	----

- 3. Relation :** A relation refers to a table that stores data in rows and columns. Each row in the table represents a record or tuple, while each column represents an attribute or field.
- 4. Relation Schema:** A relation schema defines the structure of the relation and represents the name of the relation with its attributes.

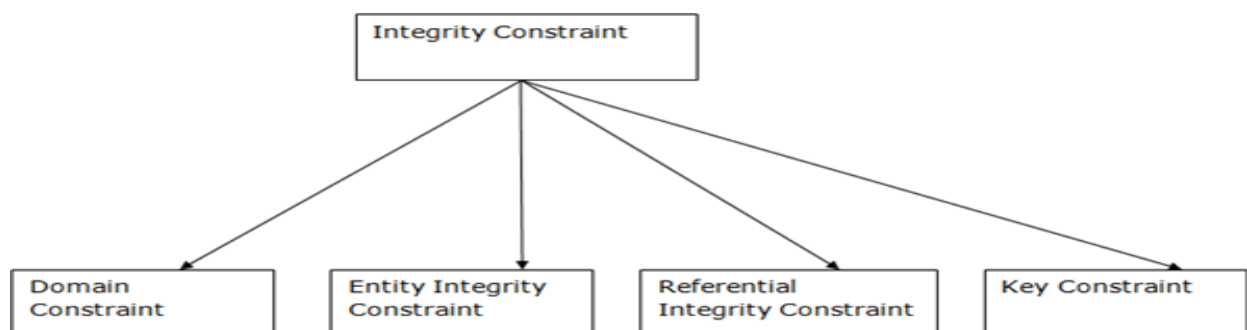
Ex: STUDENT (ROLL_NO, NAME, ADDRESS, PHONE, and AGE) is the relation schema for STUDENT.

5. **Relation Instance:** The set of tuples of a relation at a particular instance of time is called a relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is an insertion, deletion, or update in the database.
6. **Degree:** The number of attributes in the relation is known as the degree of the relation. The STUDENT relation defined above has degree 5
7. **Cardinality:** The number of tuples in a relation is known as cardinality. The STUDENT relation defined above has cardinality 4
8. **Column:** The column represents the set of values for a particular attribute. The column ROLL_NO is extracted from the relation STUDENT
9. **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by blank space. e.g.; PHONE of STUDENT having ROLL_NO 4 is NULL
10. **Domain:** A domain refers to the set of possible values that an attribute or column can hold within a relational database. It defines the data type and any constraints that apply to the values that can be stored in that attribute.
11. **Relation Key:** key is an attribute or combination of attributes, that are used to identify the rows uniquely or also help in identifying tables.

RELATIONAL INTEGRITY CONSTRAINTS

Relational integrity constraints are rules defined within a RDBMS to maintain the completeness, correctness, consistency and integrity of data stored in the database. These constraints enforce certain conditions or rules that must be satisfied by the data in order to maintain the quality and reliability of the database.

Here are some common types of relational integrity constraints:



1. Domain constraints:

- Domain constraints, also known as attribute constraints, are rules or conditions applied to individual attributes (columns) within a database table.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1 st	17
1001	Johnson	2 nd	24
1002	Leonardo	5 th	21
1003	Kate	3 rd	19
1004	Morgan	8 th	A

Not allowed. Because AGE is an integer attribute

2. Entity Integrity Constraint:

- Entity integrity constraints ensure that primary key columns do not contain null values. This ensures that each record in a table is uniquely identifiable and prevents incomplete or inconsistent data.
- An entity integrity constraint states that the primary key value cannot be null. This is because it becomes impossible to identify and distinguish rows within a table without a valid primary key value.

Example:

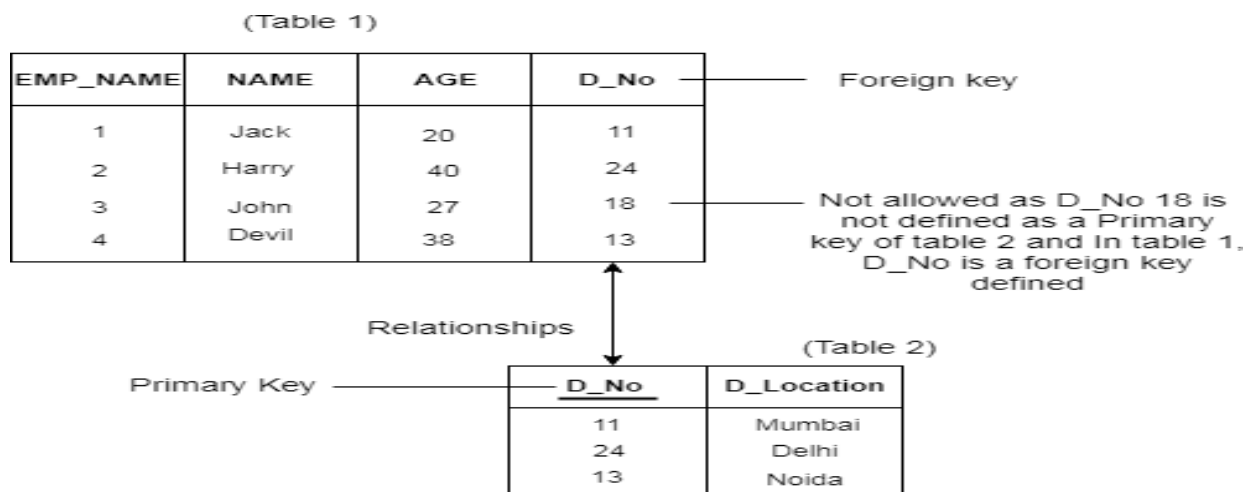
EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints:

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.



4. Key constraints

- Keys are the entity set that is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the **primary key**. A primary key can contain a unique and does not contain null values and duplicate values in the relational table.

ID	NAME	SEMENSTER	AGE
1000	Tom	1 st	17
1001	Johnson	2 nd	24
1002	Leonardo	5 th	21
1003	Kate	3 rd	19
1002	Morgan	8 th	22

Not allowed. Because all row must be unique

Types of keys:

Keys are one of the basic requirements of a relational database model. It is widely used to uniquely identify any record or row of data in the table.

Why we need keys in DBMS

- To identify each row of a table uniquely
- To maintain the data integrity
- To maintain the relationship between tables

Here are the types of keys commonly used in DBMS

1. Super key
2. Candidate Key
3. Primary Key
4. Alternate Key
5. Foreign Key
6. Composite Key
7. Compound key
8. Surrogate key

1. **Super key:** Super key is a general key, it is an attribute or combinations of attribute for unique identification of each row of table.

A super key is the **superset** of a key known as a Candidate key. It means a **candidate key** is obtained from a super key only

Ex. Emp_SSN, Emp_Id, Emp_name and Emp_email are super keys

Emp_SSN	Emp_Id	Emp_name	Emp_email
11051	01	John	john@email.com
19801	02	Merry	merry@email.com
19801	03	Riddle	riddle@email.com
41201	04	Cary	cary@email.com

So, from the above table, we conclude the following set of the sup

Set of super keys obtained

{ Emp_SSN }
{ Emp_Id }
{ Emp_email }
{ Emp_SSN, Emp_Id }
{ Emp_Id, Emp_name }
{ Emp_SSN, Emp_Id, Emp_email }
{ Emp_SSN, Emp_name, Emp_Id }

2. Candidate Key:

- A candidate key is a minimal subset of a super key
- The role of a candidate key is to identify a table row or column uniquely. Also, the value of a candidate key cannot be Null.
- Every table must have at least a single candidate key
- A table can have multiple candidate keys but only one primary key
- There can be more than one candidate key in a relationship

In the above example the candidate keys are

Candidate Keys :

Emp_SSN
Emp_Id
Emp_email

3. Primary Key:

The primary key refers to a column or a set of columns of a table that helps us identify all the records uniquely present in that table.

- Each table in a database should have a primary key.
- Primary keys must contain unique values and cannot contain NULL values.
- Primary key cannot contain duplicate values
- Primary key is always chosen from the possible candidate keys
- Primary key represented with underline (_____)

STUDENT_DETAILS		
Roll_no	Name	Marks
101	X	34
102	Y	46
103	Z	94

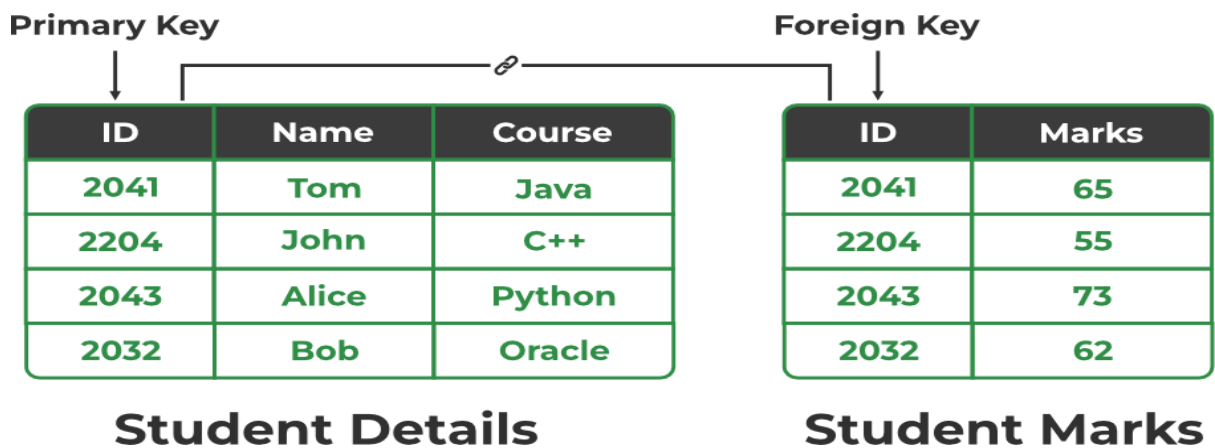
4. Alternate Key:

- The candidate key other than the primary key is called an alternate key
- An alternate is a secondary candidate key that is capable of identifying a row uniquely. However, such a key is not used as a primary key

Emp_SSN	Emp_Id	Emp_name	Emp_email
11051	01	John	john@email.com
19801	02	Merry	merry@email.com
19801	03	Riddle	riddle@email.com
41201	04	Cary	cary@email.com

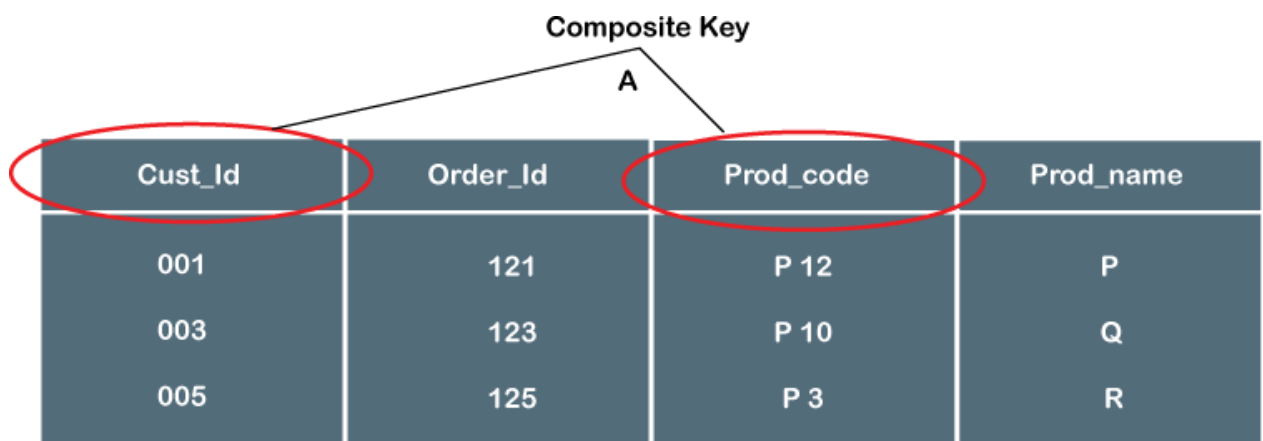
5. Foreign Key:

- A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
- We use a foreign key to establish relationships between two available tables.
- Foreign Key can be NULL as well as may contain duplicate tuples i.e. it need not follow uniqueness constraint.
- A foreign key helps us to maintain data as well as referential integrity
- Foreign key represented with dashed underline (_____)



6. Composite Key:

- Sometimes, a table might not have a single column/attribute that uniquely identifies all the records of a table. To uniquely identify rows of a table, a combination of two or more columns/attributes can be used.
- A composite key is a combination of two or more columns that together uniquely identify each record in a table



7. Compound key:

Any composite key which contains at least one attribute as the foreign key attribute, then that key is called compound key

8. Surrogate key:

If there is no any key attribute in a relation for unique identification then we can create a new key attribute which is used for unique identification that key attribute is called surrogate key

LOGICAL DATABASE DESIGN

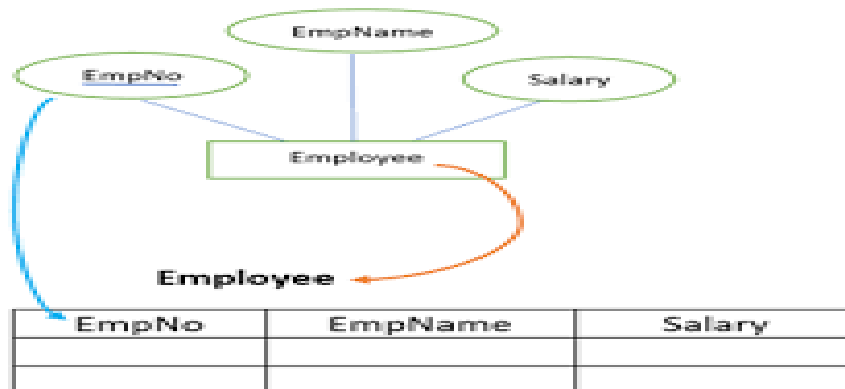
- The logical database design phase focuses on organizing data into tables and defining the relationships between these tables. Here's how logical database design is approached within the relational data model.
- After designing the ER diagram of system, we need to convert it to Relational models which can directly be implemented by any RDBMS like Oracle, MySQL etc.
- The goal of logical database design is to create well structured tables
- The tables will be able to store data about the entities in a non-redundant manner and foreign keys will be placed in the tables so that all the relationships among the entities will be supported.

ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. There are several processes and algorithms available to convert ER Diagrams into Relational model.

ER diagrams mainly comprise of

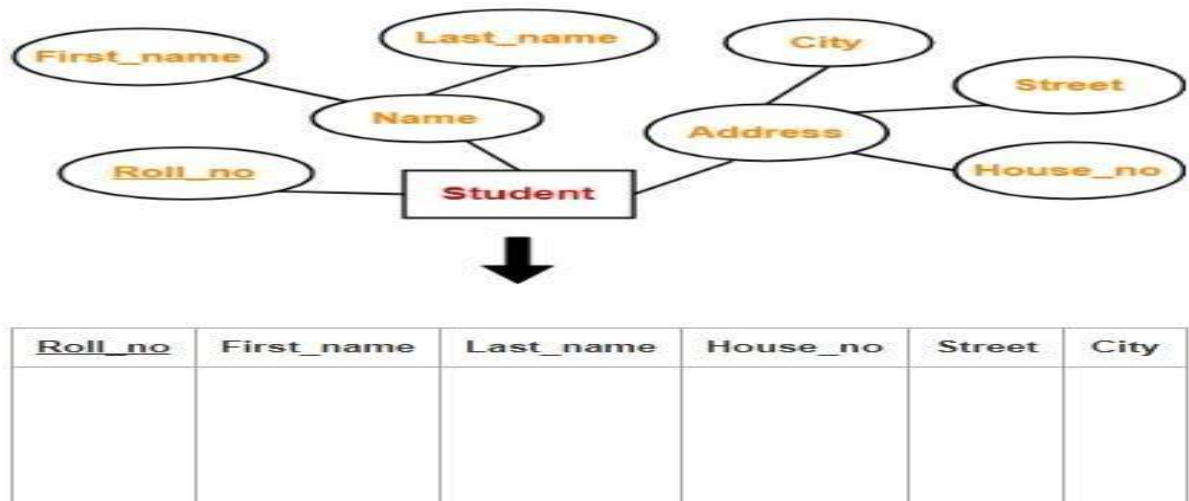
1. Mapping A Strong Entity into Relation With Simple Attributes:

- A strong entity set with only simple attributes will require only 1 table in relational model.
- Columns of the table will be the attributes of the entity set.
- Name of the relation should be entity name.
- The primary key of the table will be the key attribute of the entity set.



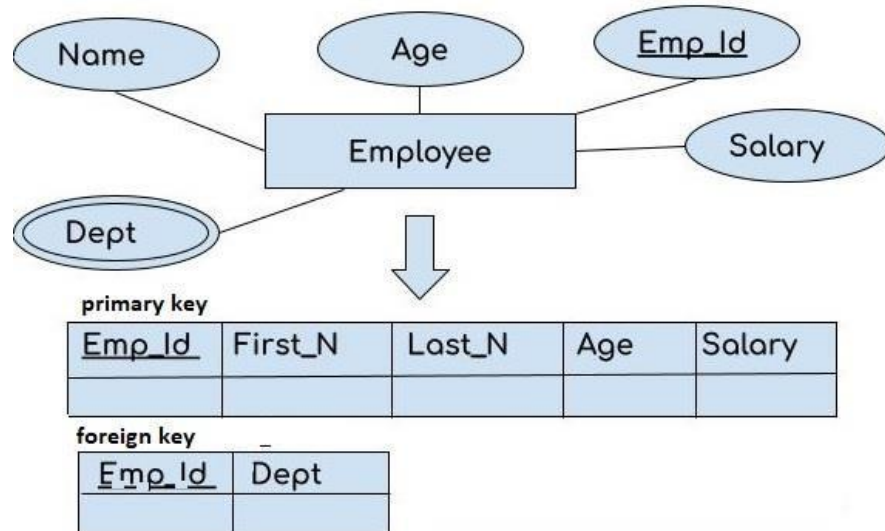
2. Mapping A Strong Entity into Relation With Composite Attributes:

- A strong entity set with any number of composite attributes will require only one table in relational model
- The components of the composite attribute will become the columns in the relationa (table) along with the simple attributes
- The primary key of the table will be the key attribute of the entity set



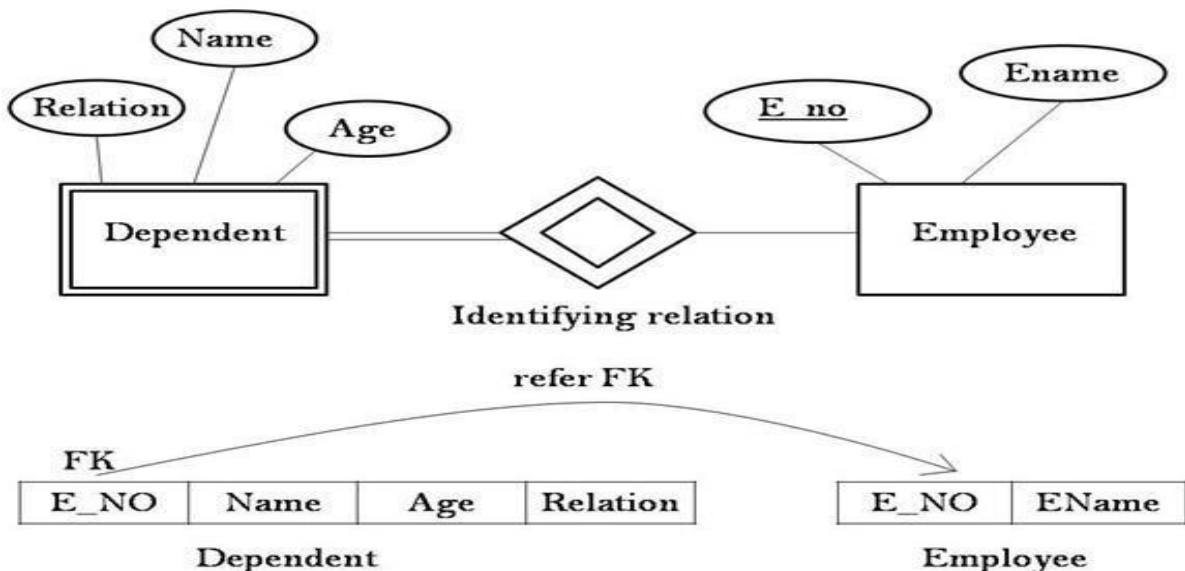
3. Mapping A Strong Entity into Relation With Multivalued Attributes:

- A strong entity set with any number of multi valued attributes will require two tables in relational model.
- One table will contain all the simple attributes with the primary key.
- Table two contain all multivalued attributes and the primary key of the first table as the foreign key



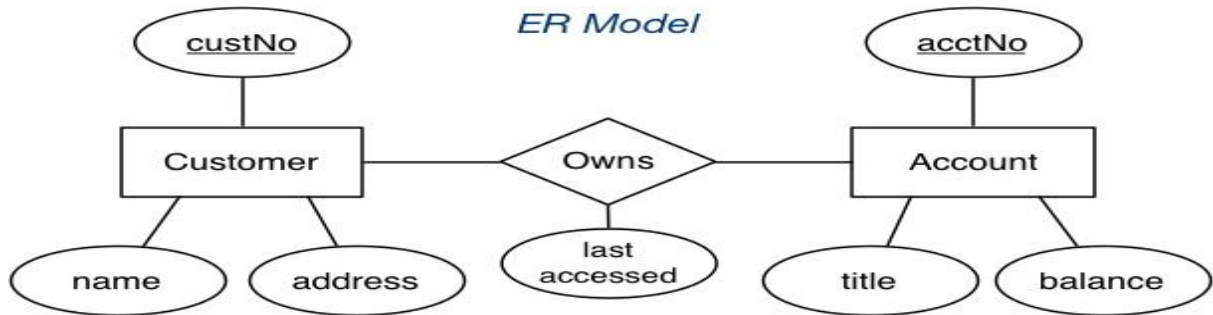
4. Mapping A Weak Entity into Relation

- Simply, a weak entity is an entity whose lifetime (related to the schema) depends on a strong entity.
- A weak entity set with any number of attributes will require two tables in relational model.
- One table will contain all the strong entity with the primary key.
- Table two contain all the attributes of weak entity and the primary key of the first table as the foreign key



5. Mapping A Relationship set into Relation

- A relationship set (with two entities) along with its attributes will require three tables in relational model.
- two tables will contain the entity sets with its attributes and primary keys.
- Table three contain all attributes of relationship set and the primary keys of the two table as the foreign keys



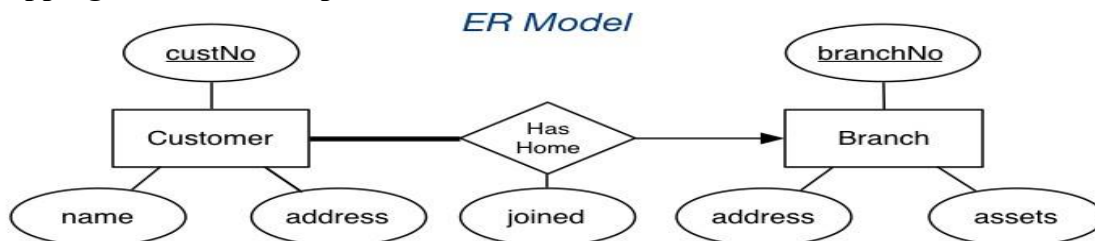
Relational Version

Customer	custNo	name	address
----------	---------------	------	---------

Account	acctNo	title	balance
---------	---------------	-------	---------

Owns	acctNo	custNo	lastAccessed
------	---------------	---------------	--------------

Mapping 1:N Relationships

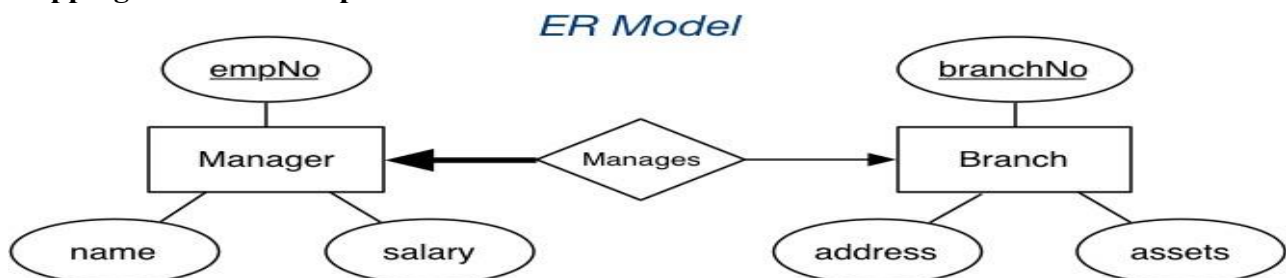


Relational Version

Customer	custNo	name	address	branchNo	joined
----------	---------------	------	---------	----------	--------

Branch	branchNo	address	assets
--------	-----------------	---------	--------

Mapping 1:1 Relationships

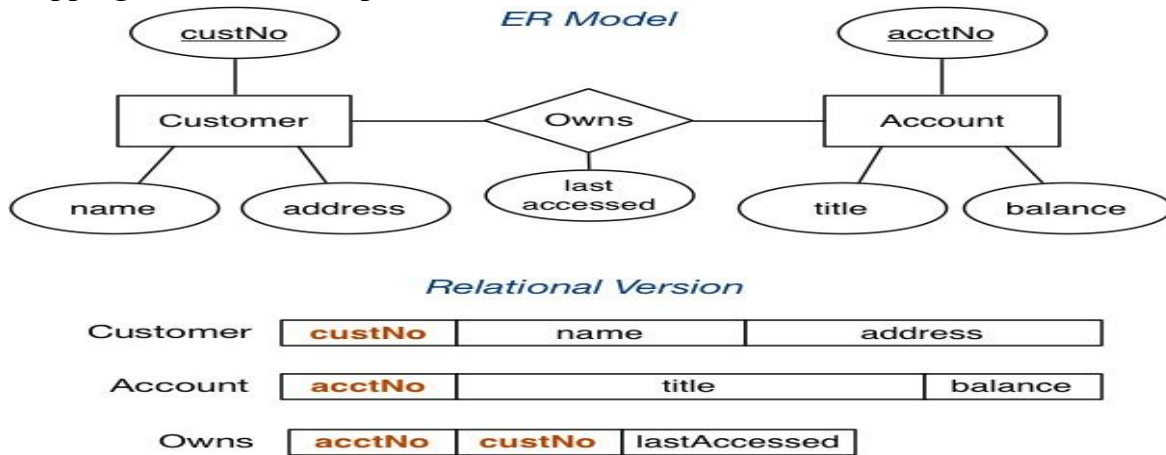


Relational Version

Manager	empNo	name	salary	branchNo
---------	--------------	------	--------	----------

Branch	branchNo	address	assets
--------	-----------------	---------	--------

Mapping N:M Relationships



Views

Introduction to views:

Views in SQL are considered as a virtual table. A view also has rows and columns like tables, but a view doesn't store physically on the disk like a table. View defines a customized query that retrieves data from one or more tables,

CREATE VIEWS:

We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain conditions.

Student_Detail

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

Student_Marks

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

Syntax:

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

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

Example:**Creating View from a single table**

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

Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM Student_Details
WHERE STU_ID < 4;
```

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

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Stephan	Delhi
Kathrin	Noida
David	Ghaziabad

Creating View from multiple tables:

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

Query:

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS
FROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

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

Destroying /Altering tables and views:

1. Deleting View:

A view can be deleted using the Drop View statement.

Syntax

```
DROP VIEW view_name;
```

Example:

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

Query:

```
DROP VIEW MarksView;
```

2. UPDATE VIEW

Rules to Update Views in SQL:

Certain conditions need to be satisfied to update a view. If any of these conditions are not met, the view can not be updated.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

Syntax:

```
UPDATE table_name
```

```
SET column1 = value1, column2 = value2, ...
```

```
WHERE condition;
```

Query:

```
UPDATE Student_Marks
```

```
SET age =22
```

```
WHERE NAME= David;
```

```
SELECT * FROM Student_Marks;
```

Student_Marks

STU_ID	NAME	MARKS	AGE
1	Stephan	97	19
2	Kathrin	86	21
3	David	74	22
4	Alina	90	20
5	John	96	18

CREATE OR REPLACE VIEW:

CREATE OR REPLACE VIEW used to create a new view, or if it already exists, to replace it.

Syntax:

CREATE OR REPLACE VIEW view_name **AS**

SELECT column1, column2,....

FROM tables

WHERE conditions;

Query:

CREATE VIEW DetailsView **AS**

SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.AGE,

FROM Student_Detail, Student_Mark

WHERE Student_Detail.NAME = Student_Marks.NAME;

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS	AGE
Stephan	Delhi	19
Kathrin	Noida	21
David	Ghaziabad	22
Alina	Gurugram	20

INSERTING ROWS INTO A VIEW

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Syntax:

INSERT INTO view_name (column1, column2, ...) **VALUES** (value1, value2, ...);

Query:

INSERT INTO DetailsView (NAME, ADDRESS, AGE) **VALUES** (ROHI, DSNR, 21);

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS	AGE
Stephan	Delhi	19
Kathrin	Noida	21
David	Ghaziabad	22
Alina	Gurugram	20
ROHI	DSNR	21

DELETING ROWS INTO A VIEW:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Syntax:

DELETE FROM view_name **WHERE** condition;

Query:

DELETE FROM DetailsView **WHERE** NAME=ROHI;

SELECT * FROM DetailsView;

Output:

NAME	ADDRESS	AGE
Stephan	Delhi	19
Kathrin	Noida	21
David	Ghaziabad	22
Alina	Gurugram	20

Advantages of View:

1. **Complexity:** Views help to reduce the complexity. Different views can be created on the same base table for different users.
2. **Security:** It increases the security by excluding the sensitive information from the view.
3. **Query Simplicity:** It helps to simplify commands from the user. A view can draw data from several different tables and present it as a single table.
4. **Consistency:** A view can present a consistent, unchanged image of the structure of the database. Views can be used to rename the columns without affecting the base table.
5. **Data Integrity:** If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets the specified integrity constraints.
6. **Storage Capacity:** Views take very little space to store the data.
7. **Logical Data Independence:** View can make the application and database tables to a certain extent independent.

Disadvantages of View:

The DML statements which can be performed on a view created using single base table have certain restrictions are:

1. You cannot INSERT if the base table has any not null column that do not appear in view.
2. You cannot INSERT or UPDATE if any of the column referenced in the INSERT or UPDATE contains group functions or columns defined by expression.
3. You can't execute INSERT, UPDATE, DELETE statements on a view if with read only option is enabled.
4. You can't be created view on temporary tables.
5. You cannot INSERT, UPDATE, DELETE if the view contains group functions GROUP BY, DISTINCT or a reference to a pseudo column rownum.
6. You can't pass parameters to the SQL server views.
7. You can't associate rules and defaults with views

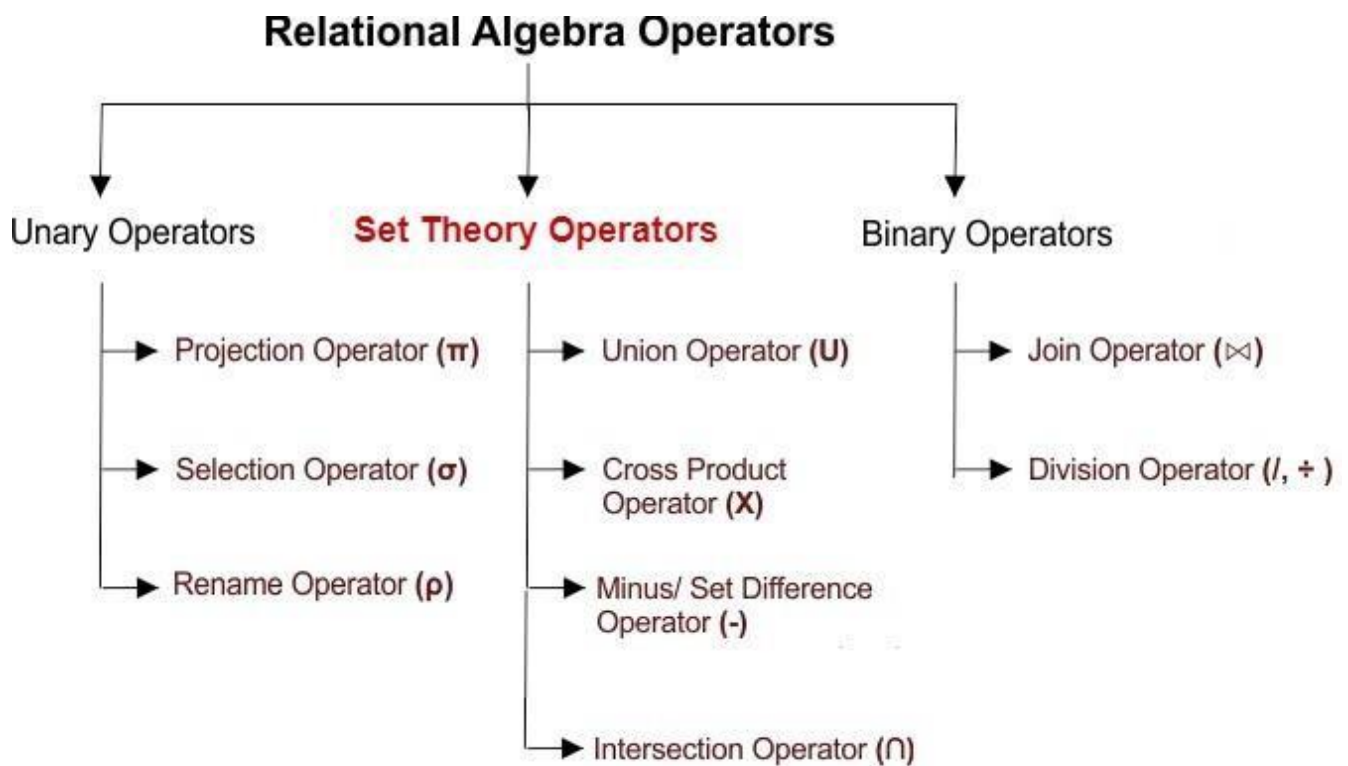
RELATIONAL ALGEBRA:

Relational algebra is a **procedural query language**, Relational algebra mainly provides a theoretical foundation for relational databases and SQL. Relational algebra works on **relational model**.

It provides a set of operations to perform various tasks on relations (tables) and their attributes (columns). They accept relations as their input and yield relations as their output.

It uses operators to perform queries.

The fundamental operations of relational algebra are as follows



UNARY OPERATORS

The unary operations in relational algebra act on one relation and result in another relation. The unary operations are selection, projection, and rename.

SELECTION OPERATOR (σ):

- Select operation chooses the subset of tuples from the relation that satisfies the given condition mentioned in the syntax of selection.
- It selects those rows or tuples from the table that satisfies the selection condition
- It works with rows(tuples)
- The selection operation is also known as horizontal partitioning

Syntax:

$\sigma_{\text{condition}}(\text{table name})$

Example

Consider the student table given below –

Regno	Branch	Section
1	CSE	A
2	ECE	B
3	CIVIL	B
4	IT	A

Now, to display all the records of student table, we will use the following command – $\sigma(\text{student})$

In addition to this, when we have to display all the records of CSE branch in student table, we will use

the following command – $\sigma_{\text{branch}=\text{cse}}(\text{student})$

Hence, the result will be as follows –

RegNo	Branch	Section
1	CSE	A

To display all the records in student tables whose regno>2, we will use the below mentioned command

– $\sigma_{\text{RegNo}>2}(\text{student})$

The output will be as follows –

RegNo	Branch	Section
3	CIVIL	B
4	IT	A

PROJECTION OPERATOR:

It displays the specific column of a table. It is denoted by pie (π). It is a vertical subset of the original relation. It eliminates duplicate tuples.

Syntax: $\pi_{\langle \text{attribute list} \rangle}(\text{Relation})$ (or) $\pi_{\langle \text{column list} \rangle}(\text{Relation})$

Example: Consider the student table:

Regno	Branch	Section
1	CSE	A
2	ECE	B
3	CIVIL	B
4	IT	A

To display regno column of student table, we can use the following command – $\Pi_{\text{regno}}(\text{student})$

RegNo
1
2
3
4

To display branch, section column of student table, use the following command –

$\Pi_{\text{branch,section}}(\text{student})$

Branch	Section
CSE	A
ECE	B
CIVIL	B
IT	A

To display regno, section of ECE students, use the following command –

$\Pi_{\text{regno,section}}(\sigma_{\text{branch=ECE}}(\text{student}))$

Output

Regno	Section
2	B

RENAME OPERATOR:

It is used to assign a new name to a relation and is denoted by ρ (rho).

Syntax: $\rho_{\text{new name}}(\text{tablename})$

Example: Suppose the following “Student” table

ID	Name	Subject	Age
1	Ali	Math	20
2	Ahmad	Science	21
3	Ali	Physics	22
4	Nauman	Chemistry	23

Table: Student

To change the name of the above table to “STD_Result” Table: Student

Query syntax: ρ STD_Result (Student)

ρ STD_Result (Student)			
ID	Name	Subject	Age
1	Ali	Math	20
2	Ahmad	Science	21
3	Ali	Physics	22
4	Nauman	Chemistry	23

Table: STD_Result

SET THEORY OPERATORS

Relational Algebra in DBMS These Set Theory operations are the standard mathematical operations on set. These operations are Binary operations that are, operated on 2 relations. These operations are used to merge 2 sets in various ways.

The set operation is mainly categorized into the following:

1. UNION OPERATOR ($\cup$)
2. INTERSECTION OPERATOR ($\cap$),
3. SET DIFFERENCE OR MINUS OPERATOR ($-$)
4. CARTESIAN PRODUCT OPERATOR ($\times$)

1. UNION OPERATOR ($\cup$):

Before we apply one of the 4 set operations on two relations, For a union operation to be valid, the following conditions must hold

- The two relations must have same type of tuples. This is also known as be Union compatibility (or Type compatibility).
- Duplicate tuples are eliminated automatically
- Attribute domains need to be compatible

Example

Consider the following tables

Table A		Table B	
column 1	column 2	column 1	column 2
A	B	E	F
C	D	A	B

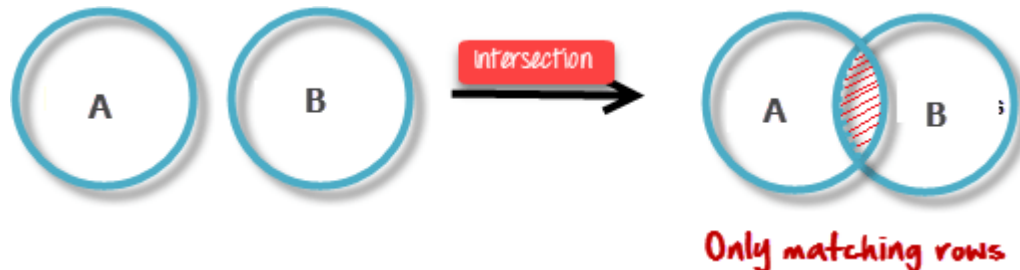
$A \cup B$ gives

Table $A \cup B$	
column 1	column 2
A	B
C	D
E	F

2. INTERSECTION OPERATOR ($\cap$):

An intersection is defined by the symbol $\cap$,

$A \cap B$ Defines a relation consisting of a set of all tuple that are in both A and B. However, A and B must be union-compatible.



Example: $A \cap B$

Table $A \cap B$	
column 1	column 2
A	B

3. SET DIFFERENCE OR MINUS OPERATOR ($-$):

Minus ($-$) Symbol denotes it. The result of $A - B$, is a relation which includes all tuples that are in A but not in B.

- The attribute name of A has to match with the attribute name in B.
- The two-operand relations A and B should be either compatible or Union compatible.
- It should be defined relation consisting of the tuples that are in relation A, but not in B.

Table $A - B$	
column 1	column 2
C	D

4. CARTESIAN PRODUCT OPERATOR ($\times$):

Cartesian product operator ($\times$) is an operation used to merge columns from two relations. Generally, a cartesian product is never a meaningful operation when it performs alone. However, it becomes meaningful when it is followed by other operations. It is also called Cross Product or Cross Join.

Example: $A \times B$

Table $A \times B$			
column 1	column 2	column 1	column 2
A	B	E	F
A	B	A	B
C	D	E	F
C	D	A	B

R25 DATABASE MANAGEMENT SYSTEM

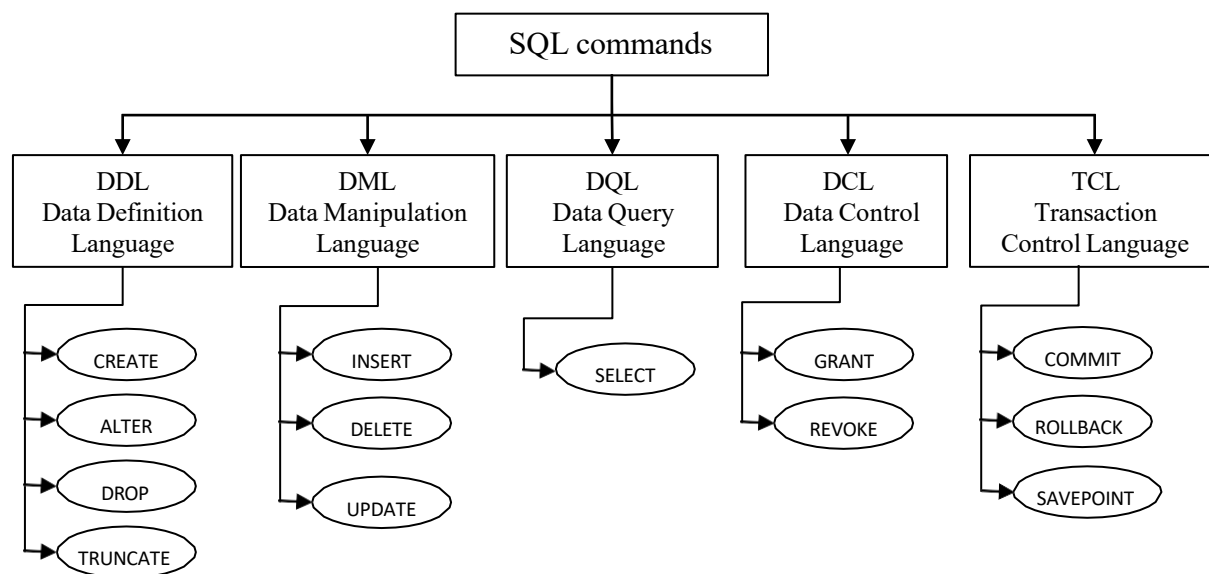
UNIT – III

SQL: QUERIES, CONSTRAINTS, TRIGGERS: form of basic SQL query, UNION, INTERSECT, and EXCEPT, Nested Queries, aggregation operators, NULL values, complex integrity constraints in SQL, triggers and active databases. **Schema Refinement:** Problems caused by redundancy, decompositions, problems related to decomposition, reasoning about functional dependencies, FIRST, SECOND, THIRD normal forms, BCNF, lossless join decomposition, multi-valued dependencies, FOURTH normal form, FIFTH normal form.

1. SQL COMMANDS

Structured Query Language (SQL) is the database language used to create a database and to perform operations on the existing database. SQL commands are instructions used to communicate with the database to perform specific tasks and queries with data. These SQL commands are categorized into five categories as:

- i. DDL: Data Definition Language
- ii. DML: Data Manipulation Language
- iii. DQL: Data Query Language
- iv. DCL : Data Control Language
- v. TCL : Transaction Control Language.



- i. **DDL(Data Definition Language) :** DDL or Data Definition Language consists of the SQL commands that can be used to **define the database schema**. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. The DQL commands are:

- **CREATE:** It is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
 - **DROP:** It is used to delete objects from the database.
 - **ALTER:** It is used to alter the structure of the database.
 - **TRUNCATE:** It is used to remove all records from a table, including all spaces allocated for the records are removed.
- ii. **DQL (Data Query Language):** DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get data from some schema relation based on the query passed to it. The DQL commands are:
- **SELECT** – is used to retrieve data from the database.
- iii. **DML (Data Manipulation Language):** The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. The DML commands are:
- **INSERT** – is used to insert data into a table.
 - **UPDATE** – is used to update existing data within a table.
 - **DELETE** – is used to delete records from a database table.
- iv. **DCL (Data Control Language):** DCL includes commands which mainly deal with the rights, permissions and other controls of the database system. The DCL commands are:
- **GRANT**-gives user's access privileges to database.
 - **REVOKE**-withdraw user's access privileges given by using the GRANT command.
- v. **TCL (transaction Control Language):** TCL commands deals with the transaction within the database. The TCL commands are:
- **COMMIT**– commits a Transaction.
 - **ROLLBACK**– rollbacks a transaction in case of any error occurs.
 - **SAVEPOINT**–sets a save point within a transaction.

2. DDL COMMANDS

DDL or Data Definition Language consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. The DQL commands are:

- i. **CREATE:** It is used to create the database or its objects like table, index, function, views, store procedure and triggers.

- a) **The ‘CREATE DATABASE’ Statement:** This statement is used to create a database.

Syntax: CREATE DATABASE Database_Name;

Example: CREATE DATABASE Employee;

It creates Employee database.

- b) **The ‘CREATE TABLE’ Statement:** This statement is used to create a table.

Syntax:

```
CREATE TABLE TableName (  
Column1 datatype(size) [column_constraint],  
Column2 datatype(size) [column_constraint],  
....  
ColumnN datatype(size) [column_constraint],  
[table_constraint]  
[,table_constraint]  
);
```

Note: The content in the square brackets indicates it is optional. If not required, you can skip it.

Column constraints

- **PRIMARY KEY** // Use only, If one column name as primary key.
- **NOT NULL** // It does not accept NULL value in that column.
- **DEFAULT value** // It store default value in that column, if no value is inserted
- **UNIQUE** // It allows to store only unique values in the column

Table constraints

- **PRIMARY KEY(column_name1, column_name2, ...)**
Use it, If one column name or multiple column names acts as primary key.
- **UNIQUE(column_name1, column_name2, ...)**
Use it, if one column name or multiple column names should contain unique values.
If multiple column names are used, then for each row, it consider values from all the columns mentioned to decide the uniqueness, but not column wise.
- **FOREIGN KEY (column_name1) REFERENCES other_table_name (column_name2)**
It is used to link data from one table to other table.
- **CHECK(condition)**
It does not allow inserting value(s), if the condition is not satisfied. The condition may also contain multiple column names.

Example 1: Creating table without any constraints

```
CREATE TABLE Employee_Info
(
    EmployeeID int,
    EmployeeName varchar(20),
    PhoneNumber numeric(10),
    City varchar(20),
    Country varchar(20)
);
```

Example 2: Using PRIMARY KEY and NOT NULL as column constraints

```
CREATE TABLE Departments
(
    DeptID int PRIMARY KEY,
    DeptName varchar(20) NOT NULL,
    Hod varchar(20),
    Location varchar(20)
);
```

Example 3: Using PRIMARY KEY, NOT NULL, UNIQUE and DEFAULT as column constraints and FOREIGN KEY as table constraint.

```
CREATE TABLE Students_Info
(
    HallTicketNo int PRIMARY KEY,
    Name varchar(20) NOT NULL,
    Mobile numeric(10) NOT NULL UNIQUE,
    DepartmentID int,
    City varchar(20) DEFAULT 'Hyderabad',
    FOREIGN KEY(DepartmentID) REFERENCES Departments (DeptID)
);
```

Example 4: Using NOT NULL, UNIQUE as column constraints and PRIMARY KEY and CHECK as table constraints.

```
CREATE TABLE Voter_list
(
    VoterID numeric(10),
    AdhaarNo numeric(12) NOT NULL UNIQUE,
    Name varchar(20) NOT NULL,
    Age int,
    Mobile numeric(10) UNIQUE,
    City varchar(20),
    PRIMARY KEY(VoterID),
    CHECK(AGE>18)
);
```

- c) **The ‘CREATE TABLE AS’ Statement:** You can also create a table from another existing table. The newly created table also contains data of existing table.

Syntax:

<pre>CREATE TABLE NewTableName AS(SELECT Column1, column2, ..., ColumnN FROM ExistingTableName WHERE [condition]);</pre>
--

Example:

```
CREATE TABLE ExampleTable AS ( SELECT EmployeeName, PhoneNumber
FROM Employee_Info );
```

ii. **DROP:** This statement is used to drop an existing table or a database.

- a) **The ‘DROP DATABASE’ Statement:** This statement is used to drop an existing database. When you use this statement, complete information present in the database will be lost.

Syntax: `DROP DATABASE DatabaseName;`

Example: `DROP DATABASE Employee;`

- b) **The ‘DROP TABLE’ Statement:** This statement is used to drop an existing table. When you use this statement, complete information present in the table will be lost.

Syntax: `DROP TABLE TableName;`

Example: `DROP TABLE Employee;`

iii. **TRUNCATE:** This command is used to delete the information present in the table but does not delete the table. So, once you use this command, your information will be lost, but not the table.

Syntax: `TRUNCATE TABLE TableName;`

Example: `TRUNCATE TABLE Employee_Info;`

iv. **ALTER:** This command is used to add, delete or modify column(s) in an existing table. It can also be used to rename the existing table and also to rename the existing column name.

- a) **The ‘ALTER TABLE’ with ADD column:** You can use this command to add a new column to the existing table.

Syntax:	<code>ALTER TABLE TableName ADD ColumnName Datatype;</code>
----------------	---

Example: Adding Blood Group column to the Employee_Info table

```
ALTER TABLE Employee_Info  
ADD BloodGroup varchar(10);
```

- b) **The ‘ALTER TABLE’ with DROP column:** You can use this command to remove a column from the existing table.

Syntax:	<code>ALTER TABLE TableName DROP ColumnName;</code>
----------------	---

Example: Removing Blood Group column from the Employee_Info table

```
ALTER TABLE Employee_Info  
DROP BloodGroup;
```

- c) **The ‘ALTER TABLE’ with MODIFY COLUMN:** This statement is used to change the data type or size of data type of an existing column in a table.

Syntax:	<code>ALTER TABLE TableName MODIFY COLUMN ColumnName Datatype;</code>
----------------	---

Example 1: Changing the size of column ‘*EmployeeName*’ in table ‘*Employee_info*’ from 20 to 30.

```
ALTER TABLE Employee_Info  
MODIFY EmployeeName varchar(30);
```

Example 2: Changing the data type of column ‘*EmployeeID*’ in the table ‘*Employee_info*’ from *int* to *char(10)*.

```
ALTER TABLE Employee_Info  
MODIFY EmployeeID char(10);
```

- d) **The ‘ALTER TABLE’ with CHANGE column name:** This statement is used to change the column name of an existing column in a table.

Syntax:	<code>ALTER TABLE TableName CHANGE COLUMN OldColumnName NewColumnName;</code>
----------------	---

Example 1: Changing the column name ‘*EmployeeName*’ to ‘*EmpName*’ in table ‘*Employee_info*’.

```
ALTER TABLE Employee_Info  
CHANGE COLUMN EmployeeName EmpName;
```

- e) **The ‘ALTER TABLE’ with RENAME table name:** This statement is used to change the table name in the database.

Syntax:	<code>ALTER TABLE OldTableName RENAME TO NewTableName;</code>
----------------	---

Example: Changing the table name from ‘*Employee_Info*’ to ‘*Employee_Data*’.

```
ALTER TABLE Employee_Info  
RENAME TO Employee_Data;
```

3. DML COMMANDS: The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. The DML commands are:

- i. **INSERT:** This statement is used to insert new record (row) into the table.

Syntax: INSERT INTO TableName[(Column1, Column2,..., ColumnN)] VALUES (value1, value2,..., valueN);
--

Example1 :

```
INSERT INTO Employee_Info( EmployeeID, EmployeeName, PhoneNumber, City,Country)  
VALUES ('06', 'Sanjana', '9921321141', 'Chennai', 'India');
```

Example2 : When inserting all column values as per their order in the table, you can omit column names.

```
INSERT INTO Employee_Info  
VALUES ('07', 'Sayantini','9934567654', 'Pune', 'India');
```

- ii. **DELETE:** This statement is used to delete the existing records in a table.

Syntax: DELETE FROM TableName WHERE Condition;
--

Example:

```
DELETE FROM Employee_Info  
WHERE EmployeeName='Preeti';
```

Note: *If where condition is not used in DELETE command, then all the rows data will be deleted. If used only rows which satisfies the condition are deleted.*

- iii. **UPDATE:** This statement is used to modify the record values already present in the table.

Syntax: UPDATE TableName SET Column1 = Value1, Column2 = Value2, ... [WHERE Condition];
--

Example:

```
UPDATE Employee_Info  
SET EmployeeName = 'Jhon', City= 'Ahmedabad'  
WHERE EmployeeID = 1;
```

Note: *If where condition is not used in UPDATE command, then in all the rows Employee Name changes to 'Jhon' and City name changes to 'Ahmedabad'. If used only rows which satisfies the condition are updated.*

- 4. DQL COMMAND:** The purpose of DQL Command is to get data from one or more tables based on the query passed to it.

- i. **SELECT:** This statement is used to select data from a database and the data returned is stored in a result table, called the **result-set**.

Syntax:	<pre>SELECT [DISTINCT] * / Column1,Column2,...ColumnN FROM TableName [WHERE search_condition] [GROUP BY column_names [HAVING search_condition_for_GROUP_BY] [ORDER BY column name ASC/DESC] ;</pre>
----------------	---

Example 1: `SELECT * FROM table_name;`

Example 2:

```
SELECT EmployeeID, EmployeeName
FROM Employee_Info;
```

The ‘SELECT with DISTINCT’ Statement: This statement is used to display only different unique values. It mean it will not display duplicate values.

Example : `SELECT DISTINCT PhoneNumber FROM Employee_Info;`

The ‘ORDER BY’ Statement: The ‘ORDER BY’ statement is used to sort the required results in ascending or descending order. The results are sorted in ascending order by default. Yet, if you wish to get the required results in descending order, you have to use the **DESC** keyword.

Example

```
/* Select all employees from the 'Employee_Info' table sorted by
City */
```

```
SELECT * FROM Employee_Info
ORDER BY City;
```

```
/*Select all employees from the 'Employee_Info' table sorted by
City in Descending order */
```

```
SELECT * FROM Employee_Info
ORDER BY City DESC;
```

```
/* Select all employees from the 'Employee_Info' table sorted by
City and EmployeeName. First it sort the rows as per city, then
sort by employee name */
```

```
SELECT * FROM Employee_Info
ORDER BY City, EmployeeName;
```

```
/* Select all employees from the 'Employee_Info' table sorted by
City in Descending order and EmployeeName in Ascending order: */
```

```
SELECT * FROM Employee_Info
ORDER BY City ASC, EmployeeName DESC;
```

AGGREGATE FUNCTIONS:

The SQL allows summarizing data through a set of functions called aggregate functions. The commonly used aggregate functions are: MIN(), MAX(), COUNT(), SUM(), AVG().

MIN() Function: The MIN function returns the smallest value of the selected column in a table.

Syntax:

```
SELECT MIN(ColumnName)
FROM TableName
WHERE Condition;
```

Example:

```
SELECT MIN(EmployeeID)
FROM Employee_Info;
```

MAX() Function: The MAX function returns the largest value of the selected column in a table.

Syntax:

```
SELECT MAX(ColumnName)
FROM TableName
WHERE Condition;
```

Example:

```
SELECT MAX(Salary) AS LargestFees
FROM Employee_Salary;
```

COUNT() Function: The COUNT function returns the number of rows which match the specified criteria.

Syntax:

```
SELECT COUNT(ColumnName)
FROM TableName
WHERE Condition;
```

Example:

```
SELECT COUNT(EmployeeID)
FROM Employee_Info;
```

SUM() Function: The SUM function returns the total sum of a numeric column that you choose.

Syntax:

```
SELECT SUM(ColumnName)
FROM TableName
WHERE Condition;
```

Example:

```
SELECT SUM(Salary)
FROM Employee_Salary;
```

AVG() Function: The AVG function returns the average value of a numeric column that you choose.

Syntax: SELECT AVG (ColumnName)
 FROM TableName
 WHERE Condition;

Example:

```
SELECT AVG(Salary)
FROM Employee_Salary;
```

The ‘GROUP BY’ Statement: This ‘GROUP BY’ statement is used with the aggregate functions to group the result-set by one or more columns.

Example:

```
-- To list the number of employees from each city.
```

```
SELECT COUNT(EmployeeID), City
FROM Employee_Info
GROUP BY City;
```

The ‘HAVING’ Clause: The ‘HAVING’ clause must be used SQL along with GROUP BY clause only. It is similar to the **WHERE** clause.

Example

```
/*       To list the number of employees in each city. The employees  
should be sorted high to low and only those cities must be included who have  
more than 5 employees:*/
```

```
SELECT COUNT(EmployeeID), City
FROM Employee_Info
GROUP BY City
HAVING COUNT(EmployeeID) > 2;
```

Operators in SQL:

The different set of operators available in SQL are as follows:

- Arithmetic operators
- Bitwise operators
- Comparison operator
- Compound operator
- Logical operator

Let us look into each one of them, one by one.

Arithmetic Operators:

Operator	Description
%	Modulus [A % B]
/	Division [A / B]
*	Multiplication [A * B]
–	Subtraction [A – B]
+	Addition [A + B]

Bitwise Operators:

Operator	Description
^	Bitwise Exclusive OR (XOR) [A ^ B]
	Bitwise OR [A B]
&	Bitwise AND [A & B]

Comparison Operators:

Operator	Description
<>	Not Equal to [A <> B]
<=	Less than or equal to [A <= B]
>=	Greater than or equal to [A >= B]
<	Less than [A < B]
>	Greater than [A > B]
=	Equal to [A = B]

Compound Operators:

Operator	Description
*=	Bitwise OR equals [A = B]
^-=	Bitwise Exclusive equals [A ^= B]
&=	Bitwise AND equals [A &= B]
%=	Modulo equals [A %= B]
/=	Divide equals [A /= B]
*=	Multiply equals [A *= B]
-=	Subtract equals [A -= B]
+=	Add equals [A += B]

Logical Operators: The Logical operators present in SQL are as follows: AND, OR, NOT, BETWEEN, LIKE, IN, EXISTS, ALL, ANY.

AND Operator: This operator is used to filter records that rely on more than one condition. This operator displays the records, which satisfy all the conditions separated by AND, and give the output TRUE.

Syntax:

```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition1 AND Condition2 AND Condition3 ...;
```

Example:

```
SELECT * FROM Employee_Info
WHERE City='Mumbai' AND City='Hyderabad';</pre>
```

OR Operator: This operator displays all those records which satisfy any of the conditions separated by OR and give the output TRUE.

Syntax:

```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE Condition1 OR Condition2 OR Condition3 ...;
```

Example:

```
SELECT * FROM Employee_Info
WHERE City='Mumbai' OR City='Hyderabad';
```

NOT Operator: The NOT operator is used, when you want to display the records which do not satisfy a condition.

Syntax:

```
SELECT Column1, Column2, ..., ColumnN
FROM TableName
WHERE NOT Condition;
```

Example:

```
SELECT * FROM Employee_Info
WHERE NOT City='Mumbai';
```

NOTE: You can also combine the above three operators and write a query as follows:

```
SELECT * FROM Employee_Info
WHERE NOT Country='India' AND (City='Bangalore' OR City='Hyderabad');
```

BETWEEN Operator: The BETWEEN operator is used, when you want to select values within a given range. Since this is an inclusive operator, both the starting and ending values are considered.

Syntax:

```
SELECT ColumnName(s)
FROM TableName
WHERE ColumnName BETWEEN Value1 AND Value2;
```

Example:

```
SELECT * FROM Employee_Salary
WHERE Salary BETWEEN 40000 AND 50000;
```

LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column of a table. There are mainly two wildcards that are used in conjunction with the LIKE operator:

- **%** : It is used to matches 0 or more character.
- **_** : It is used to matches exactly one character.

Syntax

```
SELECT ColumnName(s)
FROM TableName
WHERE ColumnName LIKE pattern;
```

Refer to the following table for the various patterns that you can mention with the LIKE operator.

Like Operator Condition	Description
WHERE CustomerName LIKE 'v%'	Finds any values that start with “v”
WHERE CustomerName LIKE '%v'	Finds any values that end with “v”
WHERE CustomerName LIKE '%and%'	Finds any values that have “and” in any position
WHERE CustomerName LIKE '_q%'	Finds any values that have “q” in the second position.
WHERE CustomerName LIKE 'u_%_ %'	Finds any values that start with “u” and are at least 3 characters in length
WHERE ContactName LIKE 'm%a'	Finds any values that start with “m” and end with “a”

Example:

```
SELECT * FROM Employee_Info
WHERE EmployeeName LIKE 'S%';
```

IN Operator: This operator is used for multiple OR conditions. This allows you to specify multiple values in a WHERE clause.

Syntax:

```
SELECT ColumnName(s)
FROM TableName
WHERE ColumnName IN (Value1,Value2...);
```

Example:

```
SELECT * FROM Employee_Info
WHERE City IN ('Mumbai', 'Bangalore', 'Hyderabad');
```

NOTE: You can also use IN while writing Nested Queries.

EXISTS Operator: The EXISTS operator is used to test if a record exists or not.

Syntax: SELECT ColumnName(s)
 FROM TableName
 WHERE EXISTS
 (SELECT ColumnName FROM TableName WHERE condition);

Example:
SELECT City
FROM Employee_Info
WHERE EXISTS (SELECT City
 FROM Employee_Info
 WHERE EmployeeId = 05 AND City = 'Kolkata');

ALL Operator: The ALL operator is used with a WHERE or HAVING clause and returns TRUE if all of the subquery values meet the condition.

Syntax: SELECT ColumnName(s)
 FROM TableName
 WHERE ColumnName operator ALL
 (SELECT ColumnName FROM TableName WHERE condition);

Example:
SELECT EmployeeName
FROM Employee_Info
WHERE EmployeeID = ALL (SELECT EmployeeID
 FROM Employee_Info
 WHERE City = 'Hyderabad');

ANY Operator: Similar to the ALL operator, the ANY operator is also used with a WHERE or HAVING clause and returns true if any of the subquery values meet the condition.

Syntax: SELECT ColumnName(s)
 FROM TableName
 WHERE ColumnName operator ANY
 (SELECT ColumnName FROM TableName WHERE condition);

Example:
SELECT EmployeeName
FROM Employee_Info
WHERE EmployeeID = ANY (SELECT EmployeeID
 FROM Employee_Info
 WHERE City = 'Hyderabad' OR City = 'Kolkata');

Aliases Statement: Aliases are used to give a column / table a temporary name and only exists for duration of the query.

Syntax: /* Alias Column Syntax. Instead of displaying the column name used in the table, it display alias name. */

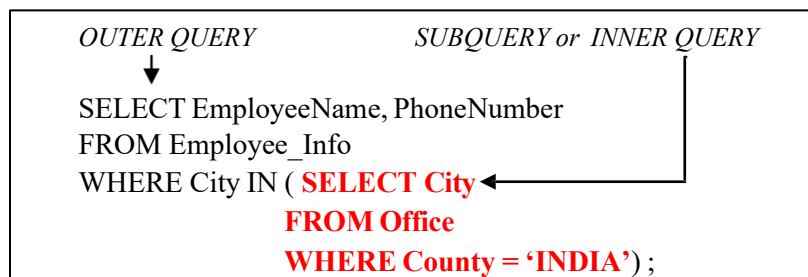
```
SELECT ColumnName AS AliasName  
FROM TableName;
```

Example:

```
SELECT EmployeeID AS ID, EmployeeName AS EmpName  
FROM Employee_Info;
```

5. NESTED QUERIES

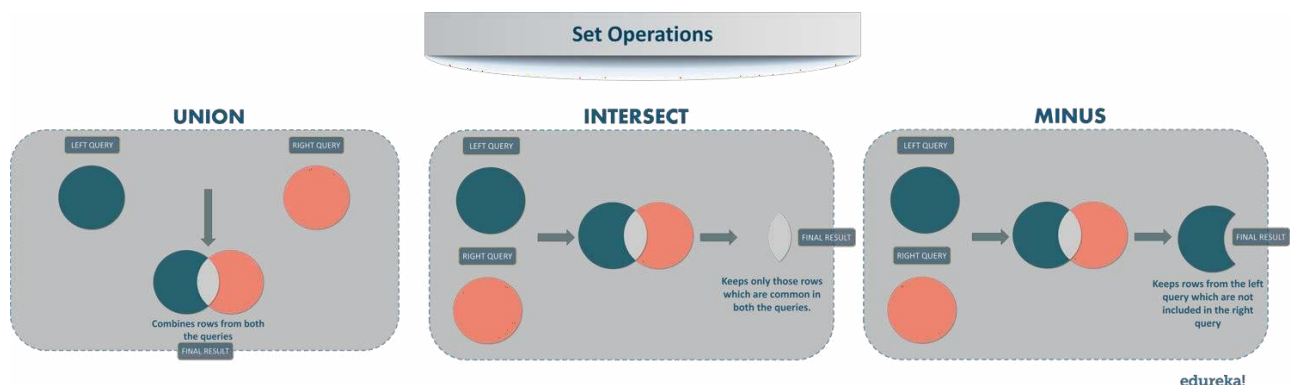
Nested queries are those queries which have an outer query and inner subquery. So, basically, the subquery is a query which is nested within another query.



First the inner query gets executed and the result will be used to execute the outer query.

6. SET OPERATIONS: UNION, INTERSECT, EXCEPT

There are mainly three set operations: UNION, INTERSECT, EXCEPT. You can refer to the image below to understand the set operations in SQL.



- i. **UNION:** This operator is used to combine the result-set of two or more SELECT statements.

Syntax:	<pre>SELECT ColumnName(s) FROM Table1 WHERE condition UNION SELECT ColumnName(s) FROM Table2 WHERE condition;</pre>
----------------	---

- ii. **INTERSECT:** This clause used to combine two SELECT statements and return the intersection of the data-sets of both the SELECT statements.

Syntax: SELECT ColumnName(s) FROM Table1 WHERE condition
 INTERSECT
 SELECT ColumnName(s) FROM Table2 WHERE condition;

- iii. **EXCEPT:** This operator returns those tuples that are returned by the first SELECT operation, and are not returned by the second SELECT operation.

Syntax: SELECT ColumnName(s) FROM Table1 WHERE condition
 EXCEPT
 SELECT ColumnName(s) FROM Table2 WHERE condition;

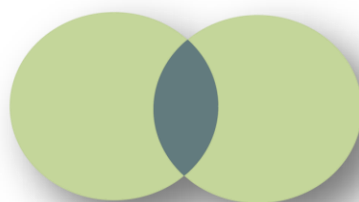
Note: UNION, INTERSECT or EXCEPT operations are possible if and only if first SELECT query and second SELECT query produces same no of columns in same order, same column names and data type. Otherwise it gives an error.

7. JOINS

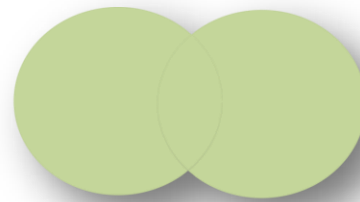
JOINS are used to combine rows from two or more tables, based on a related column between those tables. The following are the types of joins:

- **INNER JOIN:** This join returns those records which have matching values in both the tables.
- **FULL JOIN:** This join returns all those records which either have a match in the left or the right table.
- **LEFT JOIN:** This join returns records from the left table, and also those records which satisfy the condition from the right table.
- **RIGHT JOIN:** This join returns records from the right table, and also those records which satisfy the condition from the left table.

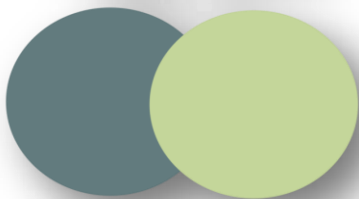
Refer to the image below.



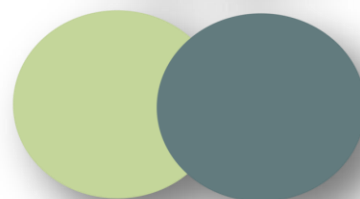
INNER JOIN



FULL JOIN



LEFT JOIN



RIGHT JOIN edureka!

Let's consider the below Technologies and the Employee_Info table, to understand the syntax of joins.

Employee_Info

EmployeeID	EmployeeName	PhoneNumber	City	Country
01	Shravya	9898765612	Mumbai	India
02	Vijay	9432156783	Delhi	India
03	Preeti	9764234519	Bangalore	India
04	Vijay	9966442211	Hyderabad	India
05	Manasa	9543176246	Kolkata	India

Technologies

TechID	EmpID	TechName	ProjectStartDate
1	01	DevOps	04-01-2019
2	03	Blockchain	06-07-2019
3	04	Python	01-03-2019
4	06	Java	10-10-2019

INNER JOIN or EQUI JOIN: This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the SQL query. This join is used mostly. NATURAL JOIN is a type INNER JOIN. We can also use it. It also gives same result.

Syntax

```
SELECT ColumnName(s)
FROM Table1
INNER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example

```
SELECT T.TechID, E.EmployeeID, E.EmployeeName
FROM Technologies T
INNER JOIN Employee_Info E ON T.EmpID = E.EmpID;
```

TechID	EmployeeID	EmployeeName
1	01	Shravya
2	03	Preeti
3	04	Vijay

FULL OUTER JOIN: The full outer join returns a result-set table with the **matched data** of two table then remaining rows of both **left** table and **right** table with missing values are filled with NULL values.

Syntax

```
SELECT ColumnName(s)
FROM Table1
FULL OUTER JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example

```
SELECT E.EmployeeID, E.EmployeeName, T.TechID
FROM Employee_Info E
FULL OUTER JOIN Technologies T ON E.EmployeeID=T.EmployeeID;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
02	Vijay	NULL
03	Preeti	2
04	Vijay	3
05	Manasa	NULL
06	NULL	4

LEFT JOIN: The left outer join returns a result-set table with the **matched data** from the two tables and then the remaining rows of the **left** table with null for the **right** table's columns.

Syntax:

```
SELECT ColumnName(s)
FROM Table1
LEFT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

Example:

```
SELECT E.EmployeeId, E.EmployeeName, T.TechID
FROM Employee_Info E
LEFT JOIN Technologies T ON E.EmployeeID = T.EmpIDID ;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
02	Vijay	NULL
03	Preeti	2
04	Vijay	3
05	Manasa	NULL

RIGHT JOIN: The right outer join returns a result-set table with the matched data from the two tables being joined, then the remaining rows of the right table and null for the remaining left table's columns.

Syntax:

```
SELECT ColumnName(s)
FROM Table1
RIGHT JOIN Table2 ON Table1.ColumnName = Table2.ColumnName;
```

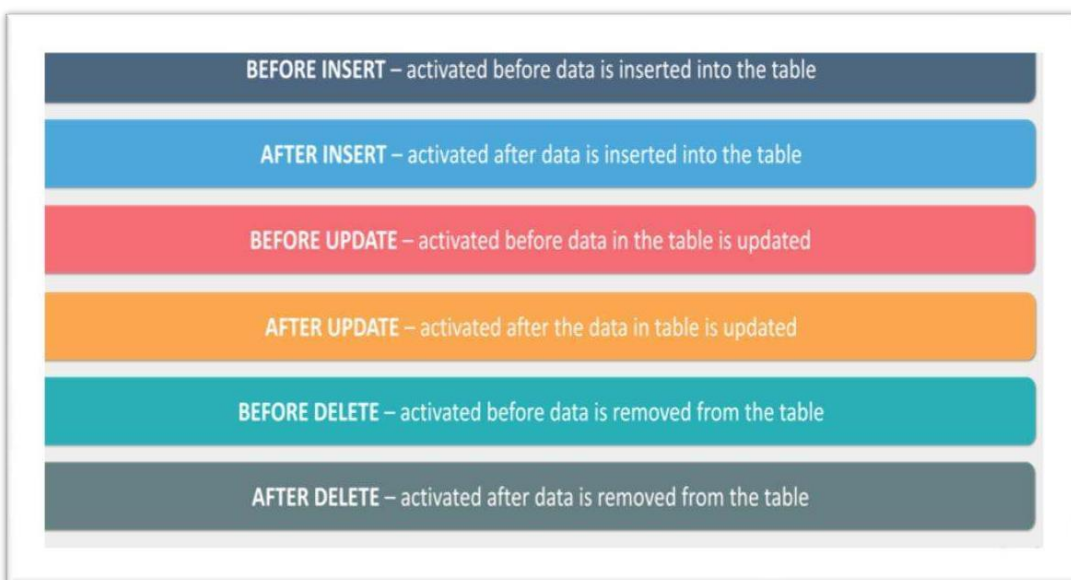
Example:

```
SELECT E.EmployeeId, E.EmployeeName, T.TechID
FROM Employee_Info E
RIGHT JOIN Technologies T ON E.EmployeeID = T.EmpIDID ;
```

EmployeeID	EmployeeName	TechID
01	Shravya	1
03	Preeti	2
04	Vijay	3
NULL	NULL	4

8. TRIGGERS

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated. So, a **trigger** can be invoked either **BEFORE** or **AFTER** the data is changed by **INSERT**, **UPDATE** or **DELETE** statement. Refer to the image below.



Syntax:

```
CREATE TRIGGER [TriggerName]
[BEFORE | AFTER]
{INSERT | UPDATE | DELETE}
on [TableName]
[FOR EACH ROW]
[TriggerBody]
```

Explanation of syntax:

- create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
- [before | after]: This specifies when the trigger will be executed.
- {insert | update | delete}: This specifies the DML operation.
- on [table_name]: This specifies the name of the table associated with the trigger.
- [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- [trigger_body]: This provides the operation to be performed as trigger is fired .

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

EXAMPLE:

```
CREATE TRIGGER nb BEFORE INSERT ON accounts FOR EACH ROW      /* Event */
Begin
    IF :NEW.bal < 0 THEN                                         /*Condition*/
        DBMS_OUTPUT.PUT_LINE('BALANCE IS NEGATIVE..');        /*Action*/
    END IF;
End;
```

A trigger called 'nb' is created to alert the user when inserting account details with negative balance value in to accounts table. Before inserting, the trigger is activated if the condition is true. When a trigger activated, the action part of the trigger is get executed.

9. NORMALIZATION

- Normalization is the process of minimizing the redundancy from a relation or set of relations.
- It is used to eliminate the Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- Normalization is done with the help of different normal form.

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form. In software industry, they are using only up to third normal form and sometimes Boyce-Codd Normal Form.

The Problem of redundancy

Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized. Redundancy leads the following problems.

- ◆ **Wastage of Memory:** Disk space is wasted due to storing same copy multiple times.
- ◆ **Storage cost increases:** When multiple copies of same data is stored, need more disk space and storage cost increases.
- ◆ **Update anomaly:** When Address of student is stored at several places; a change in the address must be made in all the places. Changing the address at some places and leaving other places leads to inconsistency problem.
- ◆ **Insertion Anomaly:** The nature of a database may be such that it is not possible to add a required piece of data unless another piece of unavailable data is also added. For example, a library database cannot store the details of a new student until that student has taken atleast one book from the library.
- ◆ **Deletion Anomaly:** When some data is deleted, it also deletes other data automatically. For example, deleting a book details from a library database, it also delete the student details who have taken the book previously.

10. 1NF (FIRST NORMAL FORM)

A relation (table) is said to be in first normal form if and only if:

- Each table cell contains only atomic values (single value).
- Each record needs to be uniquely identified by the primary key.

1NF Example:

HTNO	FIRST NAME	LAST NAME	MOBILE
501	Jhansi	Rani	9999988888 7777799999
502	Ajay	Kumar	8888888881 7897897897
503	Priya	Verma	9898989898

The above table is not in 1NF because **501** and **502** is having two values in mobile column. If we add a new column as *alternative mobile number* to the above table, then for 503 *alternative mobile number* is NULL. Moreover, if a student has 'n' mobile numbers, then adding 'n' extra column is meaningless. It is better to add extra rows. If we add extra row for each 501 and 502 then the table looks like

HTNO	FIRST NAME	LAST NAME	MOBILE
501	Jhansi	Rani	9999988888
501	Jhansi	Rani	7777799999
502	Ajay	Kumar	8888888881
502	Ajay	Kumar	7897897897
503	Priya	Verma	9898989898

But the above table violates primary key constraint. Therefore instead of adding either columns or rows, the best solution is to split the table into two tables as shown below. If we do as shown below, if a student having 'n' number of mobile numbers also can be added.

HTNO	FIRST NAME	LAST NAME
501	Jhansi	Rani
502	Ajay	Kumar
503	Priya	Verma

HTNO	MOBILE
501	9999988888
501	7777799999
502	8888888881
502	7897897897
503	9898989898

11. 2NF (SECOND NORMAL FORM)

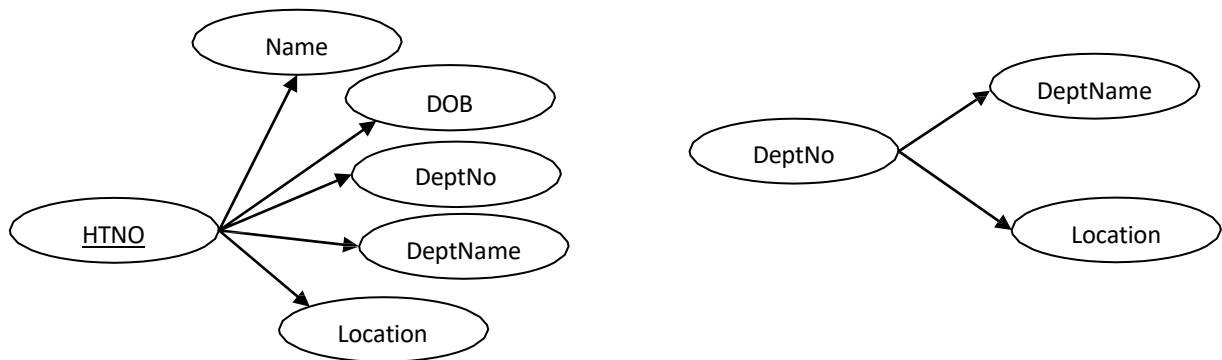
A relation is said to be in 2-NF if and only if

- It should be in 1-NF (First Normal Form)
- There should not be any partial functional dependencies

2NF Example:

<i>HTNO</i>	<i>Name</i>	<i>DOB</i>	<i>DeptNo</i>	<i>DeptName</i>	<i>Location</i>
501	Jhansi	30-10-1998	05	CSE	A-Block
502	Ajay	24-12-1999	05	CSE	A-Block
410	Priya	12-03-2000	04	ECE	B-Block
120	Rahul	30-10-1998	01	CIVIL	C-Block
415	Smitha	18-06-1999	04	ECE	B-Block

The above table is not in 2NF because there exist partial function dependencies. ***HTNO*** is a key attribute in the above table. If every non-key attribute fully dependent on key attribute, then we say it is fully functional dependent. Consider the below diagram. ***{Name, DOB, DeptNo, DeptName, Location}*** depends on ***HTNO***. But ***{DeptName, Location}*** also depends on ***DeptNo***.



It is clear that ***DeptName*** and ***Location*** not only depends upon ***HTNO*** but also on ***DeptNo***. So, there exists partial function dependency. This partial functional dependency can be removed by splitting the above table into two tables as follows.

<i>HTNO</i>	<i>Name</i>	<i>DOB</i>	<i>DeptNo</i>
501	Jhansi	30-10-1998	05
502	Ajay	24-12-1999	05
410	Priya	12-03-2000	04
120	Rahul	30-10-1998	01
415	Smitha	18-06-1999	04

<i>DeptNo</i>	<i>DeptName</i>	<i>Location</i>
05	CSE	A-Block
04	ECE	B-Block
01	CIVIL	C-Block

12. 3NF (THIRD NORMAL FORM)

A relation (table) is in third normal form if and only if it satisfies the following conditions:

- It is in second normal form
- There is no transitive functional dependency

Transitive functional dependency means, we have the following relationships in the table: A is functionally dependent on B ($A \rightarrow B$), and B is functionally dependent on C ($B \rightarrow C$). In this case, C is transitively dependent on A via B ($A \rightarrow B$ and $B \rightarrow C$ mean $A \rightarrow B \rightarrow C$ implies $A \rightarrow C$).

3NF Example:

Consider the following book details table example:

BOOK_DETAILS			
BookID	GenreID	GenreType	Price
1	1	Gardening	250.00
2	2	Sports	149.00
3	1	Gardening	100.00
4	3	Travel	160.00
5	2	Sports	320.00

The above table is not in 3NF because there exist transitive dependency. In the table able,

BookID determines **GenreID**

$\{ \text{BookID} \rightarrow \text{GenreID} \}$

GenreID determines **GenreType**.

$\{ \text{GenreID} \rightarrow \text{GenreType} \}$

$\therefore$ **BookID** determines **GenreType** via **GenreID**.

$\{ \text{BookID} \rightarrow \text{GenreType} \}$

It implies that transitive functional dependency is existing and the structure does not satisfy third normal form. To bring this table in to third normal form, we split the table into two as follows:

BOOK_DETAILS		
BookID	GenreID	Price
1	1	250.00
2	2	149.00
3	1	100.00
4	3	160.00
5	2	320.00

GENRE_DETAILS	
GenreID	GenreType
1	Gardening
2	Sports
3	Travel

13. BOYCE CODD NORMAL FORM (BCNF)

A relation (table) is said to be in the BCNF if and only if it satisfy the following conditions:

- It should be in the **Third Normal Form**.
- For any functional dependency $A \rightarrow B$, A should be a **super key**.

In simple words, it means, that for a dependency $A \rightarrow B$, A cannot be a **non-prime attribute**, if B is a **prime attribute**.

Example: Below we have a Patient table of a hospital. A patient can go to hospital many times to take treatment. On a single day many patients can take treatment.

PatientID	Name	EmailID	AdmittedDate	Drug	Quantity
101	Ram	ram@gmail.com	30/10/1998	A-10	10
102	Jhon	jho@gmail.com	30/10/1998	X-90	10
101	Ram	ram@gmail.com	10/06/2001	X-90	20
103	Sowmya	sam@gmail.com	05/03/2002	Y-30	15
102	Jhon	jho@gmail.com	05/03/2002	A-10	15

In the above table, $\{PatientID, AdmittedDate\}$ acts as Primary key. But if we know the *EmailID* value, we can find *PatientID* value.

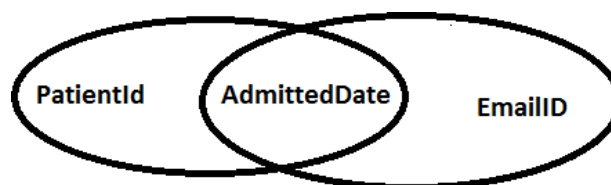
That is $EmailID \rightarrow PatientID$.

In the above dependency, *EmailID* is non-prime attribute and *PatientID* is a prime attribute. Therefore the above table is not in BCNF. In order to bring the table into BCNF, we split it into two tables as shown below.

PatientID	Name	AdmittedDate	Drug	Quantity
101	Ram	30/10/1998	A-10	10
102	Jhon	30/10/1998	X-90	10
101	Ram	10/06/2001	X-90	20
103	Sowmya	05/03/2002	Y-30	15
102	Jhon	05/03/2002	A-10	15

PatientID	EmailID
101	ram@gmail.com
102	jho@gmail.com
103	sam@gmail.com

In other words we can also define BCNF as there should not be any overlapping between candidate keys. If you consider the original table (before splitting), we can get two candidate keys $\{PatientID, AdmittedDate\}$ and $\{EmailID, AdmittedDate\}$.



As there exist overlapping in the candidate keys, the table is not in BCNF. To bring it into BCNF, we split into two tables as shown above.

14. 4-NF (FOURTH NORMAL FORM)

A relation is said to be in 4-NF if and only if it satisfies the following conditions

- It should be in the **Third Normal Form**.
- The table should not have any **Multi-valued Dependency**.

What is Multi-valued Dependency?

A table is said to have multi-valued dependency, if the following three conditions are true.

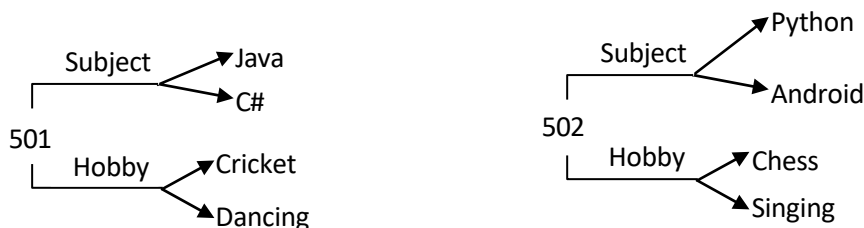
- A table should have at-least 3 columns for it to have a multi-valued dependency.
- For any dependency $A \twoheadrightarrow B$, if there exists multiple value of B for a single value of A , then the table may have multi-valued dependency. It is represented as $A \twoheadrightarrow B$.
- In a relation $R(A, B, C)$, if there is a multi-valued dependency between A and B , then B and C should be independent of each other.

If all these three conditions are true for any relation (table), then it contains multi-valued dependency. The multi-valued dependency can be explained with an example. Let the Relation R containing three columns A, B, C and four rows s, t, u, v .

	A	B	C
s	a1	b1	c1
t	a1	b1	c2
u	a1	b2	c1
v	a1	b2	c2

If $s(A) = t(A) = u(A) = v(A)$
 $s(B) = t(B)$ and $s(B) = v(B)$
 $s(C) = u(C)$ and $t(C) = v(C)$, then there exist multi-valued dependency.

Example: Consider the below college enrolment table with columns HTNO, Subject and Hobby.



As shown in the above figure, if 501 opted for subjects like Java and C# and hobbies of 501 are Cricket and Dancing. Similarly, If 502 opted for subjects like Python and Android and hobbies of 501 are Chess and Singing, then it can be written into a table with three columns as follows:

<i>HTNO</i>	<i>Subject</i>	<i>Hobby</i>
501	Java	Cricket
501	Java	Dancing
501	C#	Cricket
501	C#	Dancing
502	Python	Chess
502	Python	Singing
502	Android	Chess
502	Android	Singing

As there exist multi valued dependency, the above table is decomposed into two tables such that

<i>HTNO</i>	<i>Subject</i>
501	Java
501	C#
502	Python
502	Android

<i>HTNO</i>	<i>Hobby</i>
501	Cricket
501	Dancing
502	Chess
502	Singing

Now these tables (relations) satisfy the fourth normal form.

15. 5NF

A relation is said to be in 5-NF if and only if it satisfies the following conditions

- It should be in the **Fourth Normal Form**.
- The table should not have any **join Dependency** and joining should be lossless.

5NF is also known as Project-join normal form (PJ/NF).

A table is decomposed into multiple small tables to eliminate redundancy, and when we re-join the decomposed tables, there should not be any loss in the original data or should not create any new data. In simple words, joining two or more decomposed table should not lose records nor create new records.

Example: Consider a table which contains a record of **Subject**, **Professor** and **Semester** in three columns. The primary key is the combination of all three columns. No column itself is not a candidate key or a super key.

In the table, DBMS is taught by Ravindar and Uma Rani in semester 4, DS by Sindhusa and Venu in sem 3. In this case, the combination of all these fields required to identify valid data.

So to make the table into 5NF, we can decompose it into three relations,

<i>Subject</i>	<i>Professor</i>	<i>Semester</i>
C	Srilatha	2
DBMS	Ravindar	4
DS	Sindhusa	3
DBMS	Uma Rani	4
CN	Srikanth	5
DS	Venu	3
WT	Srinivas	5

The above table is decomposed into three tables as follows to bring it into 5-NF.

<i>Subject</i>	<i>Professor</i>
C	Srilatha
DBMS	Ravindar
DS	Sindhusa
DBMS	Uma Rani
CN	Srikanth
DS	Venu
WT	Srinivas

<i>Semester</i>	<i>Professor</i>
2	Srilatha
3	Ravindar
5	Sindhusa
3	Uma Rani
5	Srikanth
2	Venu
5	Srinivas

<i>Semester</i>	<i>Subject</i>
2	C
4	DBMS
3	DS
5	CN
5	WT

16. LOSS LESS JOIN DECOMPOSITION

Decomposition of a relation R into R1 and R2 is lossless-join decomposition if at least one of the following functional dependencies are in F+ (Closure of functional dependencies)

$$R1 \cap R2 \rightarrow R1$$

OR

$$R1 \cap R2 \rightarrow R2$$

- Consider a relation R which is decomposed into sub relations R₁ and R₂.
- This decomposition is called lossless join decomposition when we join R₁ and R₂ and if we get the same relation R that was decomposed.
- For lossless join decomposition, we always have: **R₁ ⋈ R₂**

Example 1: Consider the following relation $R(A, B, C)$. Let this relation is decomposed into two sub relations $R_1(A, B)$ and $R_2(B, C)$

R					R ₁		R ₂	
A	B	C			A	B	B	C
1	2	1			1	2	2	1
2	5	3			2	5	5	3
3	3	3			3	3	3	3

Now, let us check whether this decomposition is lossless or not. For lossless decomposition, we must have: $R_1 \bowtie R_2 = R$. Now, if we perform the natural join ($\bowtie$) of the sub relations R_1 and R_2 , we get

A	B	C
1	2	1
2	5	3
3	3	3

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition. This is because the resultant relation after joining the sub relations is same as the decomposed relation. No extraneous tuples (rows) appear after joining of the sub-relations.

Example 2: Consider the following relation $R(A, B, C)$. Let this relation is decomposed into two sub relations $R_1(A, C)$ and $R_2(B, C)$

R					R ₁		R ₂	
A	B	C			A	C	B	C
1	2	1			1	1	2	1
2	5	3			2	3	5	3
3	3	3			3	3	3	3

Now, let us check whether this decomposition is lossless or not. For lossless decomposition, we must have: $R_1 \bowtie R_2 = R$. Now, if we perform the natural join ($\bowtie$) of the sub relations R_1 and R_2 , we get

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

This relation is not same as the original relation R.

Thus, we conclude that the above decomposition is **not** lossless join decomposition. This is because the resultant relation after joining the sub relations is **not** same as the decomposed relation. Extraneous tuples (rows) appear after joining of the sub-relations.

PROBLEMS

Consider a relation R is decomposed into two sub relations R_1 and R_2 .

- If all the following conditions satisfy, then the decomposition is lossless.
- If any of these conditions fail, then the decomposition is lossy.

Condition-01: Union of both the sub relations must contain all the attributes that are present in the original relation R.

$$R_1 \cup R_2 = R$$

Condition-02: Intersection of both the sub relations must not be null. In other words, there must be some common attribute which is present in both the sub relations.

$$R_1 \cap R_2 \neq \emptyset$$

Condition-03: Intersection of both the sub relations must be a super key of either R_1 or R_2 or both.

Problem-01: Consider a relation schema $R (A, B, C, D)$ with the functional dependencies $A \rightarrow B$ and $C \rightarrow D$. Determine whether the decomposition of R into $R_1 (A, B)$ and $R_2 (C, D)$ is lossless or lossy.

Solution: To determine whether the decomposition is lossless or lossy, we will check all the conditions one by one. If any of the conditions fail, then the decomposition is lossy otherwise lossless.

Condition-01: According to condition-01, union of both the sub relations must contain all the attributes of relation R. So, we have:

$$R_1 (A, B) \cup R_2 (C, D) = R (A, B, C, D)$$

Clearly, union of the sub relations contains all the attributes of relation R. Thus, condition-01 satisfies.

Condition-02: According to condition-02, intersection of both the sub relations must not be null. So, we have-

$$R_1 (A, B) \cap R_2 (C, D) = \Phi$$

Clearly, intersection of the sub relations is null. So, condition-02 fails. Thus, we conclude that the decomposition is lossy.

Problem-02: Consider a relation schema $R (A, B, C, D)$ with the following functional dependencies

$$A \rightarrow B \quad B \rightarrow C \quad C \rightarrow D \quad D \rightarrow B$$

Determine whether the decomposition of R into $R_1(A, B)$, $R_2(B, C)$ and $R_3(B, D)$ is lossless or lossy.

Solution:

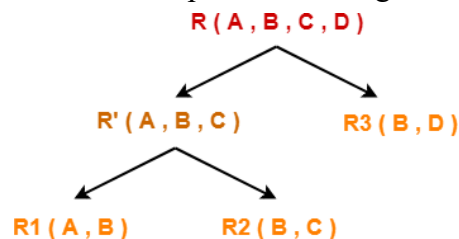
Strategy to Solve: When a given relation is decomposed into more than two sub relations, then

- Consider any one possible ways in which the relation might have been decomposed into those sub relations.
- First, divide the given relation into two sub relations.
- Then, divide the sub relations according to the sub relations given in the question.

As a thumb rule, remember-

Any relation can be decomposed only into two sub relations at a time.

Consider the original relation R was decomposed into the given sub relations as shown:



Decomposition of $R(A, B, C, D)$ into $R'(A, B, C)$ and $R_3(B, D)$ -

To determine whether the decomposition is lossless or lossy,

- We will check all the conditions one by one.
- If any of the conditions fail, then the decomposition is lossy otherwise lossless.

Condition-01: According to condition-01, union of both the sub relations must contain all the attributes of relation R . So, we have

$$R'(A, B, C) \cup R_3(B, D) = R(A, B, C, D)$$

Clearly, union of the sub relations contains all the attributes of relation R . Thus, condition-01 satisfies.

Condition-02: According to condition-02, intersection of both the sub relations must not be null. So, we have

$$R'(A, B, C) \cap R_3(B, D) = B$$

Clearly, intersection of the sub relations is not null. Thus, condition-02 satisfies.

Condition-03: According to condition-03, intersection of both the sub relations must be the super key of one of the two sub relations or both. So, we have-

$$R'(A, B, C) \cap R_3(B, D) = B$$

Now, the closure of attribute B is: $B^+ = \{ B, C, D \}$

So,

- Attribute 'B' can not determine attribute 'A' of sub relation R'.
- Thus, it is not a super key of the sub relation R'.
- Attribute 'B' can determine all the attributes of sub relation R₃.
- Thus, it is a super key of the sub relation R₃.

Clearly, intersection of the sub relations is a super key of one of the sub relations.

So, condition-03 satisfies. Thus, we conclude that the decomposition is lossless.

Decomposition of R'(A, B, C) into R₁(A, B) and R₂(B, C)-

To determine whether the decomposition is lossless or lossy,

- We will check all the conditions one by one.
- If any of the conditions fail, then the decomposition is lossy otherwise lossless.

Condition-01: According to condition-01, union of both the sub relations must contain all the attributes of relation R'. So, we have

$$R_1(A, B) \cup R_2(B, C) = R'(A, B, C)$$

Clearly, union of the sub relations contain all the attributes of relation R'. Thus, condition-01 satisfies.

Condition-02: According to condition-02, intersection of both the sub relations must not be null.

So, we have

$$R_1(A, B) \cap R_2(B, C) = B$$

Clearly, intersection of the sub relations is not null. Thus, condition-02 satisfies.

Condition-03: According to condition-03, intersection of both the sub relations must be the super key of one of the two sub relations or both. So, we have

$$R_1(A, B) \cap R_2(B, C) = B$$

Now, the closure of attribute B is: $B^+ = \{ B, C, D \}$

So,

- Attribute 'B' can not determine attribute 'A' of sub relation R₁.
- Thus, it is not a super key of the sub relation R₁.
- Attribute 'B' can determine all the attributes of sub relation R₂.
- Thus, it is a super key of the sub relation R₂.

Clearly, intersection of the sub relations is a super key of one of the sub relations. So, condition-03 satisfies. Thus, we conclude that the decomposition is lossless.

17. CLOSURE OF AN ATTRIBUTE SET:

The set of all those attributes which can be functionally determined from an attribute set is called as a closure of that attribute set. Closure of attribute set $\{X\}$ is denoted as $\{X\}^+$.

Steps to Find Closure of an Attribute Set: Following steps are followed to find the closure of an attribute set:

Step-01: Add the attributes contained in the attribute set for which closure is being calculated to the result set.

Step-02: Recursively add the attributes to the result set which can be functionally determined from the attributes already contained in the result set.

Question 1: Consider a relation $R (A , B , C , D , E , F , G)$ with the functional dependencies

$$A \rightarrow BC, \quad BC \rightarrow DE, \quad D \rightarrow F, \quad CF \rightarrow G$$

Find the closure of $\{A\}$, $\{D\}$ and $\{B,C\}$ attributes and attribute sets

Solution:

Closure of attribute A:

$$\begin{aligned} A^+ &= \{ A \} \\ &= \{ A, B, C \} && \text{(Using } A \rightarrow BC \text{)} \\ &= \{ A, B, C, D, E \} && \text{(Using } BC \rightarrow DE \text{)} \\ &= \{ A, B, C, D, E, F \} && \text{(Using } D \rightarrow F \text{)} \\ &= \{ A, B, C, D, E, F, G \} && \text{(Using } CF \rightarrow G \text{)} \end{aligned}$$

Thus,

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

Closure of attribute D:

$$\begin{aligned} D^+ &= \{ D \} \\ &= \{ D, F \} \text{ (Using } D \rightarrow F \text{)} \end{aligned}$$

We cannot determine any other attribute using attributes D and F contained in the result set.

Thus,

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

Closure of attribute set {B, C}

$$\begin{aligned} \{ B, C \}^+ &= \{ B, C \} \\ &= \{ B, C, D, E \} && \text{(Using } BC \rightarrow DE \text{)} \\ &= \{ B, C, D, E, F \} && \text{(Using } D \rightarrow F \text{)} \\ &= \{ B, C, D, E, F, G \} && \text{(Using } CF \rightarrow G \text{)} \end{aligned}$$

Thus,

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

Question2: Consider the given functional dependencies

$AB \rightarrow CD$ $AF \rightarrow D$ $DE \rightarrow F$ $C \rightarrow G$ $F \rightarrow E$ $G \rightarrow A$

Find the closure of $\{A, B\}$, $\{A, F\}$, $\{B, G\}$ and $\{C, F\}$

Solution:

Closure of $\{A, B\}$:

$$\begin{aligned}\{AB\}^+ &= \{A, B\} \\ &= \{A, B, C, D\} \quad (\text{Using } AB \rightarrow CD) \\ &= \{A, B, C, D, G\} \quad (\text{Using } C \rightarrow G)\end{aligned}$$

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

Closure of $\{C, F\}$:

$$\begin{aligned}\{CF\}^+ &= \{C, F\} \\ &= \{C, F, G\} \quad (\text{Using } C \rightarrow G) \\ &= \{C, E, F, G\} \quad (\text{Using } F \rightarrow E) \\ &= \{A, C, E, F\} \quad (\text{Using } G \rightarrow A) \\ &= \{A, C, D, E, F, G\} \quad (\text{Using } AF \rightarrow D)\end{aligned}$$

Thus, $\{CF\}^+ = \{A, C, D, E, F, G\}$

Closure of $\{B, G\}$:

$$\begin{aligned}\{BG\}^+ &= \{B, G\} \\ &= \{A, B, G\} \quad (\text{Using } G \rightarrow A) \\ &= \{A, B, C, D, G\} \quad (\text{Using } AB \rightarrow CD)\end{aligned}$$

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

Closure of $\{A, F\}$:

$$\begin{aligned}\{AF\}^+ &= \{A, F\} \\ &= \{A, D, F\} \quad (\text{Using } AF \rightarrow D) \\ &= \{A, D, E, F\} \quad (\text{Using } F \rightarrow E)\end{aligned}$$

Thus, $\{AF\}^+ = \{A, D, E, F\}$

18. FINDING THE KEYS USING CLOSURE

Super Key:

If the closure result of an attribute set contains all the attributes of the relation, then that attribute set is called as a super key of that relation.

- Thus, we can say, “**The closure of a super key is the entire relation schema.**”

Example: In the above example (Question 1),

- The closure of attribute A is the entire relation schema.
- Thus, attribute A is a super key for that relation.

Candidate Key:

- If there exists no subset of an attribute set whose closure contains all the attributes of the relation, then that attribute set is called as a candidate key of that relation.

Example: In the above example (Question 1),

- No subset of attribute A contains all the attributes of the relation.
- Thus, attribute A is also a candidate key for that relation.

Finding Candidate Keys From a Relation:

We can determine the candidate keys of a given relation using the following steps-

Step-01: Determine all essential attributes of the given relation

- Essential attributes are those attributes which are not present on RHS of any functional dependency.
- Essential attributes are always a part of every candidate key. This is because they cannot be determined by other attributes.

Example: Let $R(A, B, C, D, E, F)$ be a relation scheme with the following functional dependencies: $A \rightarrow B$, $C \rightarrow D$ and $D \rightarrow E$.

The RHS of all the above functional dependencies contain only B, D and E. The attributes which are not present on RHS of any functional dependency are A, C and F. So, essential attributes are: **A, C and F**.

Step-02: Determining all non-essential attributes using essential attributes

- The attributes of the relation that are present in RHS are non-essential attributes. They can be determined by using essential attributes.
- Now, following two cases are possible-
- **Case-01:** If all essential attributes together can determine all remaining non-essential attributes, then
 - The combination of essential attributes is the candidate key.
 - It is the only possible candidate key.
- **Case-02:** If all essential attributes together can not determine all remaining non-essential attributes, then-
- The set of essential attributes and some non-essential attributes will be the candidate key(s).
- In this case, multiple candidate keys are possible.
- To find the candidate keys, we check different combinations of essential and non-essential attributes.

PROBLEMS ON FINDING CANDIDATE KEYS

Problem-01: Let $R = (A, B, C, D, E, F)$ be a relation scheme with the following dependencies: $C \rightarrow F$, $E \rightarrow A$, $EC \rightarrow D$ and $A \rightarrow B$. Find the candidate key. Also, determine the total number of candidate keys and super keys.

Solution: We will find candidate keys of the given relation in the following steps-

Step-01: Determine all essential attributes of the given relation.

- Essential attributes of the relation are: C and E.
- So, attributes C and E will definitely be a part of every candidate key.

Step-02: Now, we will check if the essential attributes together can determine all remaining non-essential attributes. To check, we find the closure of CE. So,

$$\begin{aligned}\{CE\}^+ &= \{C, E\} \\ &= \{C, E, F\} && (\text{Using } C \rightarrow F) \\ &= \{A, C, E, F\} && (\text{Using } E \rightarrow A) \\ &= \{A, C, D, E, F\} && (\text{Using } EC \rightarrow D) \\ &= \{A, B, C, D, E, F\} && (\text{Using } A \rightarrow B)\end{aligned}$$

We conclude that CE can determine all the attributes of the given relation. So, CE is the only possible candidate key of the relation.

Total Number of Super Keys-

There are total 6 attributes in the given relation of which-

- There are 2 essential attributes- C and E.
- Remaining 4 attributes are non-essential attributes.
- Essential attributes will be definitely present in every key.
- Non-essential attributes may or may not be present in every super key.

C E A B D F

Essential attributes Non-Essential attributes

So, number of super keys possible = $2 \times 2 \times 2 \times 2 = 16$. Thus, total number of super keys possible = 16.

Problem-02: Consider the relation scheme $R(E, F, G, H, I, J, K, L, M, N)$ and the set of functional dependencies: $\{E, F\} \rightarrow \{G\}$, $\{F\} \rightarrow \{I, J\}$, $\{E, H\} \rightarrow \{K, L\}$, $\{K\} \rightarrow \{M\}$ and $\{L\} \rightarrow \{N\}$. Determine the candidate key(s).

Solution: We will find candidate keys of the given relation in the following steps-

Step-01: Determine all essential attributes of the given relation.

- Essential attributes of the relation are- E, F and H.
- So, attributes E, F and H will definitely be a part of every candidate key.

Step-02:

- We will check if the essential attributes together can determine all remaining non-essential attributes.
- To check, we find the closure of EFH.

So, we have-

$$\begin{aligned}\{ EFH \}^+ &= \{ E, F, H \} \\ &= \{ E, F, G, H \} && \text{(Using } EF \rightarrow G \text{)} \\ &= \{ E, F, G, H, I, J \} && \text{(Using } F \rightarrow IJ \text{)} \\ &= \{ E, F, G, H, I, J, K, L \} && \text{(Using } EH \rightarrow KL \text{)} \\ &= \{ E, F, G, H, I, J, K, L, M \} && \text{(Using } K \rightarrow M \text{)} \\ &= \{ E, F, G, H, I, J, K, L, M, N \} && \text{(Using } L \rightarrow N \text{)}\end{aligned}$$

We conclude that EFH can determine all the attributes of the given relation. So, **EFH** is the only possible candidate key of the relation.

***** ALL THE BEST *****

R25 DATABASE MANAGEMENT SYSTEM

UNIT – IV

Transaction Management: Transaction Concept, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for serializability, Lock Based Protocols, Timestamp Based Protocols, Validation- Based Protocols, Multiple Granularity, Recovery and Atomicity, Log–Based Recovery, Recovery with Concurrent Transactions.

1. TRANSACTION

Definition: A transaction is a single logical unit consisting of one or more database access operation.

Example: Withdrawing 1000 rupees from ATM.

The following set of operations are performed to withdraw 1000 rupees from database

- | | | | |
|------|------------------------------------|-----------------------|-------------------|
| i. | Read current balance from Database | (Let say 5000 rupees) | } one Transaction |
| ii. | Deduct 1000 from current balance | (5000 – 1000 = 4000) | |
| iii. | Update current balance in Database | (4000 rupees) | |

- Every transaction is executed as a single unit.
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one *consistent state* to another *consistent state*.
- DBMS transactions must satisfy ACID properties (atomic, consistent, isolated and durable).

2. ACID PROPERTIES

ACID properties are used for maintaining the integrity of database during transaction processing. ACID stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- **Atomicity:** This property ensure that either all of the tasks of a transaction are performed or none of them. In simple words it is referred as “*all or nothing rule*”.

Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails. When all parts of the transaction completed successfully, then the transaction said to be success. (“*all or nothing rule*”)

Example: Transferring \$100 from account **A** to account **B**.

(Assume initially, account **A** balance = \$400 and account **B** balance = 700\$.)

Transferring \$100 from account **A** to account **B** has two operations

- a) Debiting 100\$ from **A**'s balance ($\$400 - \$100 = \$300$)
- b) Crediting 100\$ to **B**'s balance ($\$700 + \$100 = \$800$)

Let's say first operation (a) passed successfully while second (b) failed, in this case **A**'s balance would be 300\$ while **B** would be having 700\$ instead of 800\$. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.

- ii. Consistency:** The consistency property ensures that the database must be in consistent state before and after the transaction. There must not be any possibility that some data is incorrectly affected by the execution of a transaction.

For example, transferring funds from one account to another, the consistency property ensures that the total values of funds in both the accounts is the same before and end of the transaction. i.e., Assume initially, **A** balance = \$400 and **B** balance = 700\$.

The total balance of **A** + **B** = 1100\$ (Before transferring 100\$ from **A** to **B**)

The total balance of **A** + **B** = 1100\$ (After transferring 100\$ from **A** to **B**)

- iii. Isolation:** For every pair of transactions, one of the transactions should not start execution before the other transaction execution completed, if they use some common data variable. That is, if the transaction T1 is executing and using the data item X, then transaction T2 should not start until the transaction T1 ends, if T2 also use same data item X.

For example, Transaction **T1**: Transfer 100\$ from account **A** to account **B**

Transaction **T2**: Transfer 150\$ from account **B** to account **C**

Assume initially, **A** balance = **B** balance = **C** balance = \$1000

	Transaction T1	Transaction T2
10:00 AM	Read A's balance (\$1000)	Read B's balance (\$1000)
10:01 AM	A balance = A Balance – 100\$ (1000-100 = 900\$)	B balance = B Balance – 150\$ (1000-150 = 850\$)
10:02 AM	Read B's balance (\$1000)	Read C's balance (\$1000)
10:03 AM	B balance = B Balance + 100\$ (1000+100 = 1100\$)	C balance = C Balance + 150\$ (1000+150 = 1150\$)
10:04 AM	Write A's balance (900\$)	Write B's balance (850\$)
10:05AM	Write B's balance (1100\$)	Write C's balance (1150\$)
10:06 AM	COMMIT	COMMIT

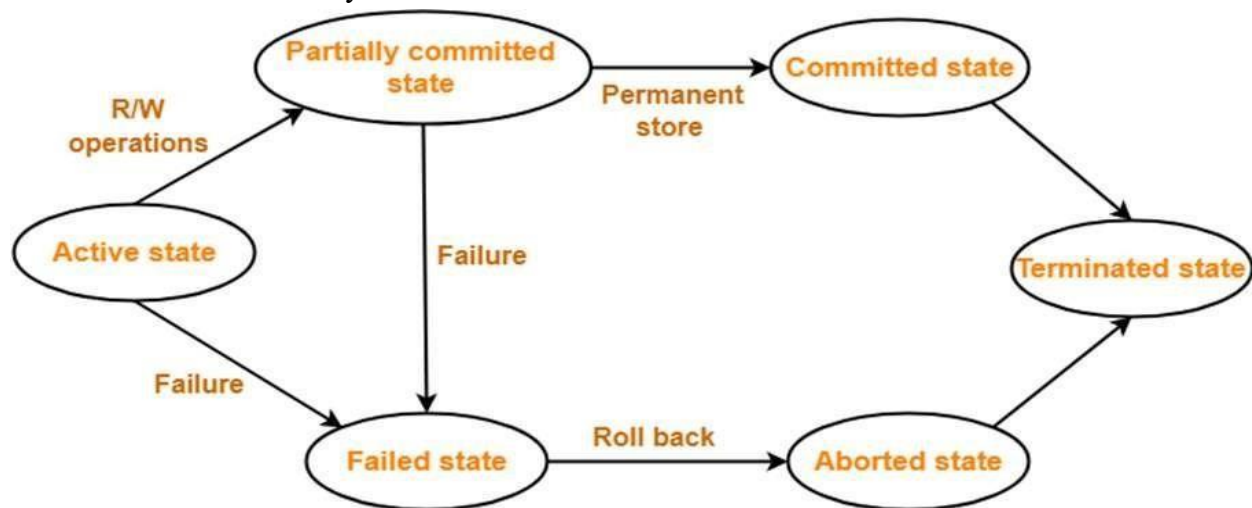
After completion of Transaction **T1** and **T2**, **A** balance = 900\$, **B** balance = 1100\$, **C** balance

=1150\$. But **B** balance should be 950\$. The **B** balance is wrong due to execution of T1 and T2 parallel and in both the transactions, Account **B** is common. The last write in account **B** is at 10:05 AM, so that **B** balance is 1100\$ (write in account **B** at 10:04 AM is overwritten).

- iv. **Durability:** Once a transaction completes successfully, the changes it has made into the database should be permanent even if there is a system failure. The recovery-management component of database systems ensures the durability of transaction. For example, assume account **A** balance = 1000\$. If **A** withdraw 100\$ today, then the **A** balance = 900\$. After two days or a month, **A** balance should be 900\$, if no other transactions done on **A**.

3. STATES OF TRANSACTION

A transaction goes through many different states throughout its life cycle. These states are called as **transaction states**. They are:



Active State:

- This is the first state in the life cycle of a transaction.
 - Once the transaction starts executing, then it is said to be in active state.
 - During this state it performs operations like READ and WRITE on some data items. All the changes made by the transaction are now stored in the buffer in main memory. They are not updated in database.
 - From active state, a transaction can go into either a partially committed state or a failed state.
-

Partially Committed State:

- When the transaction executes its last statement, then the transaction is said to be in partially committed state.
- Still, all the changes made by the transaction are stored in the buffer in main memory, but they are not updated in the database.
- From partially committed state, a transaction can go into one of two states, a committed state or a failed state.

Committed State:

- After all the changes made by the transaction have been successfully updated in the database, it enters into a **committed state** and the transaction is considered to be fully committed.
- After a transaction has entered the committed state, it is not possible to roll back (undo) the transaction. This is because the system is updated into a new consistent state and the changes are made permanent.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

Failed State:

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

Aborted State:

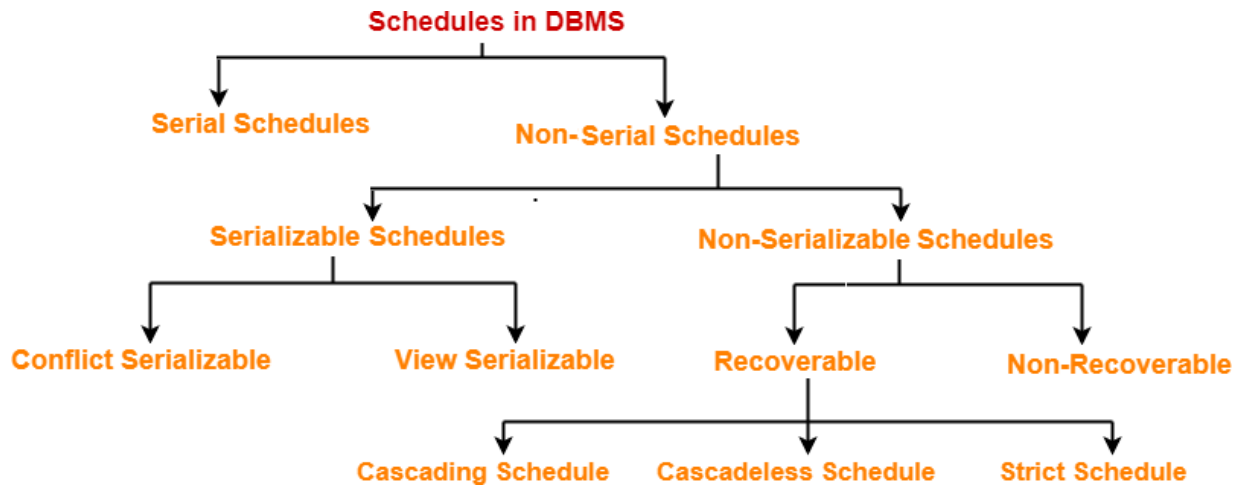
- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an **aborted state**.

Terminated State:

- This is the last state in the life cycle of a transaction.
 - After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.
-

4. TYPES OF SCHEDULES – SERIALIZABILITY

In DBMS, schedules may be classified as



i. Serial Schedules:

- All the transactions execute serially one after the other.
- When one transaction executes, no other transaction is allowed to execute.

Examples:

Schedule-1	
T1	T2
Read(A) A=A-100 Write(A) Read(B) B=B+100 Write(B) COMMIT	 Read(A) A=A+500 Write(A) COMMIT

Schedule-2	
T1	T2
 Read(A) A=A-100 Write(A) Read(B) B=B+100 Write(B) COMMIT	Read(A) A=A+500 Write(A) COMMIT

In schedule 1, after T1 completes its execution, transaction T2 executes. So, schedule-1 is a *Serial Schedule*. Similarly, in schedule-2, after T2 completes its execution, transaction T1 executes. So, schedule -2 is also an example of a *Serial Schedule*.

ii. Non-Serial Schedules:

- In non-serial schedules, multiple transactions execute concurrently.
 - Operations of all/some of the transactions are inter-leaved or mixed with each other.
 - Some non-serial schedules may lead to inconsistency of the database and may produce wrong results.
-

Examples:

Schedule-1		Schedule-2	
T1	T2	T1	T2
Read(A) A=A-100 Write(A)	Read(A) A=A+500	Read(A) A=A-100 Write(A)	A=A+500
Read(B) B=B+100 Write(B) COMMIT		Read(B) B=B+100 Write(B) COMMIT	
	Write(A) COMMIT		Write(A) COMMIT

In schedule-1 and schedule-2, the two transactions T1 and T2 executing concurrently. The operations of T1 and T2 are interleaved. So, these schedules are **Non-Serial Schedule**.

iii. Serializable Schedules:

- A non-serial schedule of 'n' transactions is equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule**.
- In other words, the results produced by the transactions in a serial schedule are equal to the result produced by the same transactions in some non-serial schedule, then that non-serial schedule is called as serializability.
- Serializable schedules behave exactly same as serial schedules.
- Even though, Serial Schedule and Serializable Schedule produce same result, there are some differences they are

Serial Schedules	Serializable Schedules
Concurrency is not allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
It leads to less resource utilization and CPU throughput.	It improves both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.

Serializability is mainly of two types. They are:

- Conflict Serializability
- View Serializability

Conflict Serializability: If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Two operations are called as **conflicting operations** if all the following conditions hold true

- (1) Both the operations belong to different transactions
- (2) Both the operations are on the **same data item**
- (3) At least one of the two operations is a write operation

Schedule – 1		Schedule – 2		Schedule - 3		Schedule - 4	
T1	T2	T1	T2	T1	T2	T1	T2
Read(A)	Read(A)	Read(A)	Write(A)	Write(B)	Read(A)	Write(B)	Write(B)

In Schedule -1, only rule (1) & (2) are true, but rule (3) is not holding. So, the operations are not conflict.

In Schedule -2, rule (1), (2) & (3) are true. So, the operations are conflict.

In Schedule -3, only rule (1) & (3) are true, but rule (2) is not holding. So, the operations are not conflict.

In Schedule -4, rule (1), (2) & (3) are true. So, the operations are conflict.

Testing of Conflict Serializability: Precedence Graph is used to test the Conflict Serializability of a schedule. The algorithm to draw precedence graph is

- (1) Draw a node for each transaction in Schedule S.
- (2) If T_a reads X value written by T_b , then draw arrow from $T_b \rightarrow T_a$.
- (3) If T_b writes X value after it has been read by T_a , then draw arrow from $T_a \rightarrow T_b$.
- (4) If T_a writes X after T_b writes X, then draw arrow from $T_b \rightarrow T_a$.

If the precedence graph has no cycle, then Schedule S is known as conflict serializable. If a precedence graph contains a cycle, then S is not conflict serializable.

Problem-01: Check whether the given schedule S is conflict serializable or not.

S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$

Solution:

Given that S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$.

The schedule for the above operations is

Schedule-1

T1	T2	T3
Read(A)	Read(A)	Read(B)
Read(B)	Read(B)	
Write(A)	Write(B)	

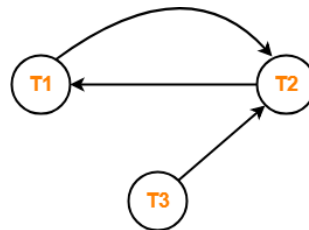
List all the conflicting operations and determine the dependency between the transactions

(Thumb rule to find conflict operations: For each Write(X) in T_a , make a pair with each Read(X) and Write(X) in T_b .

The order is important in each pair i.e., for example, Read after Write on X or write after read on X in the given schedule.)

- $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
- $R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Draw the precedence graph:



There exists a cycle in the above graph. Therefore, the schedule S is not conflict serializable.

Problem-02: Check whether the given schedule S is conflict serializable schedule.

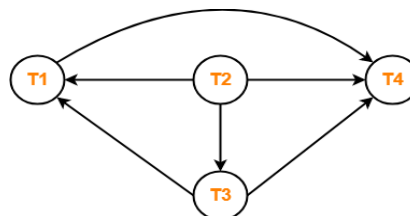
Schedule – S

T1	T2	T3	T4
Write(X) COMMIT	Read(X) Write(Y) Read(Z) COMMIT	Write(X) COMMIT	Read(X) Read(Y) COMMIT

Solution: List all the conflicting operations to determine the dependency between transactions.

- $R_2(X)$, $W_3(X)$ $(T_2 \rightarrow T_3)$
- $W_3(X)$, $W_1(X)$ $(T_3 \rightarrow T_1)$
- $W_3(X)$, $R_4(X)$ $(T_3 \rightarrow T_4)$
- $R_2(X)$, $W_1(X)$ $(T_2 \rightarrow T_1)$
- $W_1(X)$, $R_4(X)$ $(T_1 \rightarrow T_4)$
- $W_2(Y)$, $R_4(Y)$ $(T_2 \rightarrow T_4)$

Draw the precedence graph:



There exists no cycle in the precedence graph. Therefore, the schedule S is conflict serializable.

View Serializability: Two schedules S1 and S2 are said to be **view equivalent** if both of them satisfy the following three rules:

- (1) **Initial Read:** The first read operation on each data item in both the schedule must be same.
 - For each data item X, If first read on X is done by transaction T_a in schedule S1, then in schedule2 also the first read on X must be done by transaction T_a only.
- (2) **Updated Read:** It should be same in both the schedules.
 - If Read(X) of T_a followed by Write(X) of T_b in schedule S1, then in schedule S2 also, Read(X) of T_a must follow Write(X) of T_b .
- (3) **Final write:** The final write operation on each data item in both the schedule must be same.
 - For each data item X, if X has been updated at last by transaction T_i in schedule S1, then in schedule S2 also, X must be updated at last by transaction T_i .

View Serializability Definition: If a given schedule is view equivalent to some serial schedule, then it is called as a view serializable schedule.

Note: Every conflict serializable schedule is also view serializable schedule but not vice-versa

Problem 03: Check whether the given schedule S is view serializable or not

Schedule – 1

T1	T2
Read(A) Write(A)	Read(A) Write(A)
Read(B) Write(B)	
	Read(B) Write(B)

Solution:

For the given schedule-1, the serial schedule can be schedule -2

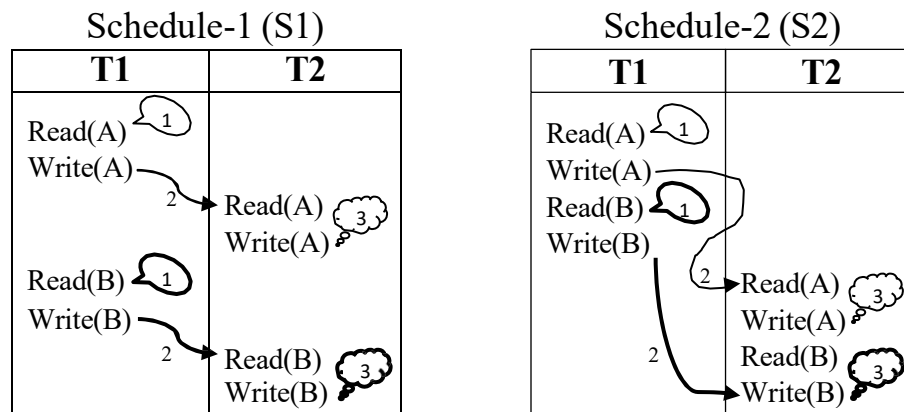
Schedule-1 (S1)

T1	T2
Read(A) Write(A)	Read(A) Write(A)
Read(B) Write(B)	
	Read(B) Write(B)

Schedule-2 (S2)

T1	T2
Read(A) Write(A) Read(B) Write(B)	Read(A) Write(A) Read(B) Write(B)

Now let us check whether the three rules of view-equivalent satisfy or not.



Rule 1: Initial Read

First Read(A) is by T1 in S1 and in S2 also the first Read(A) is by T1 only.

First Read(B) is by T1 in S1 and in S2 also the first Read(B) is by T1 only.

Rule 2: Updated Read

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Rule 3: Final Write

The final Write(A) is by T2 in S1 and in S2 also the final Write(A) is by T2 only

The final Write(B) is by T2 in S1 and in S2 also the final Write(B) is by T2 only

Conclusion: Hence, all the three rules are satisfied in this example, which means Schedule S1 and S2 are view equivalent. Also, it is proved that schedule S2 is the serial schedule of S1. Thus we can say that the S1 schedule is a view serializable schedule.

Note: Other way of solving it is, if we are able to prove that S1 is conflict serializable, then S1 is also view serializable. (Refer conflict serializable problems. Every conflict serializable schedule is also view serializable but not vice-versa.)

5. IMPLEMENTATION OF ATOMICITY AND DURABILITY

The recovery-management component of a DBMS supports atomicity and durability by a variety of schemes. The simplest scheme to implement it is Shadow copy.

Shadow copy: In shadow-copy scheme,

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

- If the transaction complete successfully, then the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the original copy of the database. The old copy of the database is then deleted. Figure below depicts the scheme, showing the database state before and after the update.

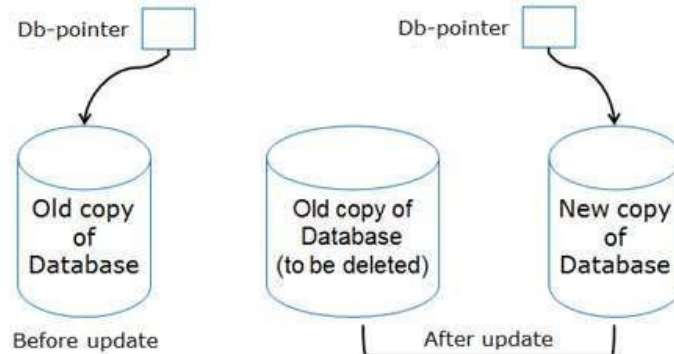


Figure: Shadow copy technique for atomicity and durability

6. RECOVERABILITY

During execution, if any of the transaction in a schedule is aborted, then this may leads the database into inconsistence state. If anything goes wrong, then the completed operations in the schedule needs to be undone. Sometimes, these undone operations may not possible. The recoverability of schedule depends on undone operations.

If a transaction reads a data value that is updated by an uncommitted transaction, then this type of read is called as a **dirty read**.

Irrecoverable Schedule: In a schedule, if a transaction T_a performs a dirty read operation from other transaction T_b and T_a commits before T_b then such a schedule is known as an **Irrecoverable Schedule**.

Example: Consider the following schedule

T1	T2
Read(A) Write(A)	
	Read(A) //Dirty Read
	Write(A)
	COMMIT
ROLLBACK	

Here,

- T2 performs a dirty read operation.

- T2 commits before T1.
- T1 fails later and roll backs.
- The value that T2 read now stands to be incorrect.
- T2 cannot recover since it has already committed.

Recoverable Schedules: In a schedule, if a transaction T_a performs a dirty read operation from other transaction T_b and T_a commit operation delayed till T_b commit, then such a schedule is known as an **Irrecoverable Schedule**.

Example: Consider the following schedule-

T1	T2
Read(A) Write(A) COMMIT	Read(A) //Dirty Read Write(A) COMMIT //Delayed

Here,

- T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.
- In case, T1 would have failed, T2 has a chance to recover by rolling back.

Checking Whether a Schedule is Recoverable or Irrecoverable:

Check if there exists any dirty read operation.

- If there does not exist any dirty read operation, then the schedule is surely recoverable.
- If there exists any dirty read operation, then
 - If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then the schedule is irrecoverable.
 - If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

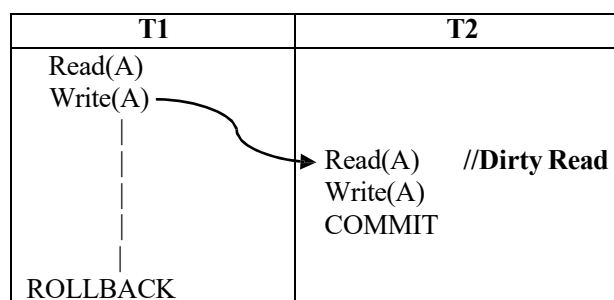
7. IMPLEMENTATION OF ISOLATION

Isolation determines how transactions integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only one transaction that is accessing the resources in a database system.

Isolation level defines the degree to which a transaction must be isolated from the data modifications made by any other transactions in the database system. The phenomena's used to define levels of isolation are:

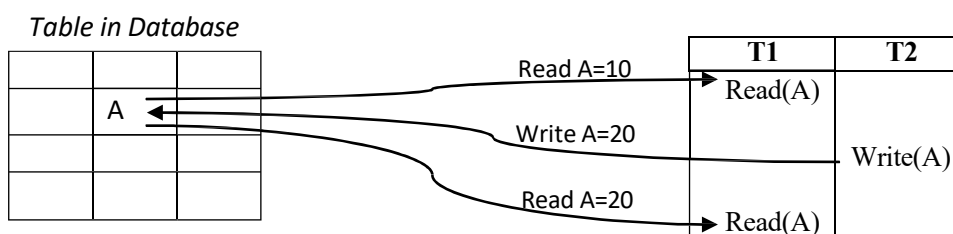
- a) Dirty Read
- b) Non-repeatable Read
- c) Phantom Read

Dirty Read: If a transaction reads a data value updated by an uncommitted transaction, then this type of read is called as dirty read.



As T1 aborted, the results produced by T2 become wrong. This is because T2 read A (Dirty Read) which is updated by T1.

Non-Repeatable Read: Non repeatable read occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **UPDATES** from another transaction.



First, T1 reads data item A and get A=10

Next, T2 writes data item A as A = 20

Last, T1 reads data item A and get A=20

Other example for Non-repeatable read:

Table: STUDENT_DATA before T2

A	B	C
100	5	10
101	5	20
102	6	30

Table: STUDENT_DATA after T2

A	B	C
100	5	15
101	5	20
102	6	30

T1: **SELECT SUM(C) FROM STUDENT_DATA WHERE B=5;**

Answer is $(10+20) = 30$

T2: UPDATE STUDENT_DATA SET C = 15 WHERE A=100;

Answer, in First row C changes to 15

T1: **SELECT SUM(C) FROM STUDENT_DATA WHERE B=5;**

Answer is $(15+20) = 35$

Phantom reads: Phantom reads occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **INSERTS** and/or **DELETES** from another transaction.

Non-repeatable read	Phantom read
When T1 perform second read, there is no change in no of rows in the given table	When T1 perform second read, the no of rows either increase or decrease.
T2 perform UPDATE operation on the given table	T2 perform INSERT and/or DELETE operation on the given table

Example for Phantom read:

Table: STUDENT_DATA before T2

A	B	C
100	5	10
101	5	20
102	6	30

Table: STUDENT_DATA after T2

A	B	C
100	5	10
101	5	20
102	6	30
103	5	25

T1: **SELECT SUM(C) FROM STUDENT_DATA WHERE B=5;**

Answer is $(10+20) = 30$

T2: INSERT INTO STUDENT_DATA VALUES(103, 5, 25);

Answer, in First row C changes to 15

T1: **SELECT SUM(C) FROM STUDENT_DATA WHERE B=5;**

Answer is $(10+20+25) = 55$

Based on these three phenomena, SQL define four isolation levels. They are:

(1) Read uncommitted: This is the lowest level of isolation. In this level, one transaction may read the data item modified by other transaction which is not committed. It mean dirty read is allowed. In this level, transactions are not isolated from each other.

- (2) **Read Committed:** This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read/write lock on the data object, and thus prevents other transactions from reading, updating or deleting it.
- (3) **Repeatable Read:** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read. So other transactions cannot read, update or delete these data items.
- (4) **Serializable:** This is the highest isolation level. A *serializable* execution is guaranteed to be a serial schedule. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

The table given below clearly depicts the relationship between isolation levels and the read phenomena and locks.

Isolation Level	Dirty Read	Non-repeatable read	Phantom Read
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

From the above table, it is clear that serializable isolation level is better than others.

8. CONCURRENCY CONTROL

- **Concurrency** is the ability of a database to execute multiple transactions simultaneously.
- Concurrency control is a mechanism to manage the simultaneously executing multiple transactions such that no transaction interfere with other transaction.
- Executing multiple transactions concurrently improves the system performance.
- Concurrency control increases the throughput and reduces waiting time of transactions.
- If Concurrency Control is not done, then it may leads to problems like lost updates, dirty read, non-repeatable read, phantom read etc. (Refer section 7 for more details)
- **Lost Updates:** It occur when two transactions update same data item at the same time. In this the first write is lost and only the second write is visible.

Concurrency control Protocols:

The concurrency can be controlled with the help of the following Protocols

- (1) Lock-Based Protocol
 - (2) Timestamp-Based Protocol
 - (3) Validation-Based Protocol
-

9. LOCK-BASED PROTOCOL

- Lock assures that one transaction should not retrieve or update a record which another transaction is updating.
- For example, traffic at junction, there are signals which indicate stop and go. When one side signal is green (vehicles allowed passing), then other side signals are red (locked. Vehicles not allowed passing). Similarly, in database transaction when one transaction operations are under execution, the other transactions are locked.
- If at a junction, green signal is given to more than one side, then there may be chances of accidents. Similarly, in database transactions, if the locking is not done properly, then it will display the inconsistent and corrupt data.

There are two lock modes: (1). Shared Lock (2). Exclusive Lock

Shared Locks are represented by S. If a transaction T_i apply shared lock on data item A, then T_i can only read A but not write into A. Shared lock is requested using lock-S instruction.

Exclusive Locks are represented by X. If a transaction T_i apply exclusive lock on data item A, then T_i can read as well as write data item A. Exclusive lock is requested using lock-X instruction.

Lock Compatibility Matrix:

- Lock Compatibility Matrix controls whether multiple transactions can acquire locks on the same resource at the same time.

		Transaction T_i applied	
		Shared	Exclusive
Transaction T_j request for	Shared	√	X
	Exclusive	X	X

- If a transaction T_i applied shared lock on data item A, then T_j can also be allowed to apply shared lock on A.
 - If a transaction T_i applied shared lock on data item A, then T_j is not allowed to apply exclusive lock on A.
 - If a transaction T_i applied exclusive lock on data item A, then T_j is not allowed to apply shared lock on A.
 - If a transaction T_i applied exclusive lock on data item A, then T_j is not allowed to apply exclusive lock on it.
 - Any number of transactions can hold shared locks on a data item, but if any transaction holds an exclusive lock on a data item, then other transactions are not allowed to hold any lock on that data item.
-

- Whenever a transaction wants to read a data item, it should apply shared lock and when a transaction wants to write it should apply exclusive lock. If the lock is not applied, then the transaction is not allowed to perform the operation.

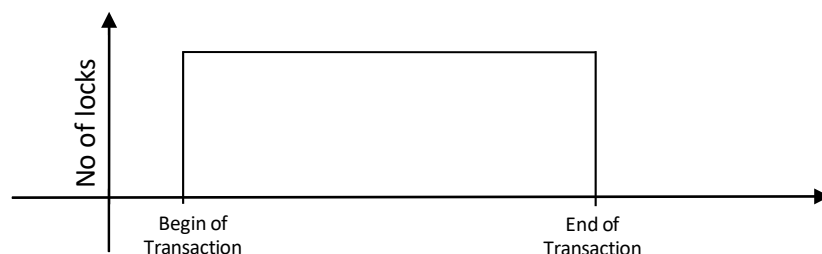
There are four types of lock protocols available. They are:

(1) Simplistic lock protocol

- It is the simplest locking protocol.
- It considers each read/write operation of a transaction as individual.
- It allows transactions to perform write/read operation on a data item only after obtaining a lock on that data item.
- Transactions unlock the data item immediately after completing the write/read operation.
- When a transaction needs to perform many read and write operations, for each operation lock is applied before performing it and release the lock immediately after completion of the operation.

(2) Pre-claiming Lock Protocol

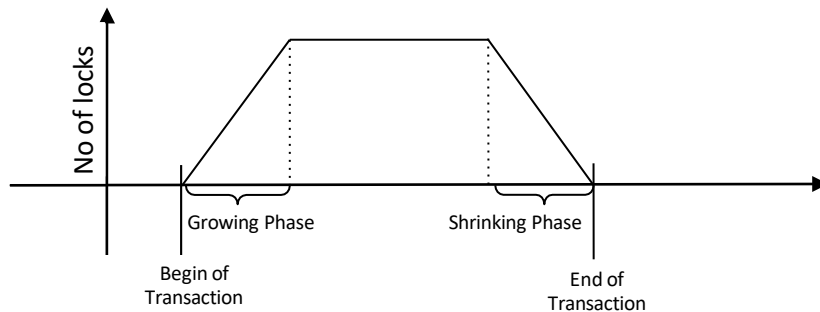
- In pre-claiming Lock Protocol, for each transaction a list is prepared consisting of the data items and type of lock required on each of the data item.
- Before initiating an execution of the transaction, it requests DBMS to issue all the required locks as per the list.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



(3) Two-phase locking (2PL) protocol

- Every transaction execution starts by acquiring few locks or zero locks. During execution it acquire all other required locks one after the other.
-

- When a transaction releases any of the acquired locks then it cannot acquire any more new locks. But, it can only release the acquired locks one after the other during remaining execution of that transaction.



The Two Phase Locking (2PL) has two phases. They are:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released. (Only get new locks but no release of locks).

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired. (Only release locks but no more getting new locks).

Example:

Time	T1	T2
0	<i>LOCK-S(A)</i>	
1		<i>LOCK-S(A)</i>
2	Read(A)	
3		Read(A)
4	<i>LOCK-X(B)</i>	
5	--	
6	Read(B)	
7	B = B + 100	
8	Write(B)	
9	<i>UNLOCK(A)</i>	
10		<i>LOCK-X(C)</i>
11	<i>UNLOCK(B)</i>	--
12		Read(C)
13		C = C + 500
14		Write(C)
15	COMMIT	
16		<i>UNLOCK(A)</i>
17		<i>UNLOCK(C)</i>
18		COMMIT

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

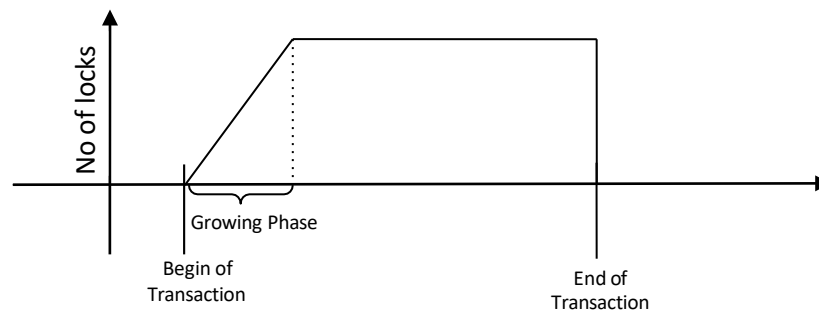
- Growing phase:** from step 1-5 (After first lock onwards)
- Shrinking phase:** from step 10-12 (After first unlock onwards)
- Lock point:** at 5 (No more new locks)

Transaction T2:

- Growing phase:** from step 2-11 (After first lock onwards)
- Shrinking phase:** from step 17-18 (After first unlock onwards)
- Lock point:** at 11 (No more new locks)

(4) Strict Two-phase locking (Strict-2PL) protocol

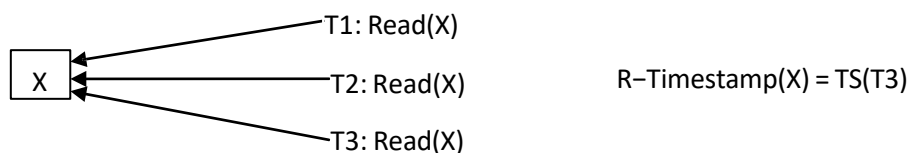
- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



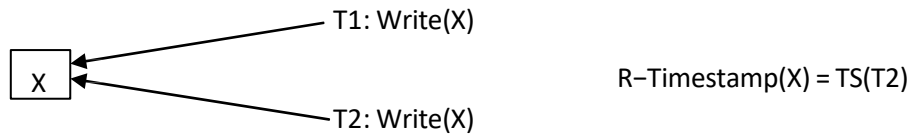
Strict-2PL does not have cascading abort as 2PL does.

10. TIMESTAMP BASED PROTOCOL

- A timestamp is issued to each transaction when it enters into the system.
- It uses either system time or logical counter as a timestamp.
- It is most commonly used concurrency protocol.
- The timestamp of transaction T is denoted as **TS(T)**.
- The system orders the transactions based on their arrival time. For example, let the arrival times of transactions T1, T2 and T3 be 9:00AM, 9:01AM and 9:02AM respectively. Then $TS(T1) < TS(T2) < TS(T3)$. ($9:00AM < 9:01AM < 9:02AM$)
- By using timestamp, the system prepares the serializability order. i.e., $T1 \rightarrow T2 \rightarrow T3$
- The read timestamp of data item X is denoted by **R-timestamp(X)**.
- **R-timestamp(X)**: It is the time stamp of the youngest transaction that performed read operation on X.



- The write timestamp of data item X is denoted by **W-timestamp(X)**.
- **W-timestamp(X)**: It is the time stamp of the youngest transaction that performed write operation on X.



There are mainly two Timestamp Ordering Algorithms in DBMS. They are:

- Basic Timestamp Ordering
- Thomas Write rule

(1). Basic Timestamp Ordering

- Check the following condition whenever a transaction **T_i** issues a **Read (X)** operation:
 - If $W_timestamp(X) > TS(T_i)$ then the operation is rejected.
 - If $W_timestamp(X) \leq TS(T_i)$ then the operation is executed.
(Read is not allowed by T_i , if any younger transactions than T_i write X)
- Check the following condition whenever a transaction T_i issues a **Write(X)** operation:
 - If $TS(T_i) < R_timestamp(X)$ then the operation is rejected. (Write is not allowed by T_i , if any younger transactions than T_i read X)
 - If $TS(T_i) < W_timestamp(X)$ then the operation is rejected and T_i is rolled back otherwise the operation is executed. (Write is not allowed by T_i , if any younger transactions than T_i write X and also T_i should be rolled back and restarted later)

(2) Thomas's Write Rule

Thomas Write Rule is a timestamp-based concurrency control protocol which ignores outdated writes. It follows the following steps:

- (i). If $R_TS(X) > TS(T_a)$, then abort and rollback T_a and reject the operation.

Transaction: T1 <i>Arrival = 9:00 AM</i> $\therefore TS(T1) = 9:00 AM$	Transaction: T2 <i>Arrival = 9:02 AM</i> $\therefore TS(T1) = 9:02 AM$	Variable A Initial A=100
 Write(A) (A=200) ⏟ Reject and Rollback T1	 Read(A) (A=100) : (A=100)	A = 100 (R_TS(A) = 9:02AM) A = 200 100

- (ii). If $W_TS(X) > TS(T_a)$, then don't execute the Write Operation of T_a but continue T_a processing. This is a case of *Outdated or Obsolete Writes*.

Transaction: T1 Arrival = 9:00 AM $\therefore TS(T1) = 9:00 AM$	Transaction: T2 Arrival = 9:02 AM $\therefore TS(T1) = 9:02 AM$	Variable A Initial A=100
Write(A) (A=500) ⏟ Reject but continue T1	Write(A) (A=400) : (A=400)	A = 400 (W_TS(A) = 9:02AM) A = 500 (Outdated write)

- (iii). If the condition in (i) or (ii) is not satisfied, then execute Write(X) of T_a and set $W_TS(X)$ to $TS(T_a)$.

Outdated writes are rejected but the transaction is continued in **Thomas Write Rule** but in Basic TO protocol will reject write operation and terminate such a Transaction.

11. VALIDATION BASED PROTOCOL

In this technique, no concurrency control checking is done while the transaction is under execution. After transaction execution is completed, then only whether concurrency violated or not is checked. It is based on timestamp based protocol. Validation Based Protocol has three phases:

1. **Read phase:** In this phase, the transaction T_a read the value of various data items that are required by T_a and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
1. **Validation phase:** After Transaction T_a execution completed, T_a perform a validation test to determine whether it can copy the temporary local variable values to actual database without causing a violation of serializability.
2. **Write phase:** If the validation of the transaction is successful (valid), then the temporary results are written to the database. Otherwise the temporary local variable values of T_a is ignored and T_a is rolled back.

To perform the validation test, we need to know when the various phases of transaction T_a took place. We shall therefore associate three different timestamps with transaction T_a .

- (i). **Start (T_a):** the time when T_a , started its execution.
- (ii). **Validation (T_a):** the time when T_a finished its execution and started its validation phase.

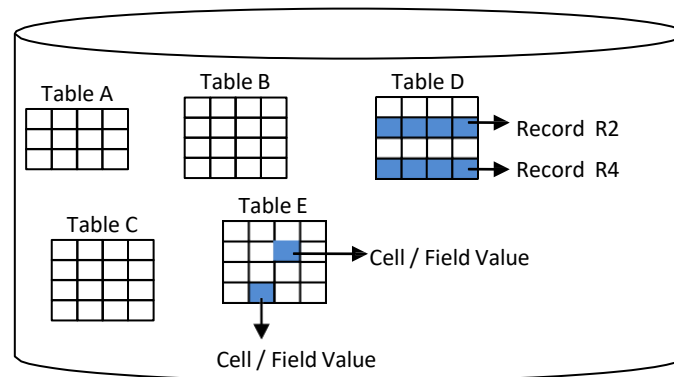
(iii). **Finish (T_a)**: the time when T_a finished its write phase.

The serializability order is determined by changing the timestamp of T as $TS(T) = \text{Validation}(T)$. Hence the serializability is determined at the validation process and cannot be decided in advance. Therefore it ensures greater degree of concurrency while executing the transactions.

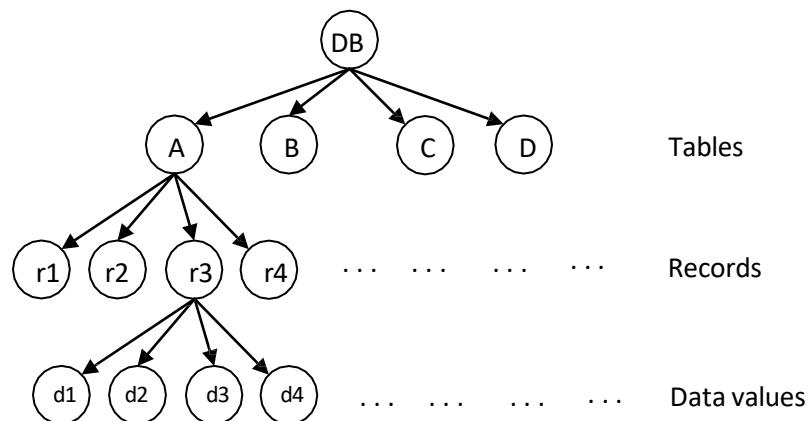
12. MULTIPLE GRANULARITY

The size of data items is often called the **data item granularity**. There exist multiple granularity levels in the DBMS. They are:

- Database
- Table
- Record / row
- Cell / field value



A database contains multiple tables. Each table contains multiple records. Each record contains multiple field values. It is shown in the above figure. For example, consider Table D and Record R2. These two are not mutually exclusive. R2 is a part of D. So granularity means different levels of data where as smaller levels are nested inside the higher levels. Inside database we have tables. Inside table we have records. Inside record we have field values. This can be represented with a tree as shown below.



A lock can be applied at a node, if and only if there does not exist any locks on the decedents (childs and grand childs) of that node. Otherwise lock cannot be applied. If lock is applied on table A, it implies that the lack is also applicable to sub-tree from node A. If lock is applied on database (at root node), it implies the lack is also applicable to all the nodes in the tree.

The larger the object size on which lock is applied, the lower the degree of concurrency permitted. On the other hand, the smaller the object size on which lock is applied, the system has to maintain larger number of locks. More locks cause a higher overhead and needs more disk space. So, what is the best object size on which lock can be applied? It depends on the types of transactions involved. If a typical transaction accesses data values from a record, it is advantageous to have the lock to that one record. On the other hand, if a transaction typically accesses many records in the same table, it may be better to have lock at that table.

Locking at higher levels needs lock details at lower levels. This information is provided by additional types of locks called **intention locks**. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

- (1) **Intention-shared (IS)**: It indicates that one or more shared locks will be requested on some descendant node(s).
- (2) **Intention-exclusive (IX)**: It indicates that one or more exclusive locks will be requested on some descendant node(s).
- (3) **Shared-intention-exclusive (SIX)**: It indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, the shared and exclusive locks, is shown in Figure.

Mode	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Note: In multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

13. RECOVERY AND ATOMICITY

Database needs to be recovered, when the following failures occur.

- (1) Transaction failure
 - (2) System crash
 - (3) Disk failure
- **Transaction failure:** During transaction execution, if it cannot proceed further, then it needs to abort. This is known as transaction failure. A single transaction failure may influence many transactions or processes. The reasons for transaction failure are:
 - **Logical errors:** It occurs due to some code error or an internal condition error.
 - **System error:** It occurs when the DBMS itself terminates an active transaction due to deadlock or resource unavailability.
 - **System crash:** The system may crash due to the external factors such as interruptions in power supply, hardware or software failure. Example: Operating System errors.
 - **Disk failure:** In early days of technology evolution, hard-disk drives or storage drives used to fail frequently. Disk failure occurs due to the formation of bad sectors, disk head crash, un-reachable to the disk or any other failure which destroys all or part of disk storage.

When a system crashes, it may have many transactions being executed and many files may be opened for them. When a DBMS recovers from a crash, it must maintain the following:

- It must check the states of all the transactions that were being executed.
-

- Few transactions may be within the middle of some operation; the DBMS should make sure the atomicity of the transaction during this case.
- It must check for each transaction whether its execution accepted or to be rolled back.
- No transaction is allowed to be in an inconsistent state.

The following techniques facilitate a DBMS in recovering as well as maintaining the atomicity of a transaction:

- Log based recovery
- Check point
- Shadow paging

14. LOG BASED RECOVERY

The log file contains information about the start and end of each transaction and any updates done by the transaction on database items. The log file is saved onto some stable storage so that if any failure occurs, then it can be used to recover the database. The results of all the operation of transaction are first saved in the log and latter updated on the database. The log information is used to recover from system failures.

The log is a sequence of records. It contains the following entries.

- When a transaction T_i starts execution, the log stores: $\langle T_i, \text{Start} \rangle$
- When a transaction T_i modifies an item X from old value V_1 to new value V_2 , the log stores: $\langle T_i, X, V_1, V_2 \rangle$
- When the transaction T_i execution completed, the log stores: $\langle T_i, \text{commit} \rangle$
- When the transaction T_i execution aborted, the log stores: $\langle T_i, \text{abort} \rangle$

Recovery using Log records

When the system is crashed, then the DBMS checks the log to find which transactions needs to be undo and which need to be redo. There are two major techniques for recovery from non-catastrophic transaction failures. They are deferred updates and immediate updates.

- Deferred database modification:** In this technique, all the changes done by the transaction are saved in the system log without modifying the actual database. Once the transaction committed, then only the changes are updated in the database. If a transaction fails before reaching its commit point, it has not changed the database in any way so
-

UNDO is not needed. It may be necessary to REDO the effect of the operations that are recorded in the system log, because their effect not yet written in the database.

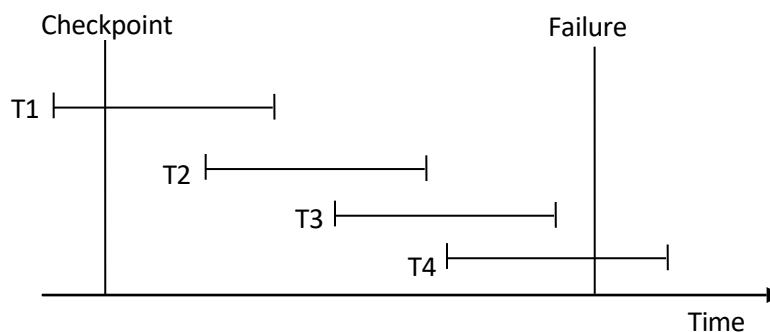
- ii. **Immediate database modification:** In this technique, the database is modified immediately after every operation. However, these operations are recorded in the log file before they are applied to the database, making recovery still possible. If a transaction fails to reach its commit point, the effect of its operation must be undone i.e. the transaction must be rolled back hence we require both undo and redo.

15. CHECKPOINT – (Recovery with Concurrent Transactions)

- In order to recover database from system crashes, all the transaction operations are first saved in the log file and latter updated on the database. The log file is saved in remote location so that it can be used to recover the database. As time passes, the entries in the log file may grow too big. At the time of recovery, searching the entire log file is very time consuming and an inefficient method. To ease this situation, the concept of 'checkpoint' is introduced.
- **Checkpoint** is a mechanism where all the previous log entries are removed from the log file and their results are updated in the database. The checkpoint is like a bookmark.
- During the execution of the transactions, after executing few operations, a check point is created and saved in the log file. Now the log file contains only entries after checkpoint related to new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads the logs backwards from the end to the last checkpoint.
 - It maintains two lists, an undo-list and a redo-list.
-

- If the recovery system sees a log with $\langle T_i, \text{Start} \rangle$ and $\langle T_i, \text{Commit} \rangle$ or just $\langle T_i, \text{Commit} \rangle$, it puts the transaction T_i in the redo-list.

For example: In the log file, transaction T1 have only $\langle T_i, \text{commit} \rangle$ and the transactions T2 and T3 have $\langle T_i, \text{Start} \rangle$ and $\langle T_i, \text{Commit} \rangle$. Therefore T1, T2 and T3 transaction are added to the redo list.

- If the recovery system finds a log with $\langle T_i, \text{Start} \rangle$ but no commit or abort, then it puts the transaction T_i in undo-list.

For example: Transaction T4 will have $\langle T_i, \text{Start} \rangle$. So T4 will be put into undo list since this transaction is not yet complete and failed in the middle.

- All the transactions in the undo-list are then undone and their logs are removed.
- All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

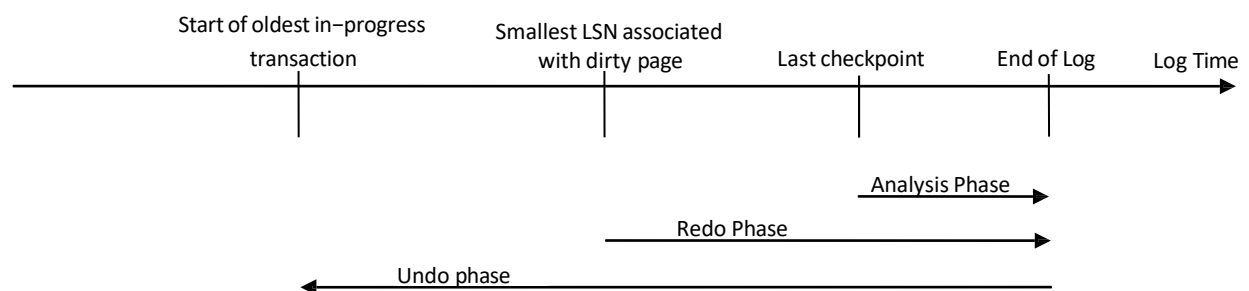
16. ARIES ALGORITHM (Algorithm for Recovery and Isolation Exploiting Semantics)

Algorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics (**ARIES**) is one of the log based recovery method. It uses the Write Ahead Log (WAL) protocol.

Write-ahead logging (WAL): In computer science, **write-ahead logging** (WAL) is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems. The change done by the transactions are first recorded in the log file and written to stable storage at remote location, before the changes are written to the database.

The recovery process of ARIES algorithm has 3 phases. They are:

- (1) Analysis phase
- (2) Redo Phase
- (3) Undo Phase

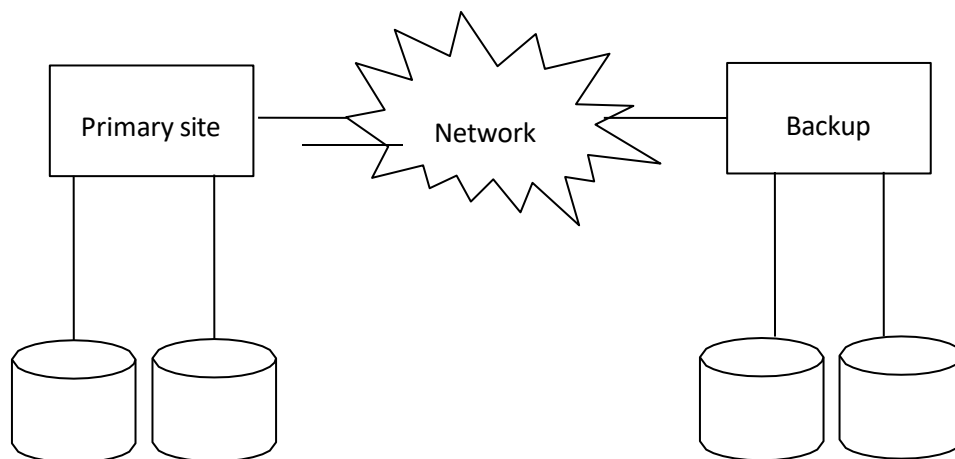


- (1) **Analysis phase:** The recovery subsystem scans the log file forward from the last checkpoint up to the end. The purpose of the scan is to obtain information about the following:
- The starting point from where the redo pass should start.
 - The list of transactions to be rolled back in the undo pass.
 - The list of dirty pages.
- (2) **Redo:** In this phase, the log file is read forward starting from smallest LSN of a dirty page to the end and each update found in the log file is redone. The purpose of this redo pass is to repeat the history to reconstruct the database to the state present at the time of system failure.
- (3) **Undo:** The log is scanned backward and updates related to loser transactions are undone. The 'loser transaction' updates are rolled back in reverse chronological order. If any of the aborted transaction operations are undone, then skip them, no need to undo once again.

17. DATABASE BACKUP

The process of creating duplicate copy of database is called database backup. Backup helps to recover against failure of media, hardware or software failures or any other kind of failures that cause a serious data crash.

Database copy is created and stored in the remote area with the help of network. This database is periodically updated with the current database so that it will be in sync with data and other details. This remote database can be updated manually called offline backup. It can be backed up online where the data is updated at current and remote database simultaneously. In this case, as soon as there is a failure of current database, system automatically switches to the remote database and starts functioning. The user will not know that there was a failure.



Some of the backup techniques are as follows:

- **Full backup or Normal backup:** Full backup is also known as Normal backup. In this, an exact duplicate copy of the original database is created and stored every time the backup made. The advantage of this type of backup is that restoring the lost data is very fast as compared to other. The disadvantage of this method is that it takes more time to backup.
 - **Incremental Backup:** Instead of backup entire database every time, **backup** only the files that have been updated since the last full backup. For this at least weekly once normal backup has to be done. While incremental database backups do run faster, the recovery process is a bit more complicated.
 - **Differential backup:** Differential is similar to incremental backup but the difference is that the recovery process is simplified by not clear the archive bit. So a file that is updated after a normal backup will be archived every time a differential backup is run until the next normal backup runs and clears the archive bit.
-

R25 DATABASE MANAGEMENT SYSTEM

UNIT – V

Data on External Storage, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes, Index data Structures, Hash Based Indexing, Tree base Indexing, Comparison of File Organizations, Indexes and Performance Tuning, Intuitions for tree Indexes, Indexed Sequential Access Methods (ISAM), B+ Trees: A Dynamic Index Structure.

1. DATA ON EXTERNAL STORAGE

Primary memory has limited storage capacity and is volatile. To overcome this limitation, secondary memory is also termed as external storage devices are used. External storage devices such as disks and tapes are used to store data permanently.

The Secondary storage devices can be fixed or removable. Fixed Storage device is an internal storage device like **hard disk** that is fixed inside the computer. Storage devices that are portable and can be taken outside the computer are termed as removable storage devices such as **CD, DVD, external hard disk**, etc.

Magnetic/optical Disk: It supports random and sequential access. It takes less access time.

Magnetic Tapes: It supports only sequential access. It takes more access time.

In DBMS, processing a query and getting output need accessing random pages. So, disks are preferable than magnetic tapes.

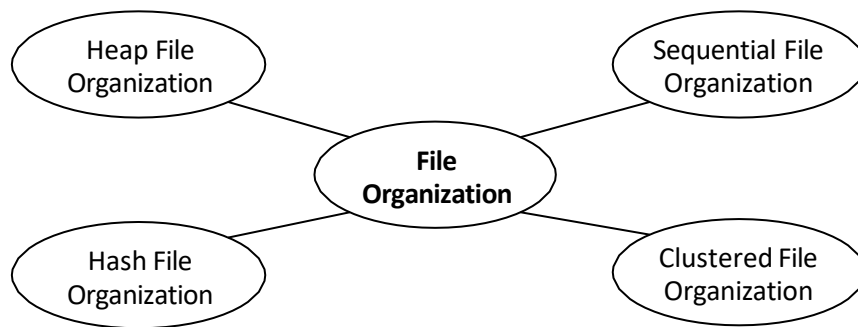
2. FILE ORGANIZATION

The database is stored as a collection of files. Each file contains a set of records. Each record is a collection of fields. For example, a student table (or file) contains many records and each record belongs to one student with fields (attributes) such as Name, Date of birth, class, department, address, etc.

File organization defines how file records are mapped onto disk blocks.

The records of a file are stored in the disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain more than one record. Sometimes, some of the files may have large records that cannot fit in one block. In this case, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record.

The different types of file organization are given below:



Heap File Organization: When a file is created using Heap File Organization mechanism, the records are stored in the file in the order in which they are inserted. So the new records are inserted at the end of the file. In this type of organization inserting new records is more efficient. It uses linear search to search records.

Sequential File Organization: When a file is created using Sequential File Organization mechanism, all the records are ordered (sorted) as per the primary key value and placed in the file. In this type of organization inserting new records is more difficult because the records need to be sorted after inserting every new record. It uses binary search to search records.

Hash File Organization: When a file is created using Hash File Organization mechanism, a hash function is applied on some field of the records to calculate hash value. Based on the hash value, the corresponding record is placed in the file.

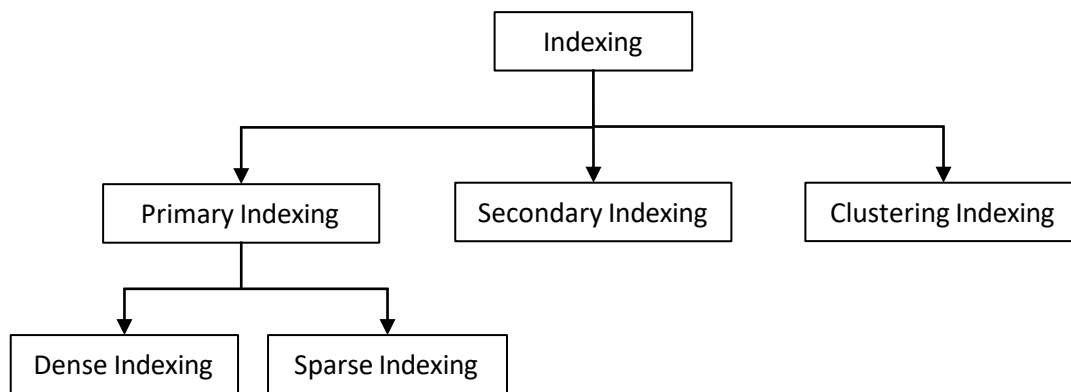
Clustered File Organization: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or search key.

3. INDEXING

If the records in the file are in sorted order, then searching will become very fast. But, in most of the cases, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. It indicates, the records are not in sorted order. In order to make searching faster in the files with unsorted records, indexing is used.

Indexing is a data structure technique which allows you to quickly retrieve records from a database file. An Index is a small table having only two columns. The first column contains a copy of the primary or candidate key of a table. The second column contains a set of disk block addresses where the record with that specific key value stored.

Indexing in DBMS can be of the following types:



i. Primary Index

- If the index is created by using the primary key of the table, then it is known as primary indexing.
- As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.
- The primary index can be classified into two types: dense index and sparse index.

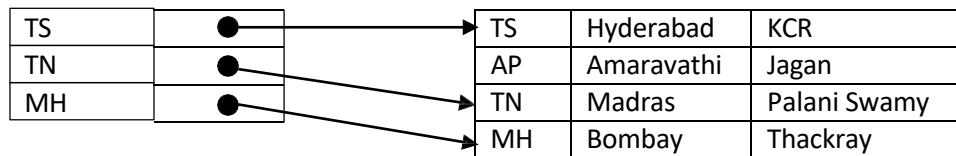
Dense index

- If every record in the table has one index entry in the index table, then it is called dense index.
- In this, the number of records (rows) in the index table is same as the number of records (rows) in the main table.
- As every record has one index entry, searching becomes faster.

TS	●	→	TS	Hyderabad	KCR
AP	●	→	AP	Amaravathi	Jagan
TN	●	→	TN	Madras	Palani Swamy
MH	●	→	MH	Bombay	Thackray

Sparse index

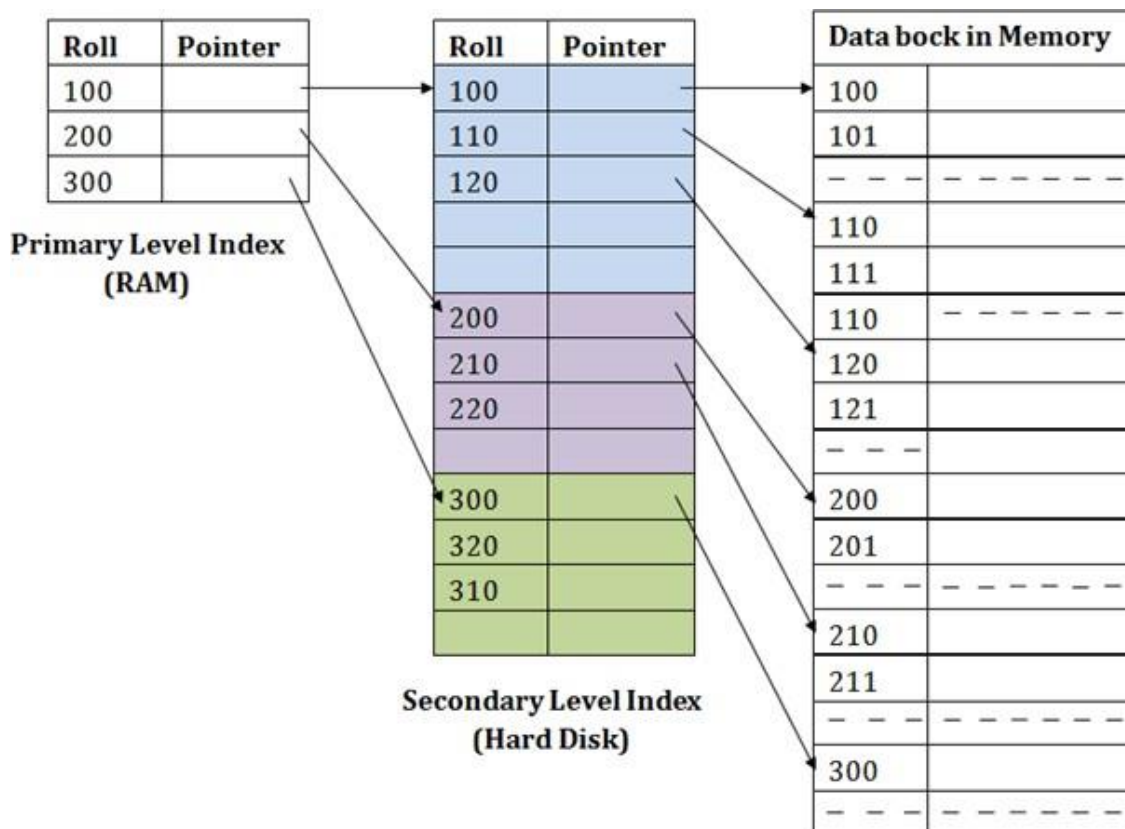
- If only few records in the table have index entries in the index table, then it is called sparse index.
- In this, the number of records (rows) in the index table is less than the number of records (rows) in the main table.
- As not all the record have index entries, searching becomes slow for records that does not have index entries.



ii. Secondary Index

When the size of the main table grows, then size of index table also grows. If the index table size grows then fetching the address itself becomes slower. To overcome this problem, secondary indexing is introduced.

- In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.
- It contains two levels. In the first level each record in the main table has one entry in the first-level index table.
- The index entries in the first level index table are divided into different groups. For each group, one index entry is created and added in the second level index table.

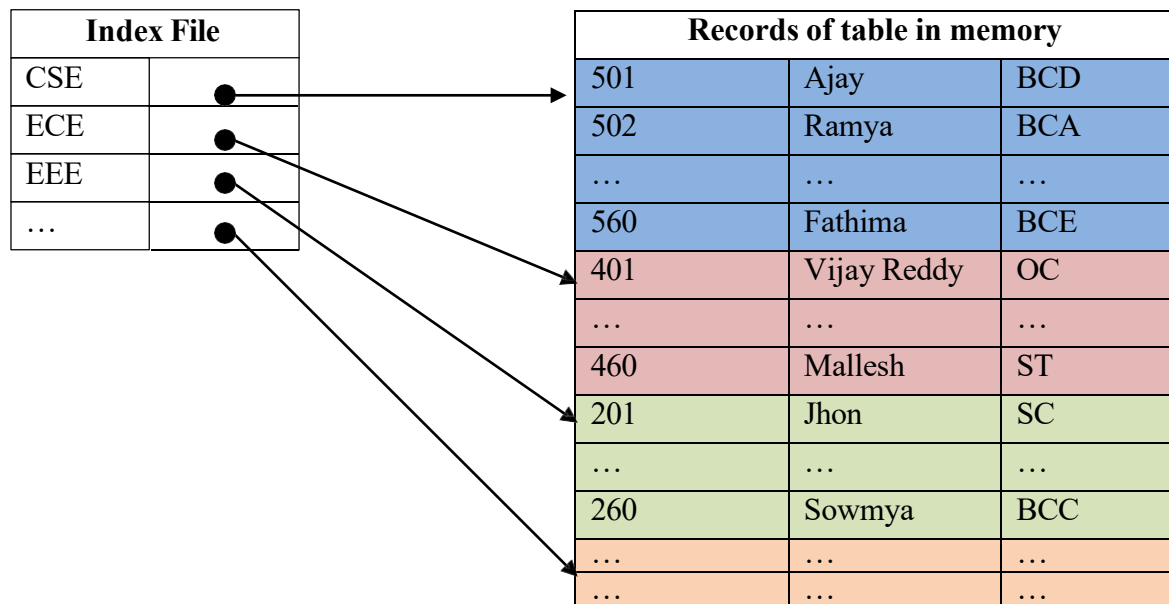


Multi-level Index: When the main table size becomes too large, creating secondary level index improves the search process. Even if the search process is slow; we can add one more level of indexing and so on. This type of indexing is called multi-level index.

iii. Clustering Index

- Sometimes the index is created on non-primary key columns which may not be unique for each record.
- In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
- The records which have similar characteristics are grouped, and indexes are created for these group.

Example: Consider a college contains many students in each department. All the students belong to the same Dept_ID are grouped together and treated as a single cluster. One index pointers point to the one cluster as a whole. The index pointer points to the first record in each cluster. Here Dept_ID is a non-unique key.

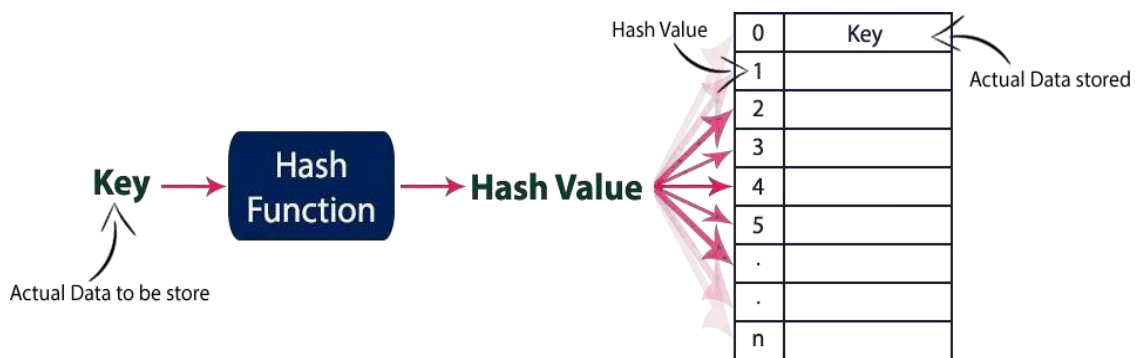


In above diagram we can see that, indexes are created for each department in the index file. In the data block, the students of each department are grouped together to form the cluster. The address in the index file points to the beginning of each cluster.

4. HASH BASED INDEXING

Hashing is a technique to directly search the location of desired data on the disk without using index structure. Hash function is a function which takes a piece of data (key) as input and produces a hash value as output which maps the data to a particular location in the hash table.

The concept of hashing and hash table is shown in the below figure



There are mainly two types of hashing methods:

- i. Static Hashing
- ii. Dynamic Hashing
 - Extended hashing
 - Linear hashing

5. STATIC HASHING

In static hashing, the hash function produce only fixed number of hash values. For example consider the hash function

$$f(x) = x \bmod 7$$

For any value of x , the above function produces one of the hash value from $\{0, 1, 2, 3, 4, 5, 6\}$. It means static hashing maps search-key values to a fixed set of bucket addresses.

Example: Inserting 10, 21, 16 and 12 in hash table.

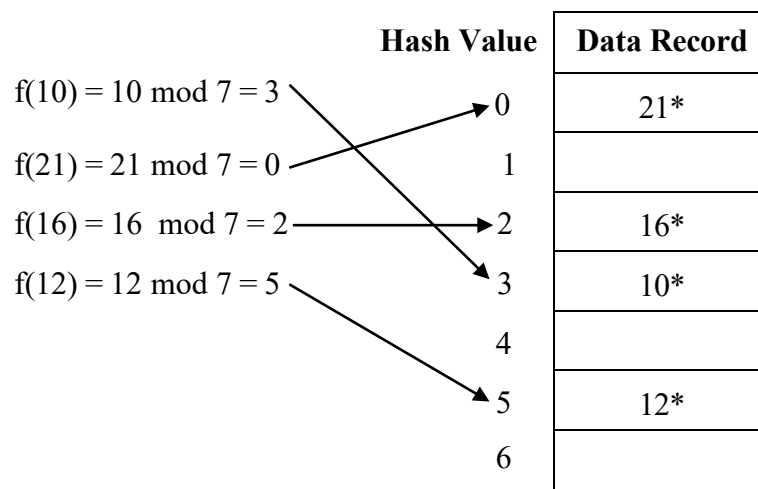


Figure 5.1: Static hashing

Suppose, latter if we want to insert 23, it produce hash value as 2 ($23 \bmod 7 = 2$). But, in the above hash table, the location with hash value 2 is not empty (it contains 16*). So, a collision occurs. To resolve this collision, the following techniques are used.

- Open addressing
- Separate Chaining or Closed addressing

i. **Open Addressing:**

Open addressing is a collision resolving technique which stores all the keys inside the hash table. No key is stored outside the hash table. Techniques used for open addressing are:

- Linear Probing
- Quadratic Probing
- Double Hashing

➤ **Linear Probing:**

In linear probing, when there is a collision, we scan forwards for the next empty slot to fill the key's record. If you reach last slot, then start from beginning.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with 2 is not empty. Then move to next slot (hash value 3), even it is also full, then move once again to next slot with hash value 4. As it is empty store 23 there. This is shown in the below diagram.

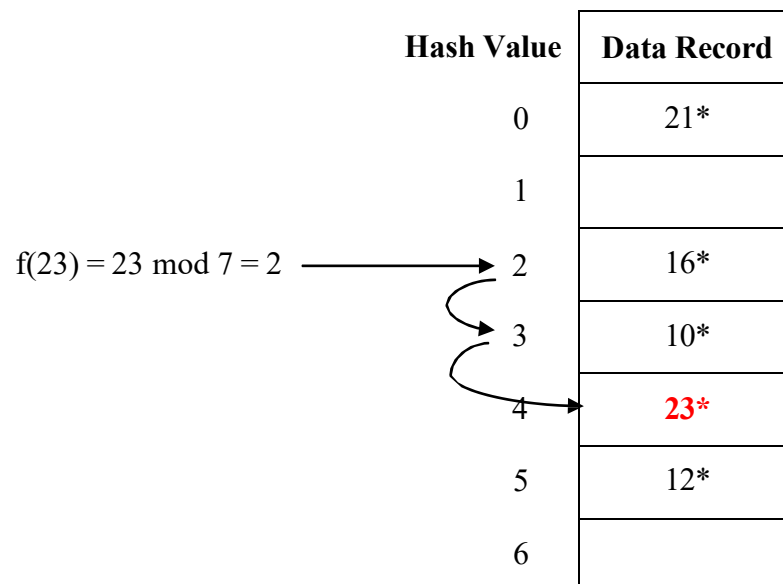


Figure 5.2: Linear Probing

➤ **Quadratic Probing:**

In quadratic probing, when collision occurs, it compute new hash value by taking the original hash value and adding successive values of quadratic polynomial until an open slot is found. If here is a collision, it use the following hash function: $h(x) = (f(x) + i^2) \bmod n$, where $i = 1, 2, 3, 4, \dots$ and $f(x)$ is initial hash value.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with hash value 2 is not empty. Then compute new hash value as $(2 + 1^2) \bmod 7 = 3$, even it is also full, and then once again compute new hash value as $(2 + 2^2) \bmod 7 = 6$. As it is empty store 23 there. This is shown in the below diagram.

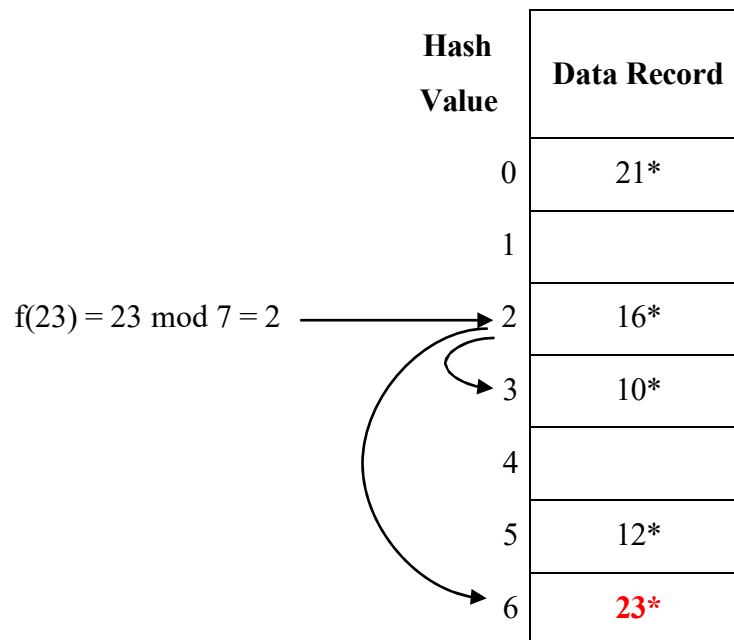


Figure 5.3: Quadratic Probing

➤ **Double Hashing**

In double hashing, there are two hash functions. The second hash function is used to provide an offset value in case the first function causes a collision. The following function is an example of double hashing: $(\text{firstHash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{tableSize}$. Use $i = 1, 2, 3, \dots$

A popular second hash function is : $\text{secondHash}(\text{key}) = \text{PRIME} - (\text{key} \% \text{PRIME})$
where PRIME is a prime smaller than the TABLE_SIZE.

Example: Consider figure 1. When we try to insert 23, its hash value is 2. But the slot with hash value 2 is not empty. Then compute double hashing value as

$$\text{secondHash (key)} = \text{PRIME} - (\text{key} \% \text{PRIME}) \rightarrow \text{secondHash (23)} = 5 - (23 \% 5) = 2$$

$$\text{Double hashing: } (\text{firstHash(key)} + i * \text{secondHash(key)}) \% \text{tableSize} \rightarrow (2 + 1 * 2) \% 7 = 4$$

As the slot with hash value 4 is empty, store 23 there. This is shown in the below diagram.

Hash Value	Data Record
0	21*
1	
2	16*
3	10*
4	23*
5	12*
6	

$f(23) = 23 \bmod 7 = 2$ →

Figure 5.4: Double Probing

ii. Separate Chaining or Closed addressing:

To handle the collision, This technique creates a linked list to the slot for which collision occurs. The new key is then inserted in the linked list. These linked lists to the slots appear like chains. So, this technique is called as *separate chaining*. It is also called as closed addressing.

Example: Inserting 10, 21, 16, 12, 23, 19, 28, 30 in hash table.

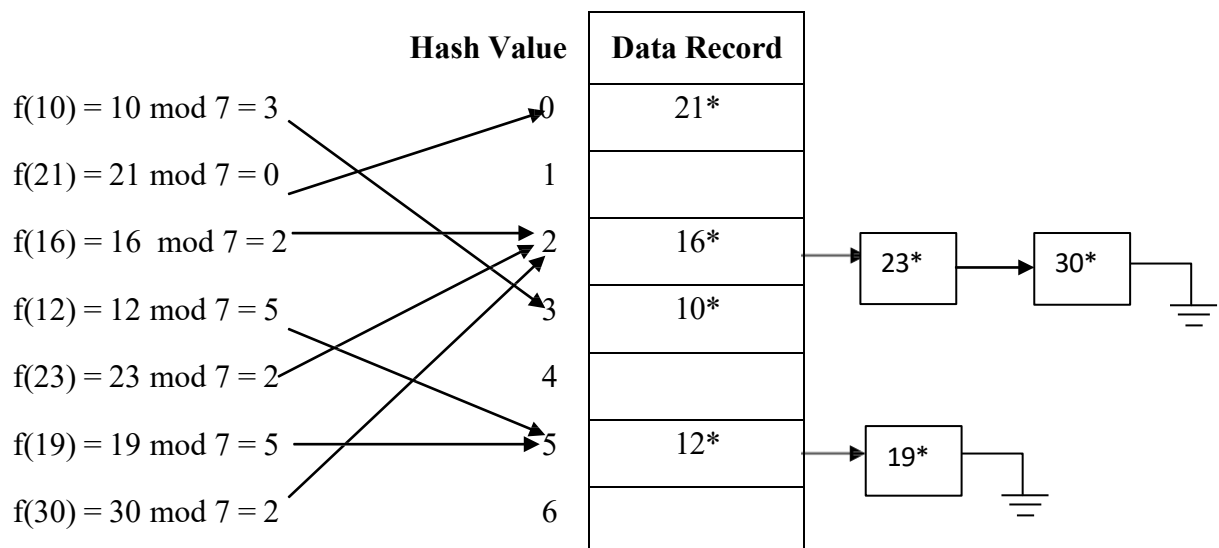


Figure 5.5: Separate chaining example

6. DYNAMIC HASHING

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing can be implemented using two techniques. They are:

- Extended hashing
- Linear Hashing

i. Extendable hashing

In extendable hashing, a separate *directory* of pointers to buckets is used. The number bits used in directory is called global depth (gd) and number entries in directory = 2^{gd} . Number of bits used for locating the record in the buckets is called *local depth*(ld) and each bucket can stores up to 2^{ld} entries. The hash function use last few binary bits of the key to find the bucket. If a bucket overflows, it splits, and if local depth greater than global depth, then the table doubles in size. It is one form of dynamic hashing.

Example: Let global depth (gd) = 2. It means the directory contains four entries. Let the local depth (ld) of each bucket = 2. It means each bucket need two bits to perform search operation. Let each Bucket capacity is four. Let us insert 21, 15, 28, 17, 16, 13, 19, 12, 10, 24, 25 and 11.

21 = 101 01	19 = 100 11
15 = 011 11	12 = 011 00
28 = 111 00	10 = 010 10
17 = 100 01	24 = 110 00
16 = 100 00	25 = 111 01
13 = 011 01	11 = 010 11

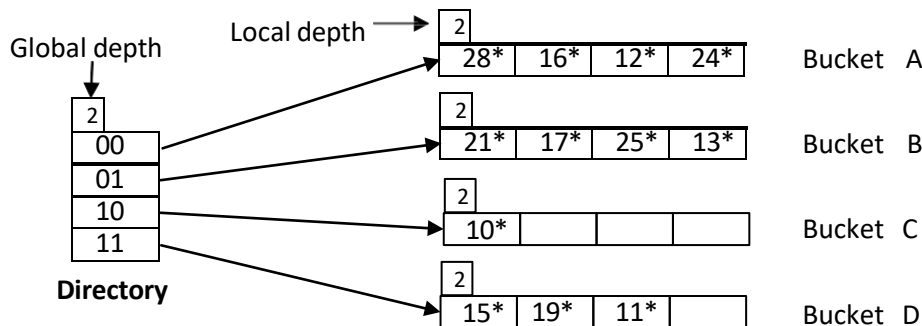


Figure 6.1: Extendible hashing example

Now, let us consider insertion of data entry 20* (binary 101**00**). Looking at directory element 00, we are led to bucket A, which is already full. So, we have to **split** the bucket by

allocating a new bucket and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image' (new bucket). To redistribute entries across the old bucket and its split image, we consider the last *three bits*; the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. That is if a key's last three bits are 000, then it belongs to bucket A and if the last three bits are 100, then the key belongs to Bucket A2. As we are using three bits for A and A2, so the local depth of these buckets becomes 3. This is illustrated in the below Figure 6.2.

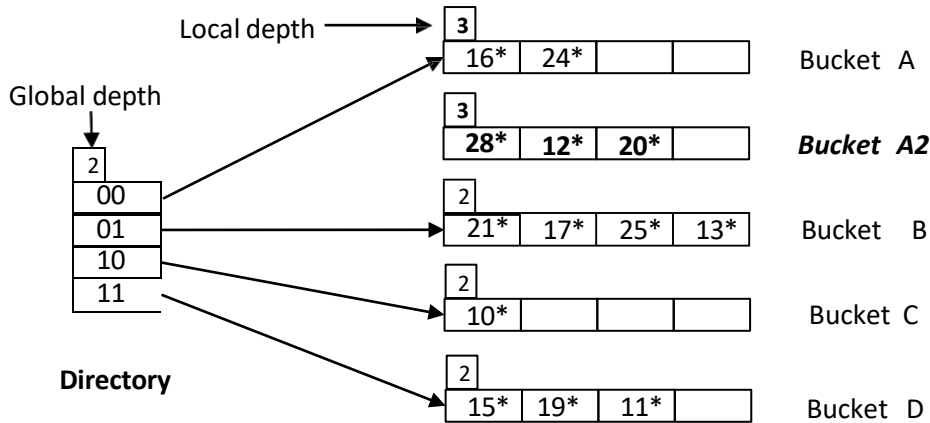


Figure 6.2: After inserting 20 and splitting Bucket A

After split, Bucket A and A2 have local depth greater than global depth ($3 > 2$), so double the directory and use three bits as global depth. As Bucket A and A2 has local depth 3, so they have separate pointers from the directory. But, Buckets B, C and D use only local depth 2, so they have two pointers each. This is shown in the below diagram.

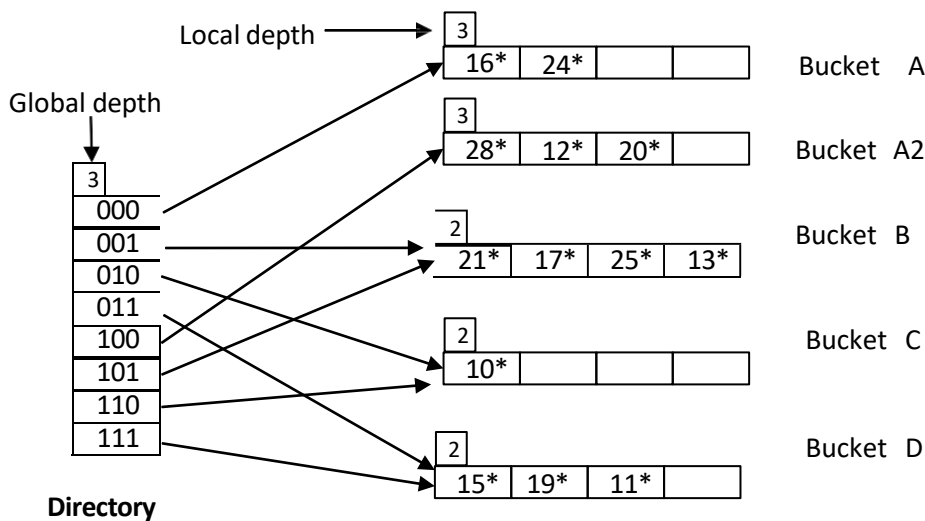


Figure 6.3: After inserting 20 and doubling the directory

An important point that arises is whether splitting a bucket necessitates a directory doubling. Consider our example, as shown in Figure 6.3. If we now insert 9* (01001), it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 101 to point to the bucket and its split image. This is shown in the below diagram. As Bucket B and its split image now have local depth three and it is not greater than global depth. Hence, a bucket split does not necessarily require a directory doubling. However, if either bucket A or A2 grows full and an insert then forces a bucket split, we are forced to double the directory once again.

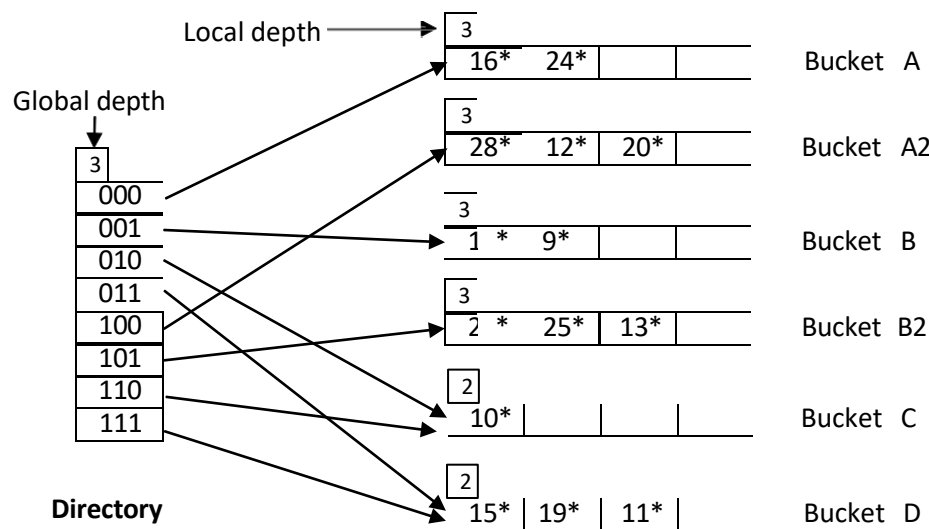


Figure 6.4: After inserting 9

Key Observations:

- A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
- When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
- If new Local Depth of the overflowing bucket is equal to the global depth, only then the directories are doubled and the global depth is incremented by 1.
- The size of a bucket cannot be changed after the data insertion process begins.

ii. Linear Hashing

Linear hashing is a dynamic hashing technique that linearly grows or shrinks number of buckets in a hash file without a directory as used in *Extendible Hashing*. It uses a family of hash functions instead of single hash function.

This scheme utilizes a *family* of hash functions $h_0, h_1, h_2, \dots$, with the property that each function's range is twice that of its predecessor. That is, if h_i maps a data entry into one of N buckets, h_{i+1} maps a data entry into one of $2N$ buckets. One example of such hash function family can be obtained by:
$$h_{i+1}(\text{key}) = \text{key} \bmod (2^i N)$$
 where N is the initial number of buckets and $i = 0, 1, 2, \dots$

Initially it use N buckets labelled 0 through $N-1$ and an initial hashing function $h_0(\text{key}) = \text{key} \% N$ is used to map any key into one of the N buckets. For each overflow bucket, one of the buckets in serial order will be splited and its content is redistributed between it and its split image. That is, for first time overflow in any bucket, bucket 0 will be splited, for second time overflow in any bucket; bucket 1 will be splited and so on. When number of buckets becomes $2N$, then this marks the end of splitting round 0. Hashing function h_0 is no longer needed as all $2N$ buckets can be addressed by hashing function h_1 . In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h_2 will be used. This process is repeated when the hash file grows.

Example: Let $N = 4$, so we use 4 buckets and hash function $h_0(\text{key}) = \text{key} \% 4$ is used to map any key into one of the four buckets. Let us initially insert 4, 13, 19, 25, 14, 24, 15, 18, 23, 11, 16, 12 and 10. This is shown in the below figure.

<i>Bucket#</i>	h_1	h_0	<i>Primary pages</i>				<i>Overflow pages</i>
0	000	00	4*	24*	16*	12*	
1	001	01	13*	25*			
2	010	10	14*	18*	10*		
3	011	11	19*	15*	23*	11*	

Next, when 27 is inserted, an overflow occurs in bucket 3. So, bucket 0 (first bucket) is splited and its content is distributed between bucket 0 and new bucket. This is shown in below figure.

<i>Bucket#</i>	h_1	h_0	<i>Primary pages</i>				<i>Overflow pages</i>
0	000	00	24*	16*			
1	001	01	13*	25*			
2	010	10	14*	18*	10*		
3	011	11	19*	15*	23*	11*	
4	100	00	4*	12*			

27*

Next, when 30, 31 and 34 is inserted, an overflow occurs in bucket 2. So, bucket 1 is splited and its content is distributed between bucket 1 and new bucket. This is shown in below figure.

<i>Bucket#</i>	<i>h₁</i>	<i>h₀</i>	<i>Primary pages</i>	<i>Overflow pages</i>
0	000	00	24* 16*	
1	001	01	13*	
2	010	10	14* 18* 10* 30*	34*
3	011	11	19* 15* 23* 11*	27* 31*
4	100	00	4* 12*	
5	101	01	25*	

When 32, 35, 40 and 48 is inserted, an overflow occurs in bucket 0. So, bucket 2 is splited and its content is distributed between bucket 2 and new bucket. This is shown in below figure.

<i>Bucket#</i>	<i>h₁</i>	<i>h₀</i>	<i>Primary pages</i>	<i>Overflow pages</i>
0	000	00	24* 16* 32* 40*	48*
1	001	01	13*	
2	010	10	18* 10* 34*	
3	011	11	19* 15* 23* 11*	27* 31* 35*
4	100	00	4* 12*	
5	101	01	25*	
6	110	10	14* 30*	

When 26, 20 and 42 is inserted, an overflow occurs in bucket 0. So, bucket 3 is splited and its content is distributed between bucket 3 and new bucket. This is shown in below figure.

<i>Bucket#</i>	<i>h₁</i>	<i>h₀</i>	<i>Primary pages</i>	<i>Overflow pages</i>
0	000	00	24* 16* 32* 40*	48*
1	001	01	13*	
2	010	10	18* 10* 34* 26*	42
3	011	11	19* 11* 27* 35*	
4	100	00	4* 12* 20*	
5	101	01	25*	
6	110	10	14* 30*	
7	111	11	15* 23* 31*	

This marks the end of splitting round. Hashing function h_0 is no longer needed as all $2N$ buckets can be addressed by hashing function h_1 . In new round namely splitting-round 1, bucket split once again starts from bucket 0. A new hash function h_2 will be used. This process is repeated.

7. INTUITIONS FOR TREE INDEXES

We can use tree-like structures as index as well. For example, a binary search tree can also be used as an index. If we want to find out a particular record from a binary search tree, we have the added advantage of binary search procedure, that makes searching be performed even faster. A binary tree can be considered as a **2-way Search Tree**, because it has two pointers in each of its nodes, thereby it can guide you to two distinct ways. Remember that for every node storing 2 pointers, the number of value to be stored in each node is one less than the number of pointers, i.e. each node would contain 1 value each.

The abovementioned concept can be further expanded with the notion of the m-Way Search Tree, where m represents the number of pointers in a particular node. If $m = 3$, then each node of the search tree contains 3 pointers, and each node would then contain 2 values. We use mainly two tree structure indexes in DBMS. They are:

- Indexed Sequential Access Methods (ISAM)
- B+ Tree

8. INDEXED SEQUENTIAL ACCESS METHODS (ISAM)

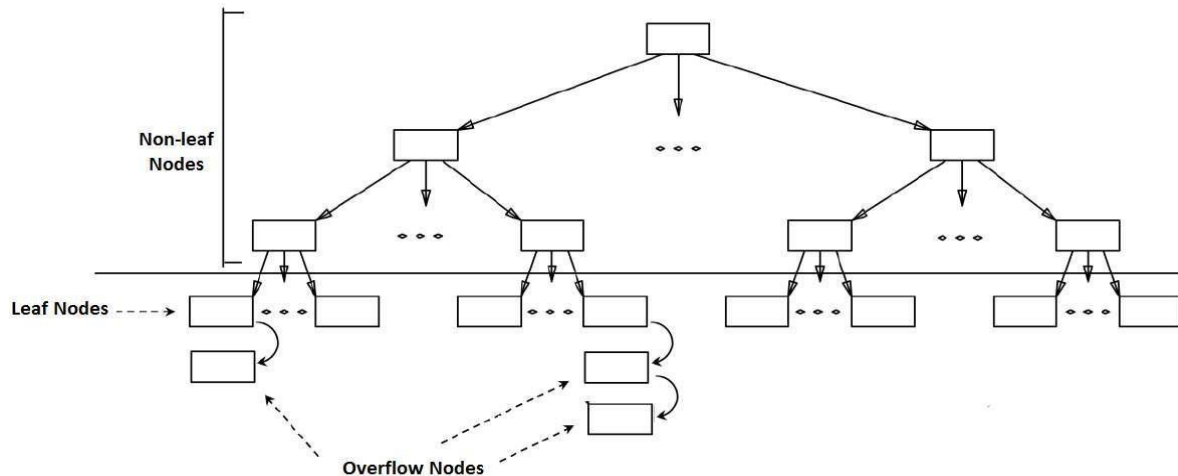
ISAM is a tree structure data that allows the DBMS to locate particular record using index without having to search the entire data set.

- The records in a file are sorted according to the primary key and saved in the disk.
- For each primary key, an index value is generated and mapped with the record. This index is nothing but the address of record.
- A sorted data file according to primary index is called an indexed sequential file.
- The process of accessing indexed sequential file is called ISAM.
- ISAM makes searching for a record in larger database is easy and quick. But proper primary key has to be selected to make ISAM efficient.

- ISAM gives flexibility to generate index on other fields also in addition to primary key fields.

ISAM contain three types of nodes:

- **Non-leaf nodes:** They store the search index key values.
- **Leaf nodes:** They store the index of records.
- **Overflow nodes:** They also store the index of records but after the leaf node is full.



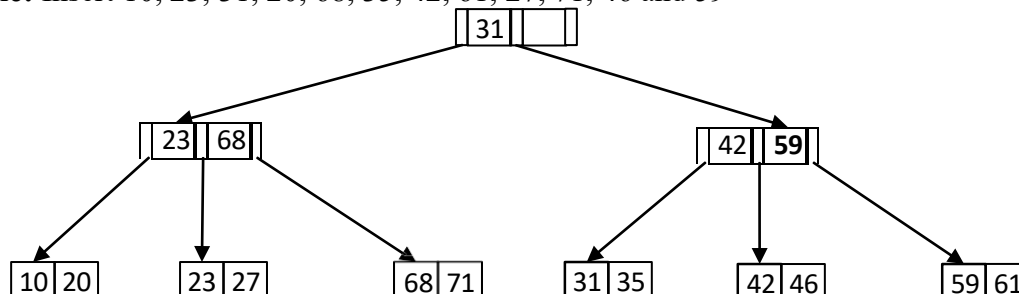
On ISAM, we can perform search, insertion and deletion operations.

Search Operation: It follows binary search process. The record to be searched will be available in the leaf nodes or in overflow nodes only. The non-leaf nodes are used to search the value.

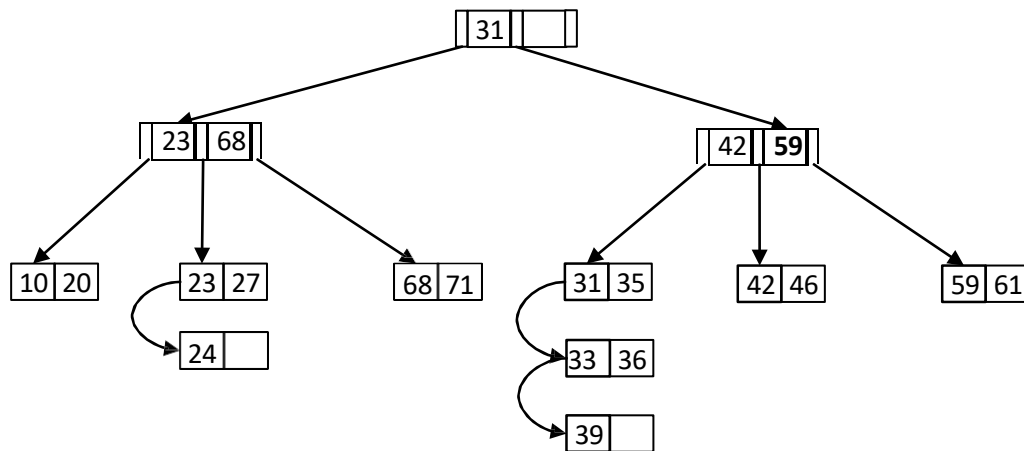
Insertion operation: First locate a leaf node where the insertion to be take place (use binary search). After finding leaf node, insert it in that leaf node if space is available, else create an overflow node and insert the record index in it, and link the overflow node to the leaf node.

Deletion operation: First locate a leaf node where the deletion to be take place (use binary search). After finding leaf node, if the value to be deleted is in leaf node or in overflow node, remove it. If the overflow node is empty after removing the deleted value, then delete overflow node also.

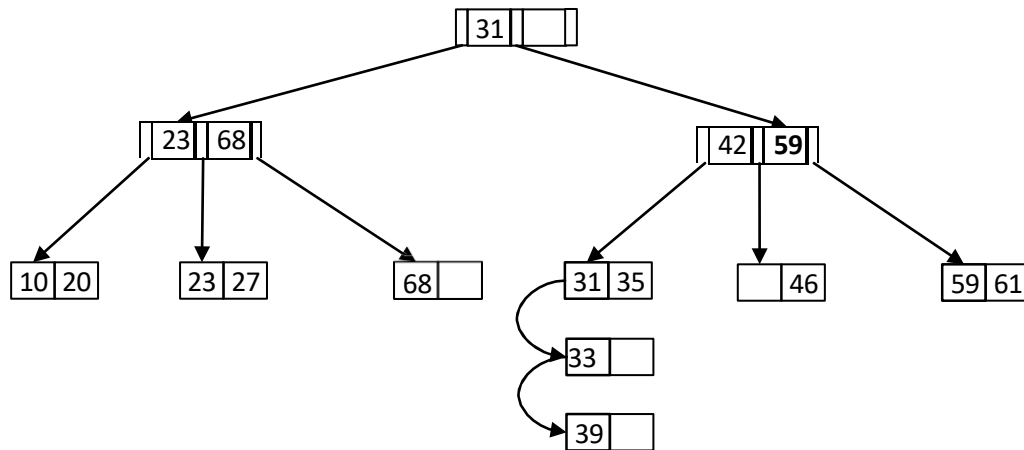
Example: Insert 10, 23, 31, 20, 68, 35, 42, 61, 27, 71, 46 and 59



After inserting 24, 33, 36, and 39 in the above tree, it looks like



Deletion: From the above figure, after deleting 42, 71, 24 and 36



9. B+ TREE

B+ Tree is an extension of Binary Tree which allows efficient insertion, deletion and search operations. It is used to implement indexing in DBMS. In B+ tree, data can be stored only on the leaf nodes while internal nodes can store the search key values.

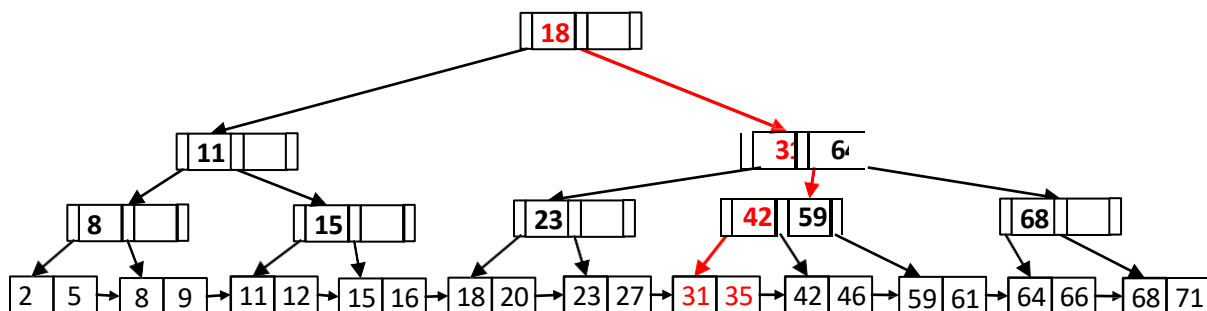
1. B+ tree of an order m can store max $m-1$ values at each node.
2. Each node can have a maximum of m children and at least $m/2$ children (except root).
3. The values in each node are in sorted order.
4. All the nodes must contain at least half full except the root node.
5. Only leaf nodes contain values and non-leaf nodes contain search keys.

B+ Search:

Searching for a value in the B+-Tree always starts at the root node and moves downwards until it reaches a leaf node. The search procedure follows binary tree search procedure.

1. Read the value to be searched. Let us say this value as X .
2. Start the search process from root node
3. At each non-leaf node (including root node),
 - a. If all the values in the non-leaf node are greater than X , then move to its first child
 - b. If all the values in the non-leaf node are less than or equal to X , then move to its last child
 - c. If for any two consecutive values in the non-leaf node, left value is less and right value greater than or equal to X , then move to the child node whose pointer is in between these two consecutive values.
4. Repeat step-3 until a leaf node reaches.
5. At leaf node compare the key with the values in that node from left to right. If the key value is found then display found. Otherwise display it is not found.

Example: Searching for 35 in the below given B+ tree. The search path is shown in red color.



B+ Insertion:

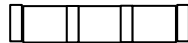
1. Apply search operation on B+ tree and find a leaf node where the new value has to insert.
2. If the leaf node is not full, then insert the value in the leaf node.
3. If the leaf node is full, then

- Split that leaf node including newly inserted value into two nodes such that each contains half of the values (In case of odd, 2nd node contains extra value).
 - Insert smallest value from new right leaf node (2nd node) into the parent node.
Add pointers from these new leaf nodes to their parent node.
 - If the parent is full, split it too. Add the middle key (In case of even, 1st value from 2nd part) of this parent node to its parent node.
 - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers.

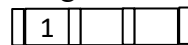
Example: Insert 1,5,3,7,9,2,4,6,8,10 into B+ tree of an order 4.

B+ tree of order 4 indicates there are maximum 3 values in a node.

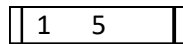
Initially



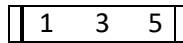
After inserting 1



After inserting 5



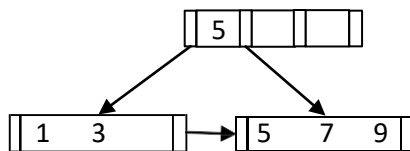
After inserting 3



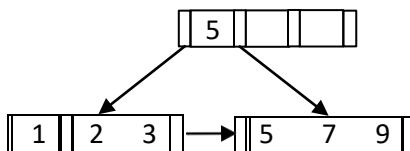
After inserting 7



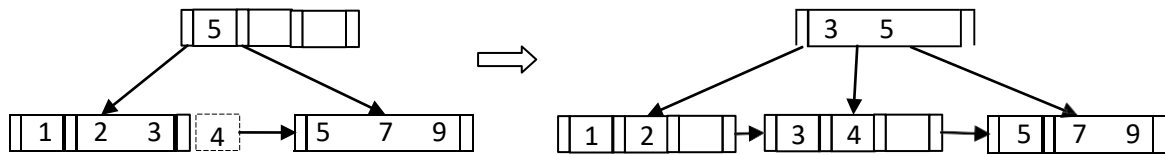
After inserting 9



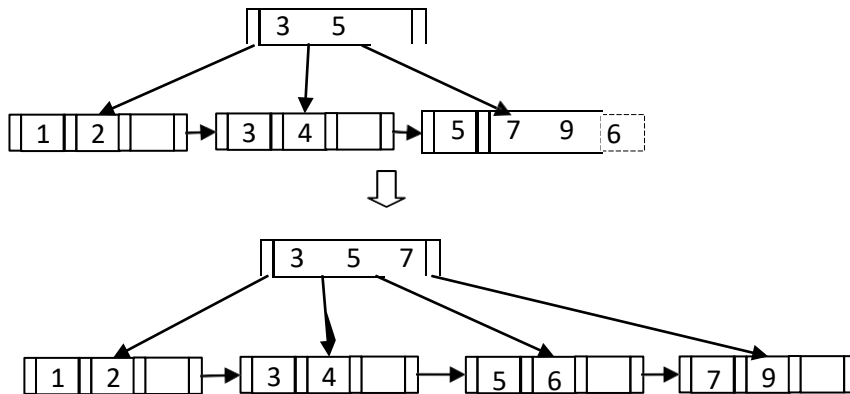
After inserting 2



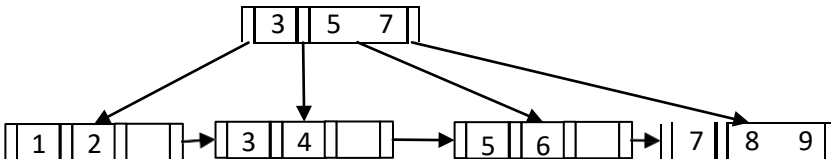
After inserting 4



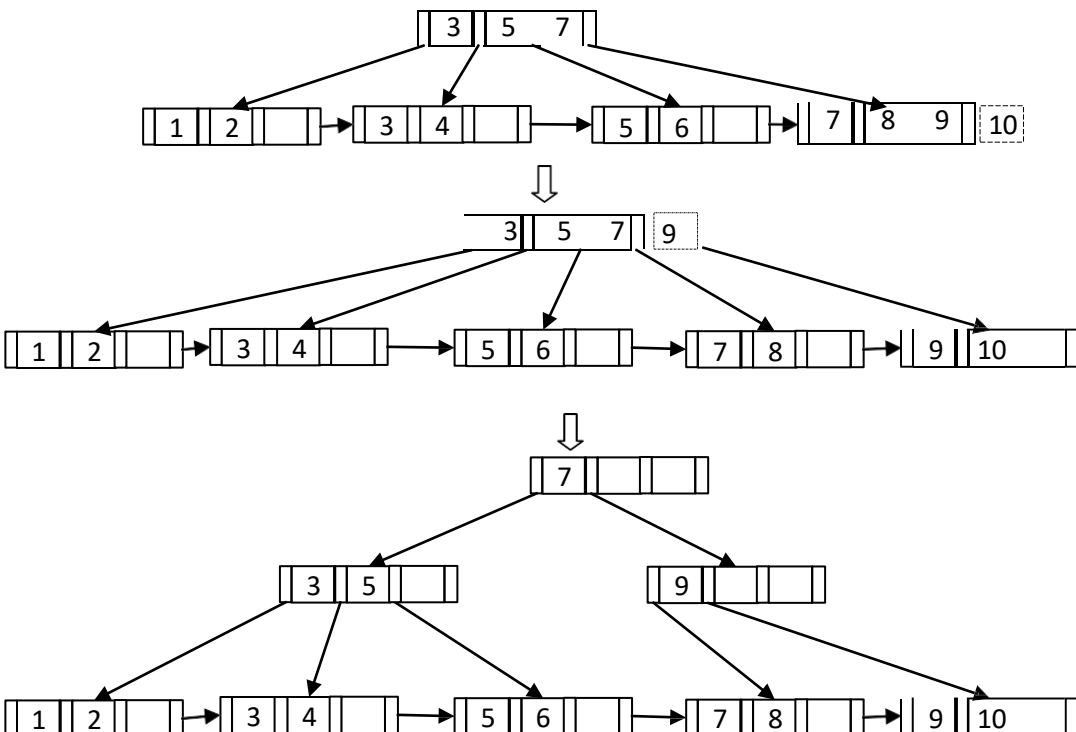
After inserting 6



After inserting 8



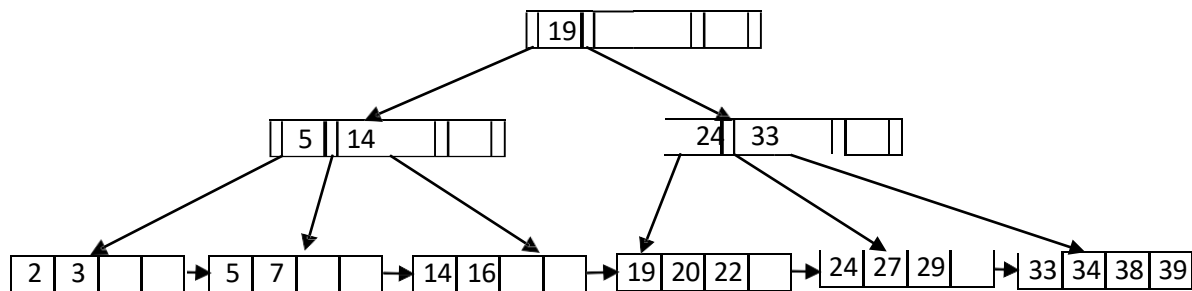
After inserting 10



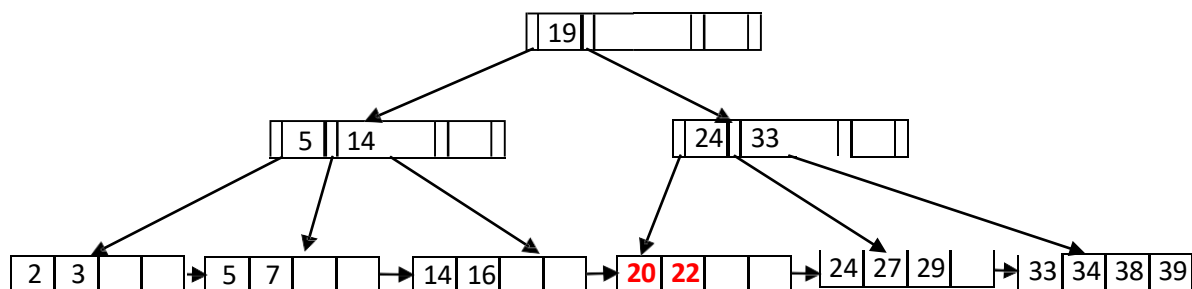
B+ Deletion

- Identify the leaf node L from where deletion should take place.
- Remove the data value to be deleted from the leaf node L
- If L meets the "half full" criteria, then its done.
- If L does not meets the "half full" criteria, then
 - If L's right sibling can give a data value, then move smallest value in right sibling to L (After giving a data value, the right sibling should satisfy the half full criteria. Otherwise it should not give)
 - Else, if L's left sibling can give a data value, then move largest value in left sibling to L (After giving a data value, the left sibling should satisfy the half full criteria. Otherwise it does not give)
 - Else, merge L and a sibling
 - If any internal nodes (including root) contain key value same as deleted value, then delete those values and replace with its successor. This deletion may propagate up to root. (If the changes propagate up to root then tree height decreases).

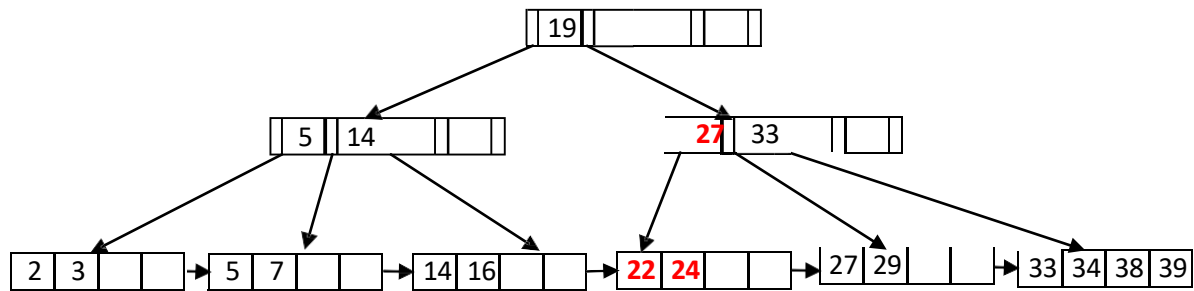
Example: Consider the given below tree and delete 19,



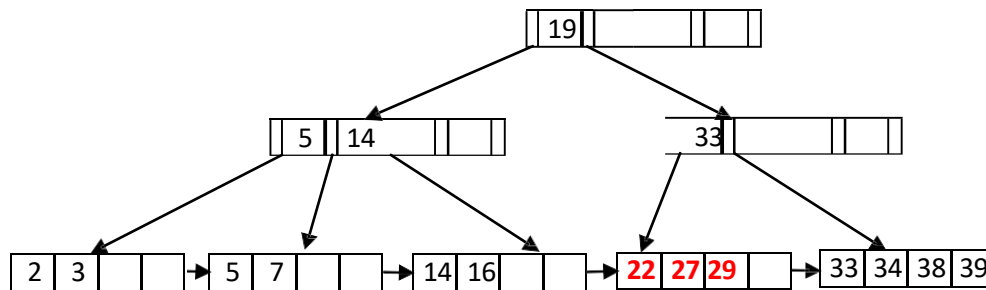
Delete 19 : Half full criteria is satisfied even after deleting 19, so just delete 19 from leaf node



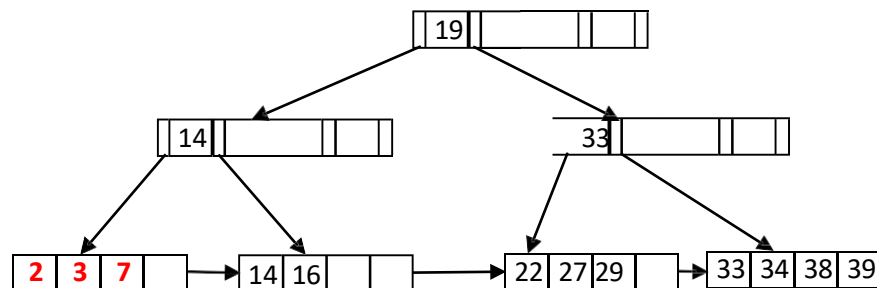
Delete 20: Half full criteria is not satisfied after deleting 20, so bring 24 from its right sibling and change key values in the internal nodes.



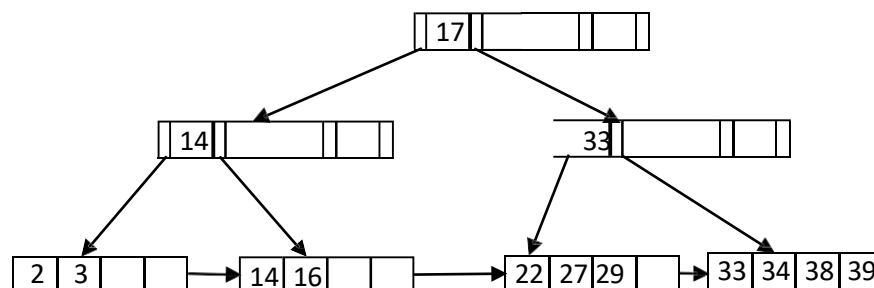
Delete 24: Half full criteria is not satisfied after deleting 24, bringing a value from its siblings also not possible. Therefore merge it with its right sibling and change key values in the internal nodes.



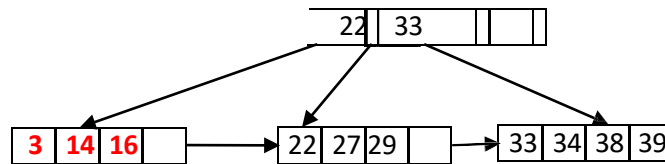
Delete 5: Half full criteria is not satisfied after deleting 5, bringing a value from its siblings also not possible. Therefore merge it with its left sibling (you can also merge with right) and change key values in the internal nodes.



Delete 7: Half full criteria is satisfied even after deleting 7, so just delete 7 from leaf node.



Delete 2: Half full criteria is not satisfied after deleting 2, bringing a value from its siblings also not possible. Therefore merge it with its right sibling and change key values in the internal nodes.



9. INDEXES AND PERFORMANCE TUNING

Indexing is very important to execute DBMS query more efficiently. Adding indexes to important tables is a regular part of performance tuning. When we identify a frequently executed query that is scanning a table or causing an expensive key lookup, then first consideration is if an index can solve this problem. If yes add index for that table.

While indexes can improve query execution speed, the price we pay is on index maintenance. Update and insert operations need to update the index with new data. This means that writes will slow down slightly with each index we add to a table. We also need to monitor index usage and identify when an existing index is no longer needed. This allows us to keep our indexing relevant and trim enough to ensure that we don't waste disk space and I/O on write operations to any unnecessary indexes. To improve performance of the system, we need to do the following:

- Identify the unused indexes and remove them.
- Identify the minimally used indexes and remove them.
- An index that is scanned more frequently, but rarely finds the required answer. Modify the index to reach the answer.
- Identify the indexes that are very similar and combine them.