L	T	P	C
3	0	0	3

(23HPC0504) DATABASE MANAGEMENT SYSTEM

Course Objectives: The main objectives of the course is to

- Introduce database management systems and to give a good formal foundation on the relational model of data and usage of Relational Algebra
- Introduce the concepts of basic SQL as a universal Database language
- Demonstrate the principles behind systematic database design approaches by covering conceptual design, logical design through normalization
- Provide an overview of physical design of a database system, by discussing Database indexing techniques and storage techniques

Course Outcomes: After completion of the course, students will be able to

- Understand the basic concepts of database management systems (L2)
- Analyze a given database application scenario to use ER model for conceptual design of the database (L4)
- Utilize SQL proficiently to address diverse query challenges (L3).
- Employ normalization methods to enhance database structure (L3)
- Assess and implement transaction processing, concurrency control and database recovery protocols in databases. (L4)

UNIT I: Introduction: Database system, Characteristics (Database Vs File System), Database Users, Advantages of Database systems, Database applications. Brief introduction of different Data Models; Concepts of Schema, Instance and data independence; Three tier schema architecture for data independence; Database system structure, environment, Centralized and Client Server architecture for the database.

Entity Relationship Model: Introduction, Representation of entities, attributes, entity set, relationship, relationship set, constraints, sub classes, super class, inheritance, specialization, generalization using ER Diagrams.

Unit II: Relational Model: Introduction to relational model, concepts of domain, attribute, tuple, relation, importance of null values, constraints (Domain, Key constraints, integrity constraints) and their importance, Relational Algebra, Relational Calculus. BASIC SQL: Simple Database schema, data types, table definitions (create, alter), different DML operations (insert, delete, update).

UNIT III: SQL: Basic SQL querying (select and project) using where clause, arithmetic & logical operations, SQL functions(Date and Time, Numeric, String conversion).Creating tables with relationship, implementation of key and integrity constraints, nested queries, sub queries, grouping, aggregation, ordering, implementation of different types of joins, view(updatable and non-updatable), relational set operations.

UNIT IV: Schema Refinement (Normalization):Purpose of Normalization or schema refinement, concept of functional dependency, normal forms based on functional dependency Lossless join and dependency preserving decomposition, (1NF, 2NF and 3 NF), concept of surrogate key, Boyce-Codd normal form(BCNF), MVD, Fourth normal form(4NF), Fifth Normal Form (5NF).

UNIT V: Transaction Concept: Transaction State, ACID properties, Concurrent Executions,

Serializability, Recoverability, Implementation of Isolation, Testing for

Serializability, lock based, time stamp based, optimistic, concurrency protocols, Deadlocks,Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Introduction to Indexing Techniques:Primary Indexing,Secondary indexing,hash based indxing, static and dynamic tree base indexing, B+ Trees, B+Trees applications towards database.

Textbooks:

1. Database Management Systems, 3rd edition, Raghurama Krishnan, Johannes Gehrke, TMH (For Chapters 2, 3, 4)

2. Database System Concepts,5th edition, Silberschatz, Korth, Sudarsan,TMH (For

Chapter 1 and Chapter 5)

Reference Books:

1. Introduction to Database Systems, 8thedition, C J Date, Pearson.

2. Database Management System, 6th edition, RamezElmasri, Shamkant B. Navathe, Pearson

3. Database Principles Fundamentals of Design Implementation and Management,Corlos Coronel, Steven Morris, Peter Robb, Cengage Learning.

Web-Resources:

1. https://nptel.ac.in/courses/106/105/106105175/
https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_01275806667282022456_shared/overview

UNIT 1- DATABASE MANAGEMENT SYSTEM

Database Management System(DBMS):-

A Database Management System(DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information.

Database Users:-

There are 4 different types of database system users, differentiated by the way they except to interact with the system. Different types of user interfaces have been designed for the different types of users.

1. Naive Users:
   Naive users are unsophisticated users who interact with the system by invoking one of the applications programs that have been written previously. The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naïve users may also simply read reports generated from the database.
2. Application Programmers:-
   Application Programmers are computer professionals who write application programs. Application Programmers can choose from many tools to develop user interfaces. Rapid Application Development(RAD) tools are tools that enable an application programmer to construct forms and reports with minimal programming effort.
3. Sophisticated Users:-
   Sophisticated Users interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysis who submit queries to explore data in the database fall in this category.
4. Specialized Users:-
   Specialized Users are sophisticated users who write specialized database applications that do not fit into the traditional data, processing framework. Among these applications are computer – aided design systems, knowledge base and expert systems, systems that store data with complex data types(for example, graphics data and audio data) and environment – modelling systems.

Advantages of Database System:-

Using a DBMS to manage data has many advantages:

- **Data Independece:-** Application programs should not ideally, be exposed to details of data representation and storage. The DBMS provides an abstract view of the data, that hides such details.

- **Efficient Data Access:-** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data Integrity and Security:-** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, it can enforce access controls that govern what data is visible to different classes of users.
- **Data Administrators:-** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different group of users use it can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.
- **Concurrent Access and Crash Recovery:-** A DBMS schedules concurrent access to the data as being accessed by only one user at a time. Further, the DBMS protects users, from the effects of system failures.

Database Applications:-

Database are widely used. Here are some representative applications:

Enterprise Information:-

- **Sales:-** For customer, product and purchase information.
- **Accounting:-** For payments, receipts, account balances, assets and other accounting information.
- **Human resources:-** For information about employee, salaries, payroll taxes, and benefits and for generation of pay checks.
- **Manufacturing:-** For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and order for items.
- **Online retailers:**- For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.

Banking and Finance:-

- **Banking:-** For customer information, accounts, loans and banking transactions.
- **Credits and Transactions:-** For purchases on credit cards and generation of monthly statements.
- **Finance:-** For storing information about holdings, sales and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customer and automated trading by the firm.

Universities:- For student information, course registrations and grades(in addition to standard enterprise information such as human resources and accounting).

Airlines:- For reservations and schedule information. Airlines were among the first to use databases in geographically distributed manner.

Telecommunication:- For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

Brief Introduction of different Data Models:-

The structure of a database is the data model, a collection of conceptual tools for describing data, data relationships, data semantics, consistency and constraints. A data model provides a way to describe the designs of a database at the physical, logical and view level.

There are a number of different data models that are classified into 4 categories:

- **Relational Model:-** The Relational Model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each columns has a unique name. Tables are also known as relations. The relational model is an example of a record-based model.
- **Entity – Relationship Model:-** The Entity – Relationship(E-R) data mode uses a collection of basic objects called entities, and relationships among these objects. An entity is a 'thing' or 'object' in the real world that is distinguishable from other objects.
- **Object – Based Data Model:-** Object – oriented programming (especially in java, c++ or c#) has become the dominant software – development methodology. This led to the development of an object – oriented data mode that can be seen as extending the E-R model with notions of encapsulation, method(functions) and object identity.
- **Semistructured Data Model:-** The Semistructured data model permits the specification of data where individual data items of the same type may have different set of attributes. The Extensible Markup Language (XML) is widely used to represent semistructured data.

Concepts of Schema, Instances and Data Independence:-

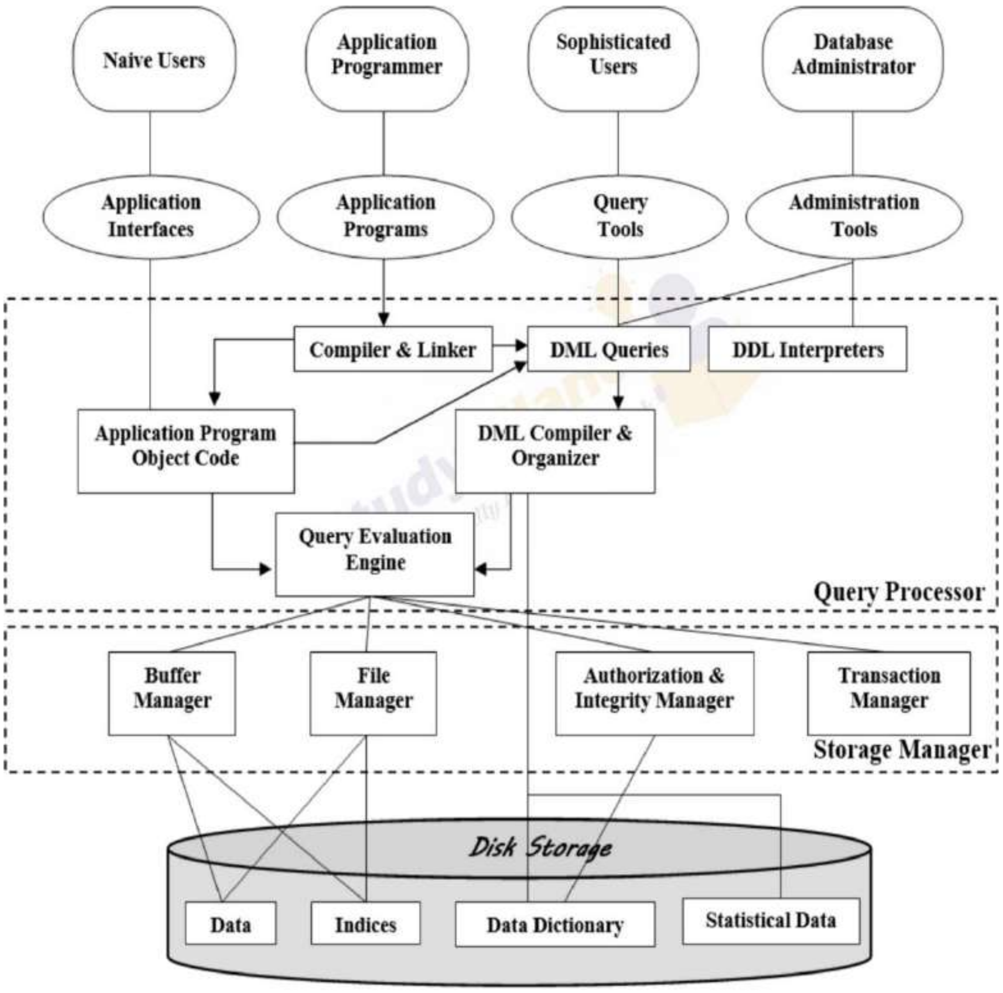
Schema:- The overall design of the database is called the database schema.

Database systems have several schema's, partitioned according to the levels of abstraction. The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level.

Instances:- Database change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance.

Data Independence:- A database system is a collection of interrelated data and set of programs that allows users to access and modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

<u>Database System Architecture:</u>



<u>Naive Users</u>: Naive users are unsophisticated users who interact with the system by invoking one of the applications programs that have been written previously. The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naïve users may also simply read reports generated from the database.

<u>Application Programmers</u>:-Application Programmers are computer professionals who write application programs. Application Programmers can choose from many tools to develop user interfaces. Rapid Application Development(RAD) tools are tools that enable an application programmer to construct forms and reports with minimal programming effort.

Sophisticated Users:- Sophisticated Users interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysis who submit queries to explore data in the database fall in this category.

Database Administrators:- One of the main reasons for using DBMS is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a database administrator.

Query Processor:-

The query processor components includes:-

- DDL interpreter:- Which interprets DDL statements
  and records the definitions in the data dictionary.
- DML compiler:- Which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
  A query can usually be translated into any of a number of alternative evaluation plans that all given the same result. The DML compiler also performs query optimization; that is, it picks the lowest cost evaluation plan from among the alternatives.
- Query evaluation engine:- Which executed low-level instructions generated by the DML compiler.

Storage Manager:-

The Storage Manager is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

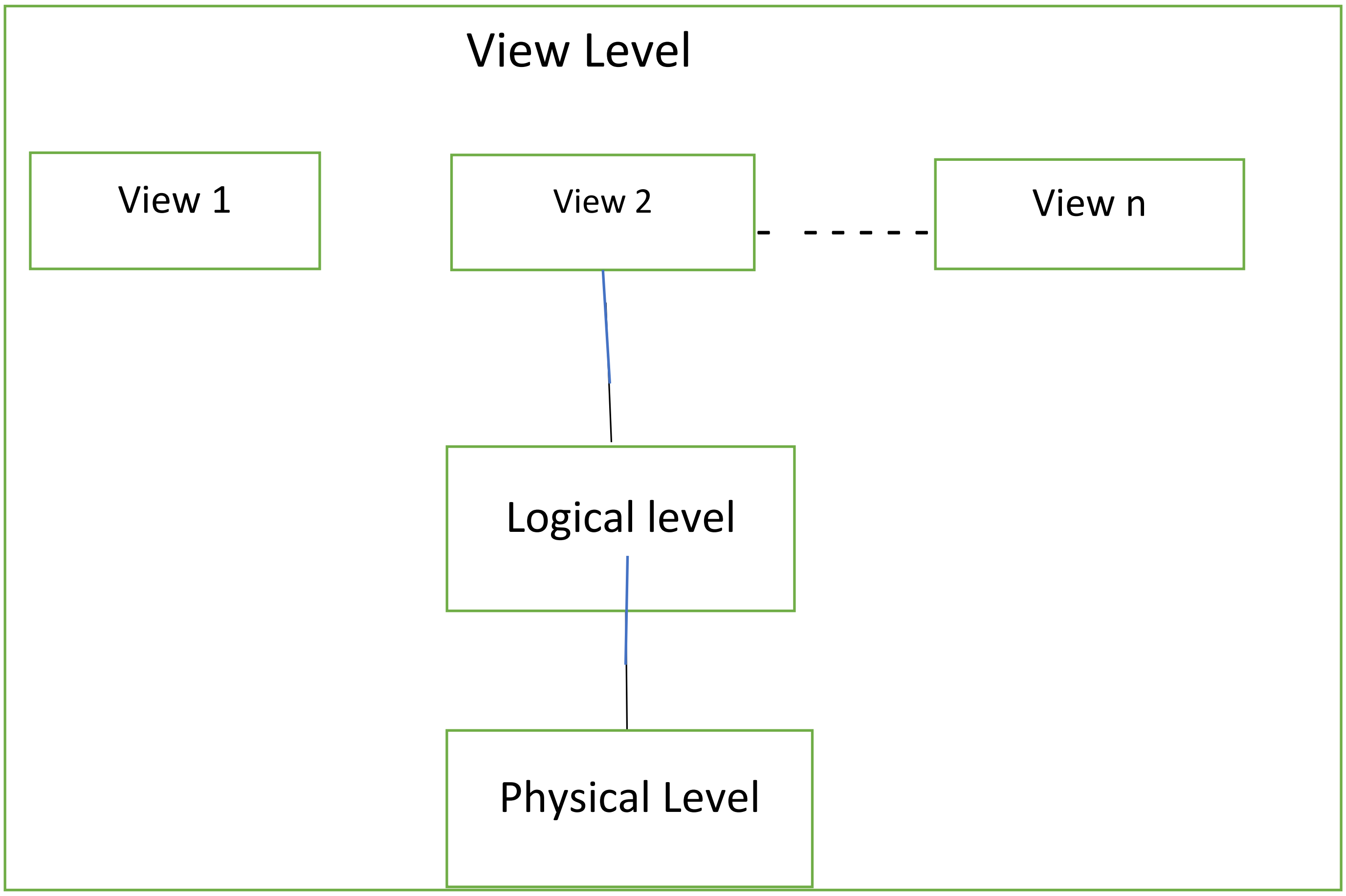
The storage manager components include:-

- **Authorization and Integrity Manager: -** Authorization and Integrity Manager, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction Manager: -** Which ensures that the database remains in a consistent state despite system failures and that concurrent transactions executions proceed without conflicting.
- **File Manager:-** Which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer Manager:-** Which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation: -

- **Data Files: -** which store the database itself.
- **Data Dictionary: -** which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices: -** which can provide fast access to data item. Like the index in the textbook, a database index provides pointers to those data items that hold a particular value.
- **Statistical Data:-** It refers to the metadata and metrics collected about the database's performance, usage and structure. This data is used to optimize query performance, storage and system management.

Three Tier Schema Architecture for data independence: -



Physical Level: - The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.
Logical Level: - The next higher level of abstraction describes what data are stored in the database and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures.
View Level: - The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many view for the same database.

Entity Relationship Model

<u>Introduction:</u>

The Entity – Relationship (E – R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.

<u>E – R Model: -</u>

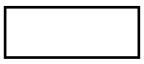
The Entity – Relationship(E-R) Model is a conceptual representation of the data structures and relationships in a database. It's a graphical representation that uses entities, attributes and relationship to describe the organization of data

Strong Entity: -

A strong entity is an entity that has a unique identifier and exists independently of other entities. It is an entity that can identified by its own attributes, without relying on another entity.

Characteristics of a Strong Entity: -

1. Has a primary key (unique identifier)
2. It is represented by
3. It Exists independently.
4. Not dependent on another entity for its existence.
5. It can be created, updated and deleted independently.

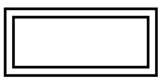


Weak Entity: -

A weak entity is an entity that depends on another entity. It has no primary key of its own. It has a foreign key that references the strong entity's primary key.

Charateristics:-

1. It has no unique identifier (primary key)
2. It is represented by
3. It is dependent on a strong entity for existence.
4. It has a foreign key referencing the strong entity.
5. It cannot be created, updated or deleted independently.



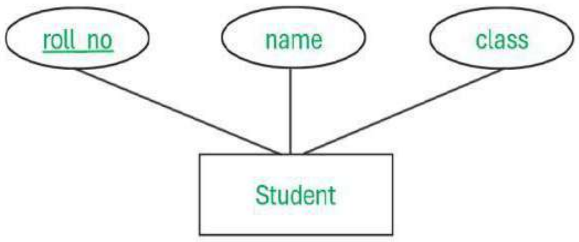
Attributes: -

Attribute is an characteristic (or) property of an entity that describes it or defines its state. Attributes are used to provide more details about an entity and help to differentiate.

<u>Types of Attributes: -</u>

1. **Simple Attribute:** - It is a type of attribute that has only one value that cannot be further divided and we cannot insert the null values. They provide basic information about entities.

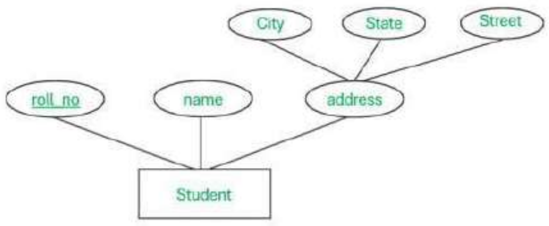
Ex: Name, class, Roll_no, DOB, etc.



2.Composite Attribute: - It is a type of attribute that has multiple values of components. And can be further divided into smaller attributes.

Ex: 1. Address: 2. Name:

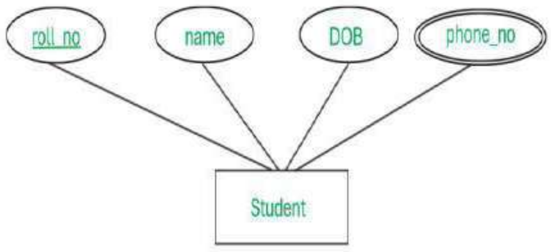
-street	-First name
-city	-Middle name
-state	-Last Name
-ZIP code	



3. Multi-Valued Attribute: -

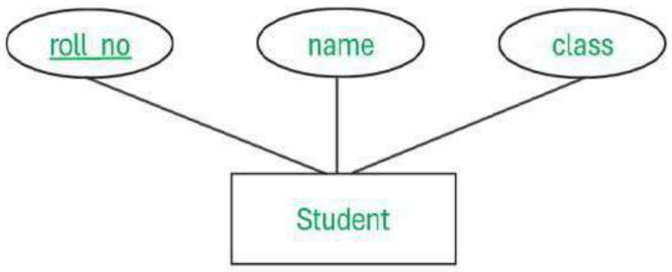
It is a type of attribute that can have multiple values for a single entity. It represents the collection of values for a property of an entity. Allows an entity to have more than one value for the attribute.

Ex: Phone Numbers – A person can have multiple phone numbers(e.g; home, work)



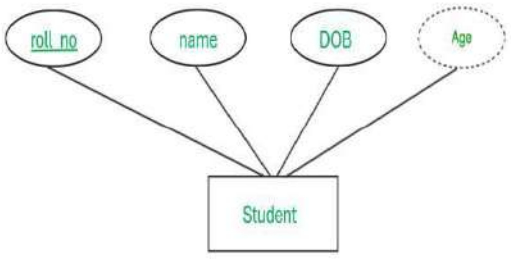
4. **Single – Valued Attribute: -**

It is a type of attribute that has only one value for a single entity. Represents a property of an entity. It cannot have multiple values for the same entity.



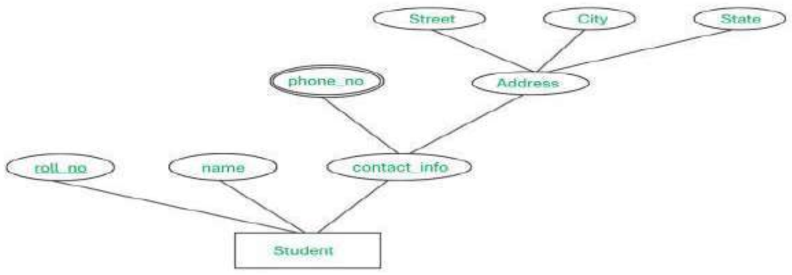
Ex: Roll_no, Name, Class

5. **Derived Attribute:-**
It is a type of attribute that is calculated or derived from other attributes. Does not store its own value. Depends on the values of other attributes.



Ex: Age

6. Complex Attribute:-It is a type of attribute that is a combination of composite attribute and multi-valued attribute. It consists of multiple components of values and has a complex data structure.



Relationship Sets: -

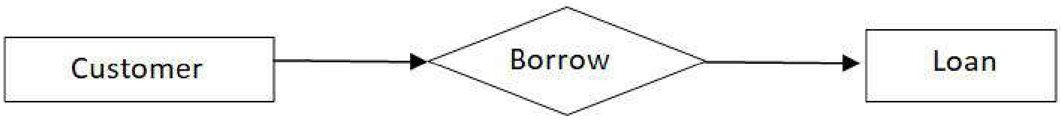
Relationship set in a Database Management System (DBMS) is essential as it provides the ability to store, recover, and oversee endless sums of information effectively in cutting-edge data administration, hence making a difference in organizations.
In a Relational database, relationship sets are built up by utilizing keys, such as primary and foreign keys, to interface related records over distinctive tables.
Understanding and appropriately characterizing relationship sets is fundamental for planning a well-organized and utilitarian database framework, guaranteeing the exact representation and administration of information.

What is a Relation?

A Relation in a database management system (DBMS) organizes information into rows and columns. This organized arrangement makes a difference in information storing and recovering information proficiently. Relations permit us to query information utilizing SQL commands like SELECT, Update, Insert, and Delete. Henceforth, it becomes easier and more demanding to control and extract data from the database.
Relations are alluded to as tables within the database management system.

What is a Relationship Set?

It is a set of Relationships of the Same type. Mathematical Relation on m > 2 (Possibly non-distinct) entity Sets.
If E1, E2, ... En are the entity sets for the Relationship Set R is a subset of {(e1, e2,... en) | e1 € E1, e2 € E2....en € En} Where (e1, e2,...en) is a Relationship.
For Example, (E-R Diagram)



Relationship Set:

Characteristics of Relationship Set

- Degree**:** Degree of a relationship set indicates to the number of properties related with the relationship set.
- Arity**:** Arity of a relationship set indicates the number of taking part relations. It can be like double (including two relations), ternary (including three relations), and so on.
- Cardinality**:** Cardinality characterizes the number of occurrences or records that can be related with each substance on both sides of the relationship.

This permits for the execution of different sorts of connections, such as one-to-one, one-to-many, and many-to-many, reflecting real-world associations between substances.

Degree of Relationship Set

It denotes the number of entity sets that belong to that particular relationship set is called the degree of that relationship set.

Degree of a Relationship Set = the number of entity set that belongs to that particular relationship set

Relationship Set Example with Tables
Let 'Customer' and 'Loan' entity sets defines the Relationship 'set Borrow' to denote the association between Customer and bank loans.

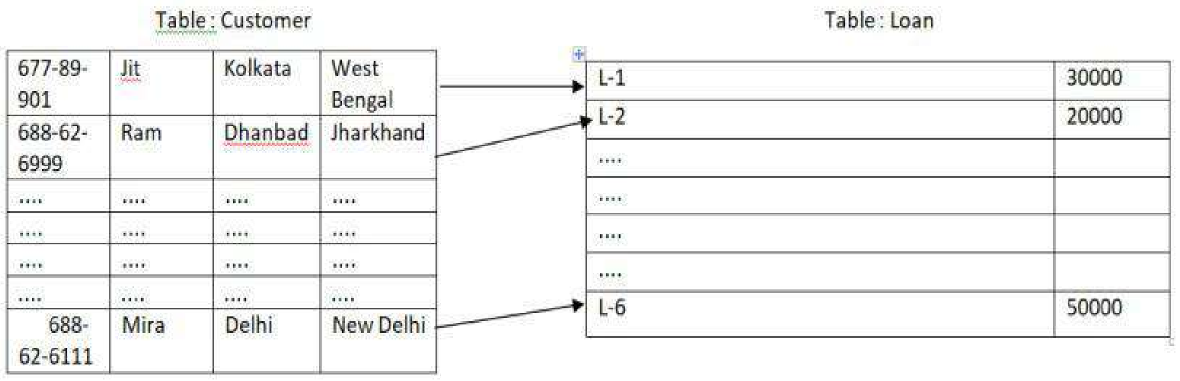


Table : Customer

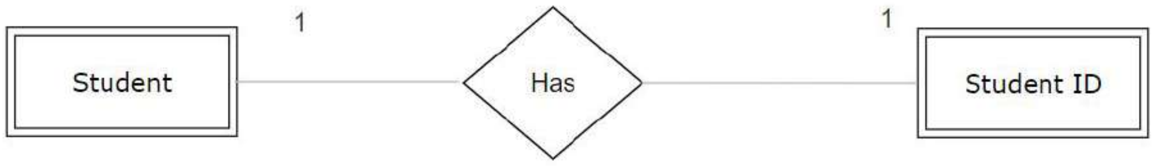
677-89-901	Jit	Kolkata	West Bengal
688-62-6999	Ram	Dhanbad	Jharkhand
....	....	....	....
....	....	....	....
....	....	....	....
....	....	....	....
688-62-6111	Mira	Delhi	New Delhi

Table : Loan

L-1	30000
L-2	20000
....	
....	
....	
....	
L-6	50000

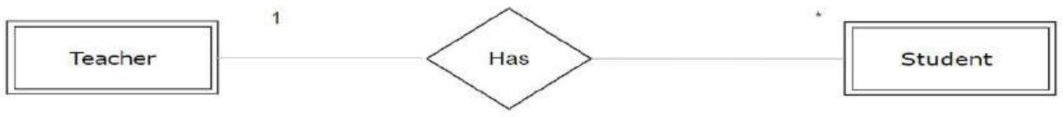
Relationship Set between Customer and Loan

In the above example, in table Customer shows a customer borrows a loan from Table Loan. In other words, the relationship borrows will be one to one.
Examples
1**.** Student-Student ID Relationship
In a one-to-one relationship, each student is assigned a unique student ID



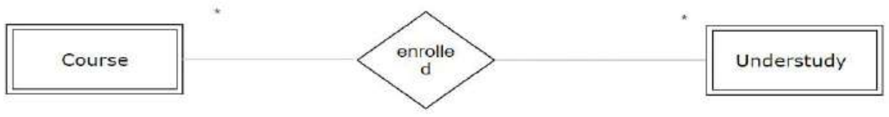
one to one

2. Teacher-Student Relationship
Consider two relations: Teacher and Student. The relationship set between them can be characterized as "mentors." Each teacher mentors in one class, but a class can have numerous students. So one teacher can have many students.
This is often an illustration of a one-to-many relationship.



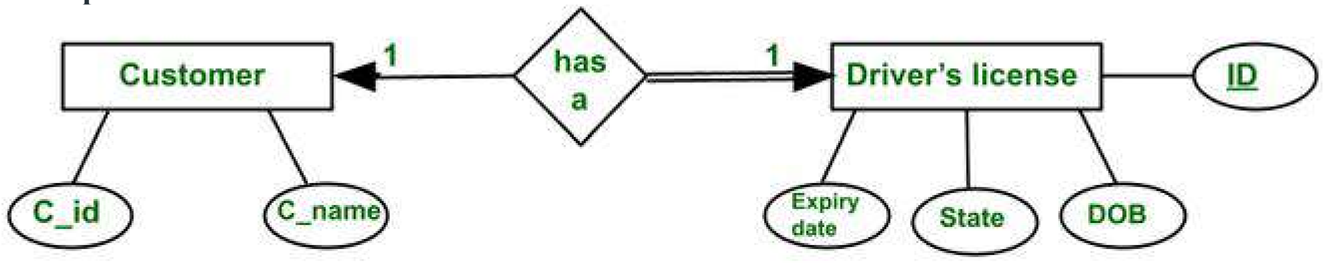
one to many

3. Student-Course Relationship
In this situation, we have two relations: Understudy and Course. The relationship set between them can be characterized as “enrolled_in.” Each understudy can be selected in numerous courses, and each course can have different understudies.
This can be an illustration of a many-to-many relationship.



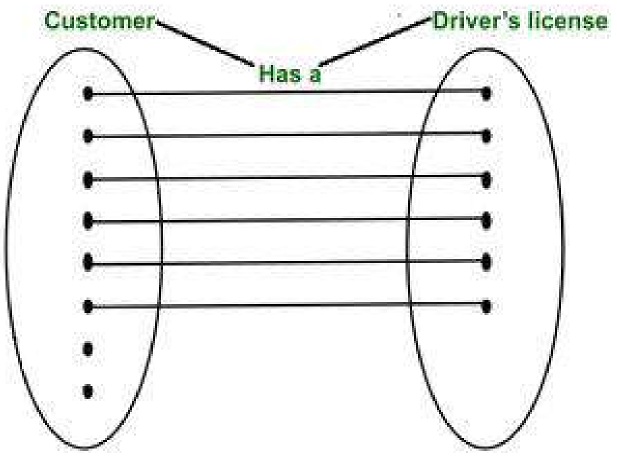
many to many

1. One to one relationship (1:1)
It is represented using an arrow(⇢,⇠)(There can be many notations possible for the ER diagram).
Example:



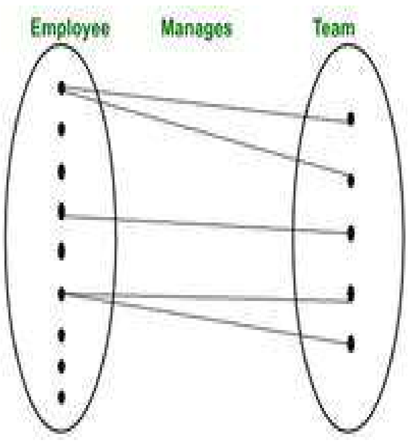
One to One relationship

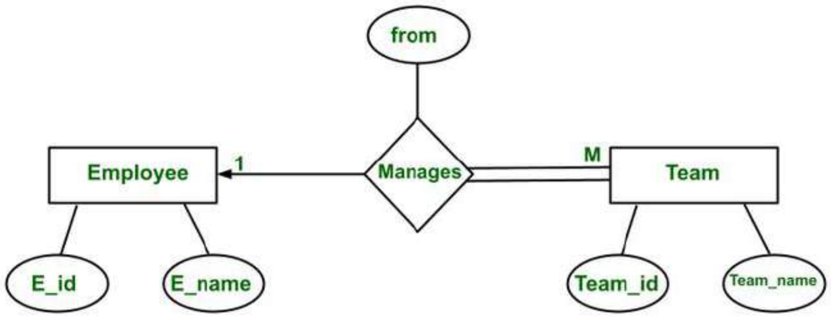
In this ER diagram, both entities customer and driving license having an arrow which means the entity Customer is participating in the relation “has a” in a one-to-one fashion. It could be read as ‘Each customer has exactly one driving license and every driving license is associated with exactly one customer.
The set-theoretic perspective of the ER diagram is



There may be customers who do not have a credit card, but every credit card is associated with exactly one customer. Therefore, the entity customer has total participation in a relation.

2. One to many relationship (1:M)
Example:

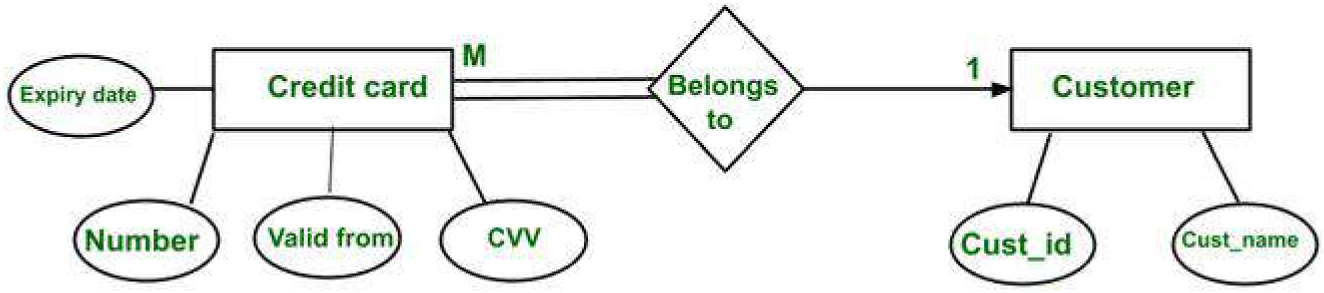




one to many relationship

This relationship is one to many because "There are some employees who manage more than one team while there is only one manager to manage a team".

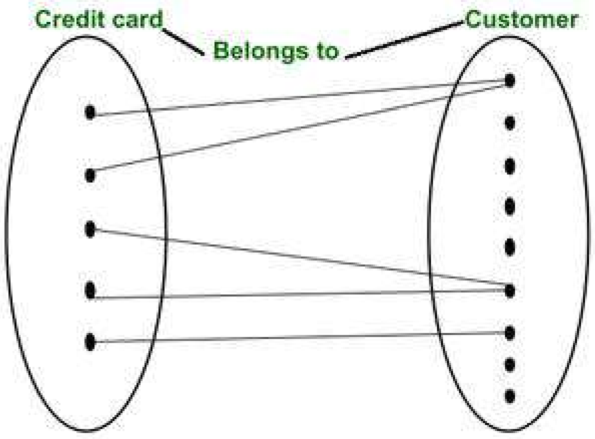
3. Many to one relationship (M:1)
Example:



Many to one realtionship

It is related to a one-to-many relationship but the difference is due to perspective. Any number of credit cards can belong to a customer and there might be some customers who do not have any credit card, but every credit card in a system has to be associated with an employee (i.e. total participation). While a single credit card can not belong to multiple customers.

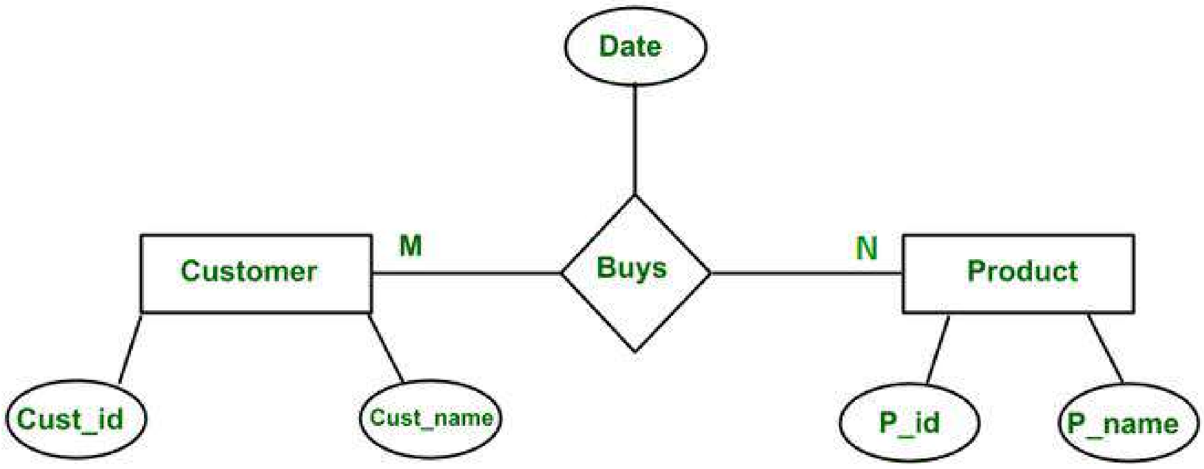
The set-theoretic perspective of the ER diagram is:



4. Many to many relationship (M:N)

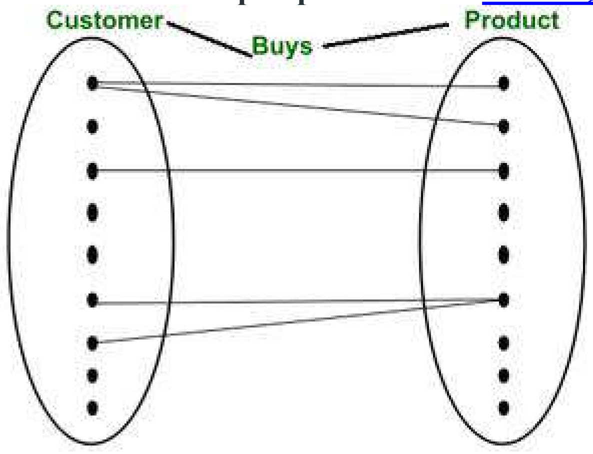
Example:

A customer can buy any number of products and a product can be bought by many customers.



Many to many Relationship

The set-theoretic perspective of the ER diagram is:

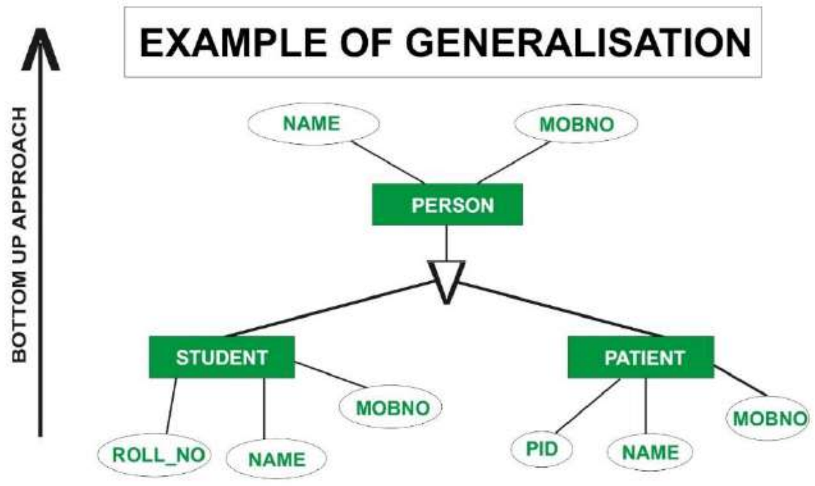


Any of the four cardinalities of a binary relationship can have both sides partial, both total, and one partial, and one total participation, depending on the constraints specified by user requirements.

Generalization

Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher-level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher-level entity called PERSON as shown in Figure 1. In this case, common attributes like P_NAME, and P_ADD become part of a higher entity (PERSON), and specialized attributes like S_FEE become part of a specialized entity (STUDENT).

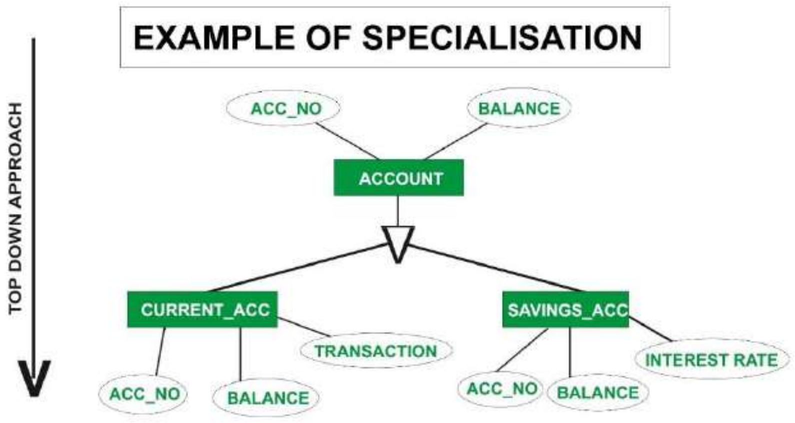
Generalization is also called as " Bottom-up approach".



Specialization

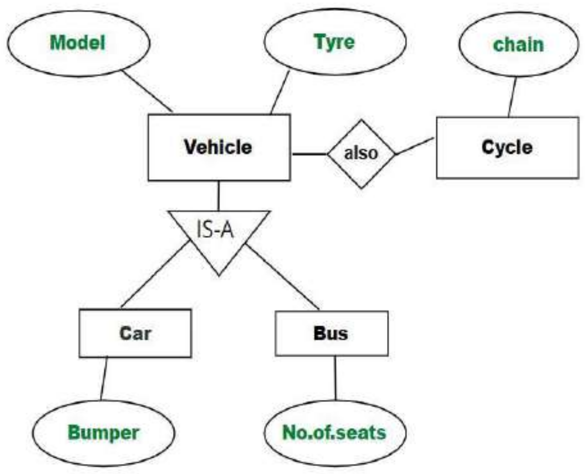
In specialization, an entity is divided into sub-entities based on its characteristics. It is a top-down approach where the higher-level entity is specialized into two or more lower-level entities. For Example, an EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER, etc. as shown in Figure 2. In this case, common attributes like E_NAME, E_SAL, etc. become part of a higher entity

(EMPLOYEE), and specialized attributes like TES_TYPE become part of a specialized entity (TESTER).
Specialization is also called as ” Top-Down approch”.



Inheritance: It is an important feature of generalization and specialization

- **Attribute inheritance** : It allows lower level entities to inherit the attributes of higher level entities and vice versa. In diagram **Car** entity is an inheritance of **Vehicle** entity ,So Car can acquire attributes of **Vehicle.** Example:car can acquire **Model** attribute of **Vehicle.**
- **Participation inheritance:** Participation inheritance in ER modeling refers to the inheritance of participation constraints from a higher-level entity (superclass) to a lower-level entity (subclass). It ensures that subclasses adhere to the same participation rules in relationships, although attributes and relationships themselves are inherited differently. In diagram Vehicle entity has an relationship with Cycle entity, but it would not automatically acquire the relationship itself with the Vehicle entity.



- Participation inheritance only refers to the inheritance of participation constraints, not the actual relationships between entities.

UNIT 2 - Relational Model

Introduction to Relational Model: -

The Relational Motel in Database Management system is a way of structuring data wing relations, which can be thought of as tables. The main construct for representing data in the relational model as a relation, A relation, consists of a relational schema and a relation instance, the relation instance as a table. It is easy to understand, and use since it is based on tables provides's mechanisms like keys and constraints to ensure data accuracy and consistency:

Example: -

Consider two relations:

•Employee Table:

Employee (Employee_ID, Name, Age, Department)

•Department Table:

Department (Department - ID, Department _ Name)

Using relational operations, you can query, join and manipulate their tables to retrieve meaningful data such as which employees belong to which department.

Concepts Of Domain: -

The concept of Domain in the relational model refers to the set of au possible values that an attribute (column) can take.It so essentially the data type or range, of allowable values for an attribute in a relation (table).

Key points about Domain:

1. Definition: A domain is a collection of atomic values. Each attribute in a relation in defined on a domain.

Ex: - An attribute like Age, the domain could be the set of all positive integers

between 1 and 100.

2. Atomicity: - A domain contains atomic (indivisible), values, meaning the values in a domain are simple and cannot be broken down further.

Ex: - The domain of an Employee ID might be a set of integers.

3. **Data Type: -**The domain typically aligns with a specific data type like:

•**Integer:** A set of whole numbers (e.g., 1, 2,3,**)**

•**Post:** A set of floating-point numbers (e.., 314, 2-5,.)

•**String:** A set of character, stings (e.., "John", "HR Department", etc.)

•**Date:** A set of date values (e.g., 2024-10-08, etc.)

•**Boolean:** A set of two possible values, typically True and false.

4. Constraints on Domain: Domains can have constraints that define specific rules for allowable values:

•**Range:** À domain can restrict values within a specific range (e.g.,Age must be between 18 and 65)

•**Pattern:** for example, the domain of an email address might enforce a specific patten, such as requiring an @ symbol.

•**Uniqueness:** In some cases, a domain may require that values be unique, as with a primary key.

Examples Of Domain in a Table:

Consider a relation

Students (Student -ID, Name, Each attribute in this student relation has its own domain:

Attribute	**Domain**
Student_ID	Positive integers (, 1 , 2,3)
Name	strings (e.g., "Alice", "John")
Age	Integers between 1 8 a n d 30
Gender	Values from the set ('Male', 'Female','Others')

•**Student_ID** is a Positive integer, so the domain consists of all possible positive integers.

•**Name** is a string, so the domain is a set of valid character strings.

•**Age** limited to Integers within a specific range, such as between 18 and 30.

•**Gender** in restricted to one of three possible values: Male, Female or other:

Age, Gender):

Concepts of attributes:

In the relational model of a database Mangement system. Attributes are the columns of a table (relation). That describes the properties or characteristics of the entities in the table.

Key concepts of Attributes:

•**Definition:** An attribute represents a data-field in a table, storing specific information about an entity.

Ex: - In a student table, the attributes, could be student_ID, Name, Age, and Course.

•**Name and Domain:** Every attribute has a name (used to identity it) and a domain, which, defines the allowable set of values for that attribute.

Ex: The attribute Age might have a domain restricting values to positive integers.

•**Data Type:** Attributes have associated data types, which define the kind of values they can hold, such as:

● **integer:** whole numbers (e.g., Age, Employee _ID)

•**String:** A set of characters (e.g., Name, Address)

● **Date:** A Valid date (e.g., Date -of-Birth)

•**Boolean:** Logical values (True or false)

Concepts Of Tuple: -

In a relational model of a Database Management System (DBMS), a tuple refers to a single so in a table, representing a single record or entry. A tuple is a collection of attributes values, each of which corresponds to a column in the table. Key Concepts of a Tuple:

•**Definition:** A tuple is a so in a table. It represents one complete instance of data in the relation (table), where each attribute (column) holds a specific value for that instance.

- **Attribute - Value Pair:** Each tuple is made up of attribute –value pairs.
- **Ex: -** If a table has three attributes (Name, Age, and Department), a tuple might contain values like ("John", 30, "HR").
- •**Uniqueness:** Each tuple in a relation in generally unique. If a table has a primary key, the values in that column must be unique, ensuing that no that tuples are Identical in terms of that key.
- **Atomic Values:** A tuple must contain atomic values, meaning each value in a tuple is indivisible.
- **Ex: -** The Age attribute in a tuple must have a single value like 30, not a list of value

•**Cardinality of Relation:** The Cardinality of a relation is the number of tuples it contains. For instance, if a table has 10 rows, its cardinality is ID.

Example of a Tuple:

Consider the following Employee table:

Employee_id	**Name**	**Age**	**Department**
1	John Doe	30	HR
2	Jane smith	28	IT
3	Alice	35	Marketing

Here, each row represents, a tuple, and the table contains three tuples. Let's break down one tuple:

•**Tuple 1:** (1, " John Doe", 30, "HR")

Employee_ID: 1

Name: John Doe

Age: 30 **Department:** HR

•**Tuple 2:** (2, "Tane Smith", 28, " IT")

Employee _ID: 2

Name: Jane Smith

Age: 28 **Department:** IT

•**Tuple 3:** (8: " Alice", 35 " Marketing")

Employee_ID: 3

Name: Alice **Age:** 35

Department: Marketing

Concepts of Relation: -

In the context of a relational database management system. A relation as a fundamental Concept that represents a table. At consists of tuples (rows) and attributes (columns), where each attribute has a specific data type. Key Concepts of Relation:

1. Definition: - A relation is a mathematical concept that sub presents a collection of tuples sharing the same attribute

2. Attributes and Tuples: -

•**Attributes: -** There are the columns of the table. Each attribute has a name and a defined data type (e.g., integer, string).

•**Tuples: -** these are the rows of the table, representing individual records. Each tuple contains values for all the attributes of the relation.

3. **Schema:** The schema of a relation defines its structure, including the names of attributes and this data types.

4. **primary Key: -** A primary key is an attribute (or a set of attributes that uniquely identifies each tuple in a relation. It ensures that no two tuples are identical based on the primary key.

5. **Foreign Key: -**A foreign key is an attribute in one relation that refers to the primary key of another relation, establishing a link between the two tables.

Example of a Relation: -

Consider a relation (table) named Employee:

Employee_ID	**Name**	**Age**	**Department**	**Salary**
1	John smith	30	HR	60000
2	John Doe	28	IT	75000
3	Alice	35	Marketing	70000
4	Rahul	40	Finance	80000

Breakdown of the Relation:

● **Attributes:**

Employee_ID: Integer, unique identifier for each employee (primary (key).

Name: String, name of the employee

Age: Integer, age of the employee

Departments: string, department where, the Employee works.

Salary: Decimal, Salary of the employee

Tuples: each row in a table is a tuple representing an employee. For example, the fast tuple (1, "John smith", 30 "HR", 60000) contains all the information for the employee with Employee_ID 1. **Importance of null values: -**

1. **Representation of Missing Data:** NULL indicates that a value as not available at the time of data entry, allowing for the correct handling of incomplete records.

Example: - an a table of employees, if an employee's phone number as not available during data entry, you can store. it as NULL instead of using an arbitrary value like "o" or “N/A”.

INSERT INTO Employees (Employee-ID, Name, Phone) VALUES

(1,''John Smith', NULL);

Here, phone is stored as NuLL to indicate missing data

2. **Representation of Unknown Data:** NULL allows for data to be stored when the value is unknown but might be provided later distinguishing between missing and unknown value.

Example: A customer signs up without providing an email address. The email field can be left as Null, indicating that the value, is unknown

INSERT INTO Customers (Customer_ID, Name, Email) VALUES (1,'Alice', NULL);

3. **Representation of Inapplicable Data:** In some cases, certain fields may not apply to all records, such as employee benefits for freelance workers. NULL can represent these inapplicable fields. **4 Data Integrity: -** NULL enforces data integrity by ensuring that databases can capture incomplete, or uncertain data without relying or misleading place holder values like "N/A" or "o".

5. **Handling of Optional Fields:** NULL is essential, when designing tables with optional fields, where certain fields are not mandatory and may not have values for every record.

6. **Facilitation of Queries:** NULL allows SQL queries to differentiate between records that have data and those that don't, enabling specific query conditions with is NULL and IS NOT NULL.

7. **Standard Approach:** NULL provides a standardized way of dealing with the absence of data across relational databases. Ensuring consistency in data management practices

<u>**Constraints: -**</u>

In relational model constraints play a critical rule in maintaining the integrity and consistency of data within relational databases.

Constraints in the relational model are rules applied to relational tables (or relations) to restrict the types of data that can be stored n than. They ensure that the data, to certain standards and rules defined by the database schema.

<u>**Types Of Constraints and their importance:**</u>

1. Domain constraint

2. Key Constraint

3. Integrity Constraint

1. <u>**Domain Constraint:**</u> - A domain constraint in the relational

model in a rule the restricts the set of permissible values that can be stored in a particular attribute (column) of a relation (table). It defines the valid values for an attribute in teams of data type, length, range, or a specific set of Values. Domain constraints ensure that data entered into the database in valid and confirms to the defined specifications.

Example: -

CREATE TABLE Customers (Customer_ID INT PRIMARY KEY, Name VARCHAR (100) NOT NULL, Age INT CHECK (Age Between 18 AND

100)- - Domain Constraint);

customer_ID: A unique identifier for each customer (Primary Key).

Name: A non-nullable column to store customer's name.

Age: This column has a check constraint that restrict the values to be between 18 and 100.

Importance of domain Constraint: -

•Domain constants ensure that only valid data types and values stored in a column.

•All entries in a column with conform to the same standards, making It easier to understand and analyze the data.

•Domain Constraints help, prevent common data entry errors at the point of data insertion. By restricting the range of acceptable values, they minimize the chances of user mistakes.

•when data is validated at the time of entry, it reduces the need for extensive data cleaning and correction later. This Simplifies database maintenance and management.

•Domain, Constraint can be used to enforce business rules, directly within the database schema.

•Domain constraints contribute the overall integrity of the database

2. <u>**Key Constraint: -**</u>

A key Constraint is a statement that a certain minimal Subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifier a tuple according to a key constraint is called a candidate Key.

•Two distinct "tuples an a legal instance (an instance that satisfies to all ICs, including the key constant) cannot have identical values in all the fields of a key.

•No subset of the set of fields in a key is a unique identifier for a tuple.

There are different types of key constraints:

1. primary key: Uniquely identifies each record in a table. A primary Key must contain unique values and cannot contain NULL Values.

2. Unique Key: Ensures that all values in a column (or a combination of Columns) are unique, but unlike a primary key is can contain NULL values.

Ex: - CREATE TABLE Students (Student_ID INT PRIMARY KEY,

-- primary key Constraint

Name VARCHAR (100) NOT NULL, Email VARCHAR (100) UNIQUE

Unique key Constraint);

Student_ID: this column is defined as the primary key. It uniquely entities each student and cannot be NULL or duplicate.

Name: A non-nullable Column to store the student's name.

Email: This column has a unique key Constraint, ensuring that no two students can have the same email address.

Importance of Key Constant:

•Primary keys encore that each record can be uniquely identified, which is essential for, data integrity.

•Key constraints prevent duplicate records, which helps maintain the accuracy of the data.

•Key constants enable efficient data retrieval and indexing.

•primary keys are often used in foreign key constants, establishing relationship between tables and maintaining referential integrity.

•Key constraints help organize data effectively, allowing for better data management and retrieval strategies.

•They enable the creation of relationship between different tables, essential for normalized database design.

3. Integrity Constant: -

Integrity Constraint in a relational model are rules that ensures the accuracy and consistency of data within a database. These constants are critical for maintaining the integrity of the database by enforcing certain rules on the data entered.

Types of integrity Constraints:

•**Entity Integrity:** Ensures that each table has a primary key, and that this key cannot contain NULL values.

•**Referential Integrity:** Ensures that a relationship between tables remain consistent. specifically, it mandates that a foreign Key must either be NULL or must match a primary key in another table.

•**Domain Integrity:** Enforces valid entries, for a given column through restrictions on the types and ranges of data.

•**User- Defined Integrity:** Custom rules defined by users to enforce specific business rules or requirements.

Example: Students Table:

CREATE TABLE Students (student-ID INT PRIMARY KEY, Name VARCHAR (100) NOT NULL);

The students table has a primary key (Student_ID) that uniquely identifies each student.

Importance of Integrity Constraint:

• Integrity constrains help ensure that the data stored in the database is accurate and conforms to specified formats and rules.

•They promote consistency across the database by ensuring that all data to defined rules.

• Integrity constants prevent the insertion of invalid or nonsensical data.

•Referential integrity constants ensures that relationships between tables are valid.

•Integrity constraints contribute to the ensures quality of the data in the database

•They allow the enforcement of business rules directly within the database schema.

Relational Algebra: -

Relational algebra is one of the two formal queries languages within the relational model. It is a procedural language. It provides a set of operations. That take one or more relation(tables)as input and provide a new relation as output.

Relational Algebra Operations: -

•**Fundamental Operations: -** Fundamental operations are the basic operations that can be performed on relational data. They are Used to retrieve, manipulate, and combine data from one on more relations(tables).

The fundamental operations of relational algebra:

1. Selection (σ):

•It is used to select required tuples of the relations.

•It retrieves rows form a relation that satisfy a specified Condition (predicate).

•It is denoted by "**σ**"

Notation: - σ condition (Relation)

Example: - σ salary <50k (Employee)

2. Project (π):

•It is used to project required column data from a relation.

•It retrieves specific columns from a relation eliminating duplicate rows

•It in denoted by "**π**"

Notation: π column 1, column 2(Relation)

Example: π Id, name, salary (sigma salary <50k (Relation))

3. Union (U):

Union operation in relational algebra is the same as union operation in set theory.

•The union operation combines the tuples of two relations, removing duplicates.

•It Is denoted by "U"

Notation: Relation 1 **U** Relation 2

Example: π ID, name (Employee 1 **U** Employee 2)

Relational Algebra: -

Relational algebra is one of the two formal queries languages within the relational model. It is a procedural language. It provides a set of operations. That take one or more relation(tables)as input and provide a new relation as output.

Relational Algebra Operations: -

•Fundamental Operations: - Fundamental operations are the basic operations that can be performed on relational data. They are Used to retrieve, manipulate, and combine data from one on more relations(tables).

The fundamental operations of relational algebra:

4. Selection (σ):

•It is used to select required tuples of the relations.

•It retrieves rows form a relation that satisfy a specified Condition (predicate).

•It is denoted by "**σ**"

Notation: - σ condition (Relation)

Example: - σ salary <50k (Employee)

Additional operations are desired from the fundamental operations. They are built using one or more fundamental operations and serve as shortcuts for common patterns of data manipulation.

The Additional operations are:

1. **Intersection (⋂):**

•The intersection operation returns the tuples that are present in both of the given relations.

•It in equivalent to selecting tuples that are common to both relations.

- It is denoted by “⋂”

Notation: Relation1 ⋂ Relation2

Example: **π**id, name (Employee 1 ⋂ Employee 2)

2. **Natural Join (H):**

•A natural join as a type of join that automatically combines tuples from two relations based on all common attributes.

•It Is denoted by “H”

Notation: Relation1 H Relation2

Example: **π**id, name (Employee1 H Employee2)

•Outer join:

•It as a type of join that returns not only the matching rows between two tables but also the non-matching row from one or both tables.

•Outer joins can be used to retrieve data even if one of the related tables does not have a matching record, felling in the missing values with NULL.

•Left outer join (⟕): **-** A left outer Join returns all the rows from the left table, along with the matching rows from the right table. If there is no match, NULL values are returned for the columns from the right table.

Ex:-

Student Table

Student_ID	**Name**
1	Alice
2	Bob
3	Carol

Enrollment Table

Enrollment_ID	**Student_id**	**Course**
101	1	math
102	2	science

Result:

Student_ID	Name	Course
1	Alice	math
2	Bob	science
3	Carol	NULL

2. **<u>Right outer join (⟖):</u> -** A right outer join returns all rows from the right table, along with the matching rows from the left table. If there is no match, NULL values are returned for the columns toom the left table.

Result:

Student_ID	Name	Course
1	Alice	math
2	Bob	science

2. **<u>Full outer join (⟗):</u> -** A full outer join returns all the rows from both tables. It there is a match between the tables, it shows the matching rows. If no match & found, NULL values are returned for the missing columns from either table.

Result:

Student_ID	Name	Course
1	Alice	math
2	Bob	science
3	Carol	NULL

<u>Relational Calculus:</u>

Relational calculus es an alternative to relational algebra. The relational calculus is a non-procedural language in relational databases. Relational calculus focuses on what results to obtain, leaving the specific execution details to the database System. In Relational calculus, two important quantifiers are used to express conditions in queries:

•Universal Quantifier (∀)

•Existential Quantifier (∃)

1. **Universal Quantifier: -** The universal quantifier states that a predicate must be true for all possible values in the specified domain.
Notation: ∀x P(x)

This means "for every value of x, the predicate P(x) is true”.

2. **Existential Quantifier: -** The existential Quantifier states that a predicate is true for at least one value in the specified domain.

Notation: ∃x P(x)

This means "there exist at least one value of x such that the
predicate P(x) is true".

There are two types of relational calculus:

•Tuple Relational Calculus (TRC)

•Domain Relational Calculus (DRC)

1. Tuple Relational Calculus (TRC): - In Tuple Relational Calculus, queries are expressed using variables that represent tuples (row) from relations (tables). A tuple relational calculus query has the from {T | P(T)}.

where, T is a tuple variable (representing rows in a relation). P(T) is a predicate (logical condition) that must be true for the tuple T.

Example: consider a student's table:

Student_ID	**Name**	**Age**
1	Alice	20
2	Bob	22
3	Carol	19

To find the name of students who are older than 20:

{T. Name | T E students ^ T Age > 20} In this query:

•T is a tuple variable representing a tuple from the student's table.

•T.name specifies that we want the Name attribute.

•The condition T.Age > 20 restricts the result to only those students whose age in greater than 20.

2. Domain Relational Calculus (ÐRC): -

In Domain Relational Calculus, queries are expressed using variables that represent individual values (or domains) rather than entire tuples. The basic structure of a DRC query is:

{x1, x2, ..., Xn | P (x1, ×2……., xn) }.

where;
•X1, x2,……,xn are domain variables representing values of attributes.

•P (x1, x2,….,xn) is a predicate (logical Condition) that must be true for the variables.

Example: -

Student_ID	**Name**	**Age**
1	Alice	20
2	Bob	22
3	Carol	19

To find the names of students coho are older than 20:

- {Name | (∋ Age (students (Name, Age) ^ Age > 20)}
- Name is a domain variable representing the name attribute
- The condition Age > 20 restricts the result to those students chose age is greater than 20.

BASIC SQL:

Simple Database Schema:

A Simple database schema refers to the structure or blueprint of a database, defining how data is organized into tables and how the relationships between these tables are established. Primary components are:

1. Tables:

•A table is a collection of related data organised in rows and columns

• Tables store entities (objects) in the database. Each table represents a different entity type (eg., students, Courses, Employees).

Example: CREATE TABLE Students (student_ID INT PRIMARY KEY, FirstName VARCHAR (50), LastName VARCHAR (50), Age INT, Department VARCHAR (100));

2. Columns (Attributes):

•Columns are the vertical division of a table that represent specific attributes of the entity.

•Each column holds a particular piece of data for each record (row) in the table.

Example: In the students table, Attributes are Student_ID, FirstName, LastName, Age and Department

3. Rows (Records):

•Rows are the horizontal entries in a table that represent Individual records or instances of the entity.

•Each row contains data for each attribute defined by the columns.

Example:

Student_ID	**Name**	**Last name**	**Age**	Department
1	Alice	John son	20	CSE
2	Bob	smith	22	AI&DS

4. Primary Key:

•A primary key is a column (or a combination of columns) that uniquely identifies each row in a table.

•It ensures that each record can be uniquely identified and prevents duplicate entries.

Example: - In the Students table, student-ID is the primary key.

5. Foreign Key:

•A foreign key is a column that creates a relationship between two tables by referencing the primary key of another table.

•It establishes a connection between related Data in different tables.

Example: - Student_ID could be a foreign key referencing students (Student_ID).

6. Unique Key: -

- Ensures that all values in a column or set of columns are unique.

•Schemas provide a structured way to organize and store data.

•It defines the relationships between tables.

Data types:

In SQL, data types define the type of data that can be stored in each column of a table. They ensure consistency and accuracy in data storage by restricting the kinds of data that can be entered.

1. Numeric Data Types:

Numeric data types are used to store numbers, they can be integers or decimals:

- **INT:** Used for storing whole numbers.

- **DECIMAL / NUMERIC:** Used for storing numbers with fixed decimal points.

2. Character/ String Data types:

string data types are used to store text.

- **CHAR(n):** Fixed length string where n defines the number of characters. If the text is shorter than n, it is padded with spaces.

- **VARCHAR(n):** Variable-length string where n defines the maximum number of characters.

- **TEXT:** Stores large blocks of text.

3. Date and Time Data Types:

Date and time data types store date, time, and date –time names.

• **DATE:** Stores date values (eg., YYYY-MM-DD).

• **TIME:** Stores time values (e.g., HH: MM: SS).

• **DATETIME:** stores both date and time values.

• **TIMESTAMP:** stores both date and time, typically used to track events.

4. Boolean Data type:

The Boolean data type stores true or false values

• **BOOLEAN:** Stores a TRUE or FALSE value. some database represents these as 1 (TRUE) and 0 (FALSE).

5. Binary Data Types:

Binary data types store binary data such as images, files or multimedia content

• **BLOB:** Binary Large object. Used for storing large amounts of binary data like images or audio.

6. Special Data Types:

These data types are used for specific purposes:

•**UP: S**tores a universally unique identifier, typically used for primary keys Instead of integer IDs.

• **ENUM:** Allows a column to have one value from a predefined list of possible values.

SQL COMMANDS

SQL (standard Query language) is, a standard language used to manage and manipulate relational databases It allow user to create, retrieve, update, and delete data stored in a database. SQL is used for tasks like querying data, inserting Records, updating data and creating database structures such as tables.

SQL commands are:

1.DDL (Data Definition Language): - DDL is a subset of SQL commands used to define and manage database structures, such as tables, schemas, and indexes. DDL commands focus on the creation, modification, and deletion of database objects.

DDL commands are:

1. **CREATE:** Used to create new database objects like tables, indexes

views, or schemas.

Syntax: - Create table table_name (colum_name1 data_type (size), Column_name 2 data_type (size));

Example: - create table student (st_id int (10), st_name varchar(in).

St_ph_no varchar (10));

→ table created

→ To check that the table is created by using desc command

→ desc student;

Name	Type
ST_ ID	NUMBER(10)
ST_ NAME	VARCHAR(10)
ST_PH_NO	VARCHAR(10)

2. **ALTER: -** It is used to alter the structure of the database.

Syntax: - To add a column.

alter table table_name and column (column_name data_type (Size); **Example: -** alter table student add column (st_address varchar (10);

→ table altered

→ To check that the table is altered or not using the Command

1 → desc student;

Name	Type
ST- ID	NUMBER (10)
ST-NAME	VARCHAR (10)
ST_PH-NO	VARCHAR (10)
ST-ADDRESS	VARCHAR (10)

→ To drop a column

Syntax: - alter table table name drop column column_name;

Example: - alter table student drop column st_name;

→ table altered

→ to check that the table is dropped or not using the command

→ desc student;

Name	Type
ST_ID	NUMBER (10)
ST_PH_NO	VARCHAR (10)
ST_ADDRESS	VARCHAR (10)

3. **TRUNCATE: -** It is used to remove, all records, from a table Including all spaces allocated for the records are removed. **Syntax: -** truncate table table_name:

Example: truncate table student;

→ It shows table Truncated

4. **DROP:** To delete objects from the database.

Syntax: - drop table table_name;

Example: - drop table student;

→ table dropped

→ to check that the table is dropped type desc student;

→ It shows object student does not exist.

5. **Rename: -** It is used to rename an object. **Syntax: -** rename old-table-name to new_table_name;

Example: - rename student to student_info;

→ table renames

→ use command Student_into

student_ info

Name	Type
ST-ID	NUMBER (10)
ST-PH-NO	VARCHAR (10)
ST-ADDRESS	VARCHAR (10)

DML [Data Manipulation Language): - DML is a subset of SQL commands used to manage and manipulate the data within database tables. DML focuses on performing operation such as inserting, updating deleting. and retrieving data.

DML commands are:

1. **INSERT: -** It is used to insert data into a table

Syntax: - insert into table_name values (value1, value2…….value n);

Example: - insert into student_info values (101, 'Sonu' '975432);

→ 1 row inserted.

→ insert into student_ into values (102, 'sita', '372146…');

→ 1 row inserted

→ insert into student info values (103, 'Mohan', '765329…');

→ 1 row inserted.

→ insert into student_info values (104, 'shyam', 850921'):

→ 1 row inserted

→ to display the table by using the command

ST_ID	ST_ NAME	ST_ PH_NO
101	Sonu	975432….
102	Sita	372146….
103	Mohan	765329….
104	Shyam	850921….

2. **UPDATE: -** Used to modify existing records in a table. **Syntax: -** Update table table_name set column_name = 'value' where condition;

Example: Update table table_name set st_name = 'Lucky' where st_id = 102;

→ 1 row updated.

→ to point the table, select * tom student_info;

ST- ID	**ST_ NAME**	**ST-PH- NO**
101	Sonu	975432...
102	Lucky	372146...
103	Mohan	765329...
104	Shyam	850921...

3. **DELETE: - Delete** all records from a table, the space for the records remain.

syntax: delete from table name where condition;

Example: delete from student_info where st_id = 101;

→ 1 row deleted

→ to print the table, select * from student _info;

ST_ID	**ST_ NAME**	**ST_ PH_NO**
102	Lucky	372146...
103	Mohan	765329...
104	Shyam	850921...

DCL (Data Control Language):

DCL is a subset of SQL Commands used to control access to data in a database. These commands are primarily focused on permissions and security.

DCL commands are: -

1. **GRANT:** Used to give users specific privileges on database Objects, such as

SELECT, INSERT, UPDATE, Or DELETE permissions
Example: Grant the ability to select and insert data to a user named John

GRANT SELECT, INSERT ON employees To John;

→This gives the user john permission to SELECT and INSERT data on the employee's table.

2. **REVOKE: -** used to remove previously granted privileges from a user or role.

Example: Revoke the INSERT permissions from the user john REVOKE INSERT ON employees FROM John;

→ This revokes the permission. for john to insert data into the

employee's table.

4) TCL (Transaction Control Language): Transaction control Language commands are used to manage transactions in a database. A transaction is a sequence of one or more SQL operations that are executed as a single unit of work.

TCL Commands are: -

1. **COMMIT:** Used to permanently save the changes made during a transaction.
2. **ROLL BACK:** used to undo changes made during a transaction. reverting the database to its previous state before the transaction started.

DRL (Data Retrieval Language):

DRL also known as DQL (data Query language) is a subset of SQL commands used to retrieve data from a database. The primary command is the SELECT Statement

DRL Command:

SELECT: It is used, to query and retrieve data from a database. It can retrieve data from one or more tables based on Specific conditions.

Syntax: select* From table_ name;

Example: Select * From student _ info;

ST- ID	**ST-NAME**	**ST-PH-NO**
101	Lucky	372146...
103	Mohan	765329…
104	Shyam	850921...

UNIT 3 - SQL

The SQL **WHERE clause** allows to filtering of records in queries. Whether you're retrieving data, **updating records**, or deleting entries from a database, the WHERE clause plays an important role in defining which rows will be affected by the query. Without it, **SQL queries** would return all rows in a table, making it difficult to target specific data.

In this article, we will learn the **WHERE clause** in detail—from basic concepts to advanced ones. We'll cover practical examples, discuss common operators, provide optimization tips, and address real-world use cases.

What is the SQL WHERE Clause?
The SQL WHERE clause is used to specify a condition while **fetching** or modifying data in a database. It **filters the rows** that are affected by the **SELECT**, UPDATE, DELETE, or **INSERT** operations. The condition can range from simple comparisons to complex expressions, enabling precise targeting of the data.

Syntax:
*SELECT **column1,column2** FROM table_name WHERE column_name operator value;*

Parameter Explanation:

column1,column2: fields in the table

table_name: name of table

column_name: name of field used for filtering the data

operator: operation to be considered for filtering

value: exact value or pattern to get related data in the result

Examples of WHERE Clause in SQL

To create a basic employee table structure in SQL for performing all the where clause operation.

Query:

```
CREATE TABLE Emp1(EmpID INT PRIMARY KEY, Name VARCHAR(50), Country
VARCHAR(50), Age int(2), mob int(10));


-- Insert some sample data into the Customers table
INSERT INTO Emp1 (EmpID, Name,Country, Age, mob)
VALUES (1, 'Shubham',  'India','23','738479734'),
      (2, 'Aman ',  'Australia','21','436789555'),
      (3, 'Naveen', 'Sri lanka','24','34873847'),
      (4, 'Aditya',  'Austria','21','328440934'),
      (5, 'Nishant', 'Spain','22','73248679'));
```

Output:

EmpID	Name	Country	Age	mob
1	Shubham	India	23	738479734
2	Aman	Australia	21	436789555
3	Naveen	Sri lanka	24	34873847
4	Aditya	Austria	21	328440934
5	Nishant	Spain	22	73248679

SQL | Arithmetic Operators

Arithmetic Operators are:

+	[Addition]
-	[Subtraction]
/	[Division]
*	[Multiplication]
%	[Modulus]

Addition (+):-

It is used to perform **addition operation** on the data items, items include either single column or multiple columns.

Implementation:

```
SELECT employee_id, employee_name, salary, salary + 100
  AS "salary + 100" FROM addition;
```

Output:

employee_id	**employee_name**	**salary**	**salary+100**
1	alex	25000	25100

2	rr	55000	55100
3	jpm	52000	52100
4	ggshmr	12312	12412

Here we have done addition of 100 to each Employee’s salary i.e, addition operation on single column.

Let’s perform **addition of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary + employee_id
  AS "salary + employee_id" FROM addition;
```

Output:

employee_id	employee_name	salary	salary+employee_id
1	alex	25000	25001
2	rr	55000	55002
3	jpm	52000	52003
4	ggshmr	12312	12316

Here we have done addition of 2 columns with each other i.e, each employee’s employee_id is added with its salary.

<u>Subtraction (-) :</u>

It is use to perform **subtraction operation** on the data items, items include either single column or multiple columns.

Implementation:

```
SELECT employee_id, employee_name, salary, salary - 100
   AS "salary - 100" FROM subtraction;
```

Output:

employee_id	employee_name	salary	salary-100

12	Finch	15000	14900
22	Peter	25000	24900
32	Warner	5600	5500
42	Watson	90000	89900

Here we have done subtraction of 100 to each Employee's salary i.e, subtraction operation on single column.

Let's perform **subtraction of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary - employee_id
   AS "salary - employee_id" FROM subtraction;
```

Output:

employee_id	employee_name	salary	salary – employee_id
12	Finch	15000	14988
22	Peter	25000	24978
32	Warner	5600	5568
42	Watson	90000	89958

Here we have done subtraction of 2 columns with each other i.e, each employee's employee_id is subtracted from its salary.

Division (/) : For **Division** refer this link- Division in SQL

Multiplication (*) :

It is use to perform **multiplication** of data items.

Implementation:

```
SELECT employee_id, employee_name, salary, salary * 100
   AS "salary * 100" FROM addition;
```

Output:

employee_id	**employee_name**	**salary**	**salary * 100**
1	Finch	25000	2500000
2	Peter	55000	5500000
3	Warner	52000	5200000
4	Watson	12312	1231200

Here we have done multiplication of 100 to each Employee's salary i.e, multiplication operation on single column.

Let's perform **multiplication of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary * employee_id
    AS "salary * employee_id" FROM addition;
```

Output:

employee_id	**employee_name**	**salary**	**salary * employee_id**
1	Finch	25000	25000
2	Peter	55000	110000
3	Warner	52000	156000
4	Watson	12312	49248

Here we have done multiplication of 2 columns with each other i.e, each employee's employee_id is multiplied with its salary.

Modulus (%) :

It is use to get **remainder** when one data is divided by another.

Implementation:

```
SELECT employee_id, employee_name, salary, salary % 25000
    AS "salary % 25000" FROM addition;
```

Output:

employee_id	**employee_name**	**salary**	**salary % 25000**
1	Finch	25000	0
2	Peter	55000	5000
3	Warner	52000	2000
4	Watson	12312	12312

Here we have done modulus of 100 to each Employee's salary i.e, modulus operation on single column.

Let's perform **modulus operation between 2 columns**:

```
SELECT employee_id, employee_name, salary, salary % employee_id
    AS "salary % employee_id" FROM addition;
```

Output:

employee_id	**employee_name**	**salary**	**salary % employee_id**
1	Finch	25000	0
2	Peter	55000	0
3	Warner	52000	1
4	Watson	12312	0

Here we have done modulus of 2 columns with each other i.e, each employee's salary is divided with its id and corresponding remainder is shown.

Basically, **modulus** is use to check whether a number is **Even** or **Odd**. Suppose a given number if divided by 2 and gives 1 as remainder, then it is an *odd number* or if on dividing by 2 and gives 0 as remainder, then it is an *even number*.

Concept of NULL :

If we perform any arithmetic operation on **NULL**, then answer is *always* null.

Implementation:

```
SELECT employee_id, employee_name, salary, type, type + 100
   AS "type+100" FROM addition;
```

Output:

employee_id	employee_name	salary	type	type + 100
1	Finch	25000	NULL	NULL
2	Peter	55000	NULL	NULL
3	Warner	52000	NULL	NULL
4	Watson	12312	NULL	NULL

Here output always came null, since performing any operation on null will always result in a *null value*.

SQL – Logical Operators

SQL logical operators are used to test for the truth of the condition. A logical operator like the Comparison operator returns a boolean value of **TRUE**, **FALSE**, or **UNKNOWN.** In this article, we will discuss different types of Logical Operators.

Logical operators are used to combine or manipulate the conditions given in a query to retrieve or manipulate data .there are some logical operators in SQL like OR, AND etc.

Types of Logical Operators in SQL

Given below is the list of logical operators available in SQL.

Operator	**Meaning**
AND	TRUE if both Boolean expressions are TRUE.
IN	TRUE if the operand is equal to one of a list of expressions.
NOT	Reverses the value of any other Boolean operator.
OR	TRUE if either Boolean expression is TRUE.
LIKE	TRUE if the operand matches a pattern.
BETWEEN	TRUE if the operand is within a range.
ALL	TRUE if all of a set of comparisons are TRUE.
ANY	TRUE if any one of a set of comparisons is TRUE.
EXISTS	TRUE if a subquery contains any rows.
SOME	TRUE if some of a set of comparisons are TRUE.

Example:

In the below example, we will see how this logical operator works with the help of creating a database.

Step 1: Creating a Database

In order to create a database, we need to use the CREATE operator.

Query

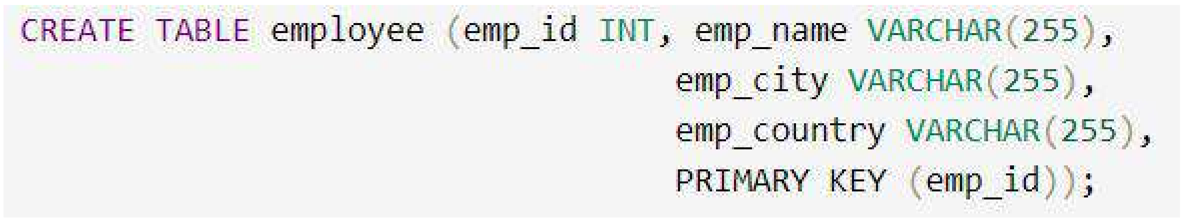
```
CREATE DATABASE xstream_db;
```

Step 2: Create table employee

In this step, we will create the table employee inside the x stream_db database.

Query

```
CREATE TABLE employee (emp_id INT, emp_name VARCHAR(255),
                       emp_city VARCHAR(255),
                       emp_country VARCHAR(255),
                       PRIMARY KEY (emp_id));
```



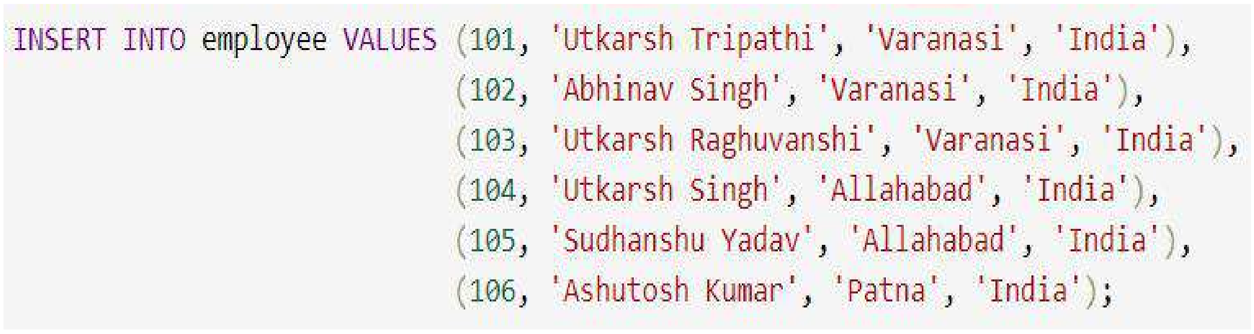
```
CREATE TABLE employee (emp_id INT, emp_name VARCHAR(255),
                                   emp_city VARCHAR(255),
                                   emp_country VARCHAR(255),
                                   PRIMARY KEY (emp_id));
```

Create Table

In order to insert the data inside the database, we need to use the INSERT operator.

Query

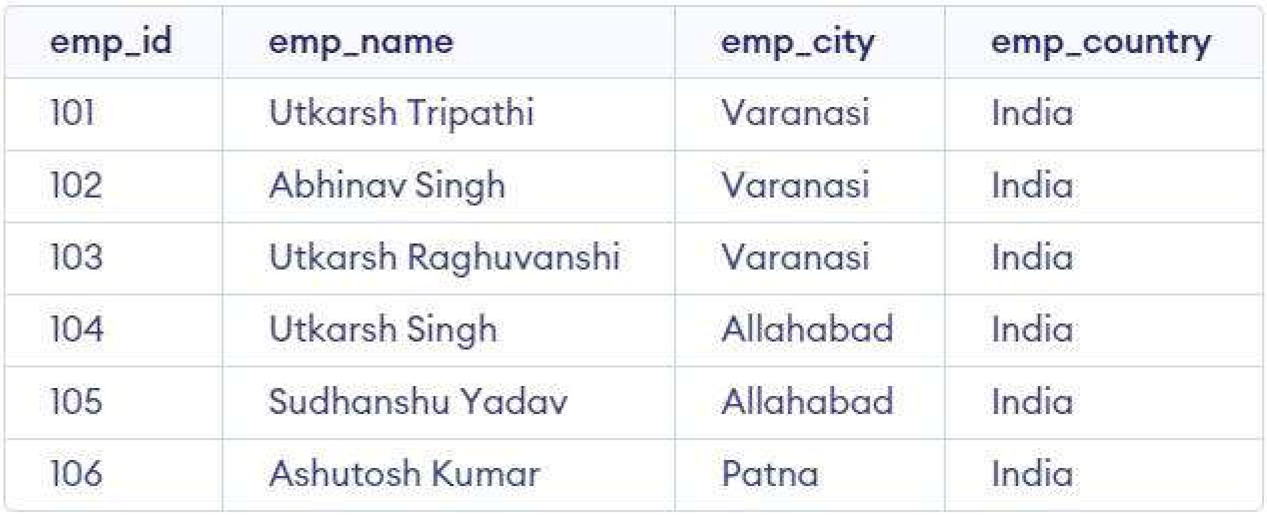
```
INSERT INTO employee VALUES (101, 'Utkarsh Tripathi', 'Varanasi', 'India'),
                  (102, 'Abhinav Singh', 'Varanasi', 'India'),
                  (103, 'Utkarsh Raghuvanshi', 'Varanasi', 'India'),
                  (104, 'Utkarsh Singh', 'Allahabad', 'India'),
                  (105, 'Sudhanshu Yadav', 'Allahabad', 'India'),
                  (106, 'Ashutosh Kumar', 'Patna', 'India');
```



```
INSERT INTO employee VALUES (101, 'Utkarsh Tripathi', 'Varanasi', 'India'),
                            (102, 'Abhinav Singh', 'Varanasi', 'India'),
                            (103, 'Utkarsh Raghuvanshi', 'Varanasi', 'India'),
                            (104, 'Utkarsh Singh', 'Allahabad', 'India'),
                            (105, 'Sudhanshu Yadav', 'Allahabad', 'India'),
                            (106, 'Ashutosh Kumar', 'Patna', 'India');
```

Insert Value

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India
104	Utkarsh Singh	Allahabad	India
105	Sudhanshu Yadav	Allahabad	India
106	Ashutosh Kumar	Patna	India

employee Table

Now the given below is the list of different logical operators.

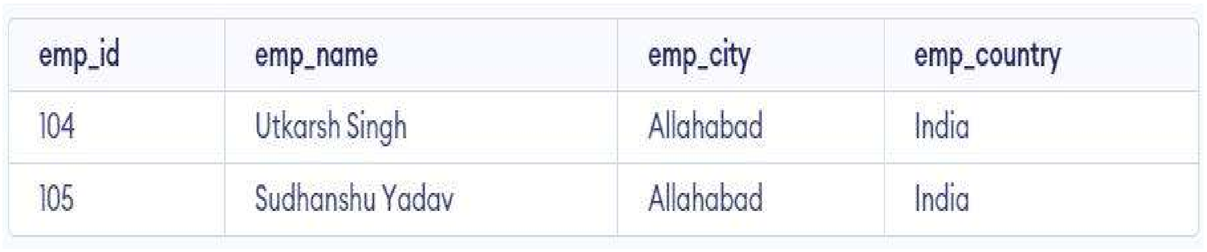
AND Operator

The AND operator is used to combines two or more conditions but if it is true when all the conditions are satisfied.

Query

SELECT * FROM employee WHERE emp_city = 'Allahabad' AND emp_country = 'India';

Output



emp_id	emp_name	emp_city	emp_country
104	Utkarsh Singh	Allahabad	India
105	Sudhanshu Yadav	Allahabad	India

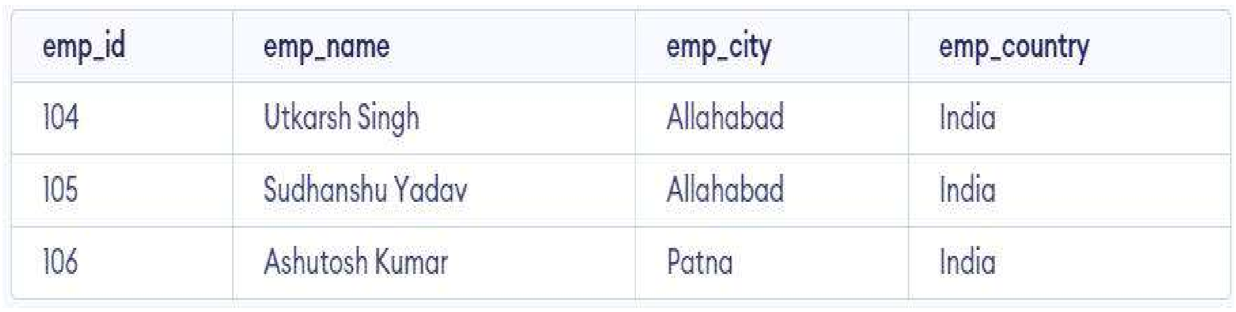
IN Operator

It is used to remove the multiple OR conditions in SELECT, INSERT, UPDATE, or DELETE. and We can also use NOT IN to minimize the rows in your list and any kind of duplicate entry will be retained.

Query

SELECT * FROM employee WHERE emp_city IN ('Allahabad', 'Patna');

Output



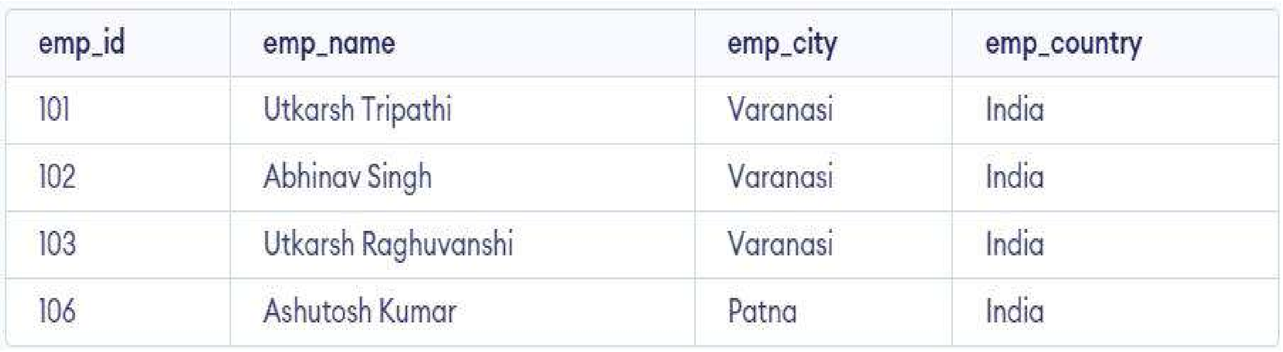
emp_id	emp_name	emp_city	emp_country
104	Utkarsh Singh	Allahabad	India
105	Sudhanshu Yadav	Allahabad	India
106	Ashutosh Kumar	Patna	India

NOT Operator

Query

SELECT * FROM employee WHERE emp_city NOT LIKE 'A%';

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India
106	Ashutosh Kumar	Patna	India

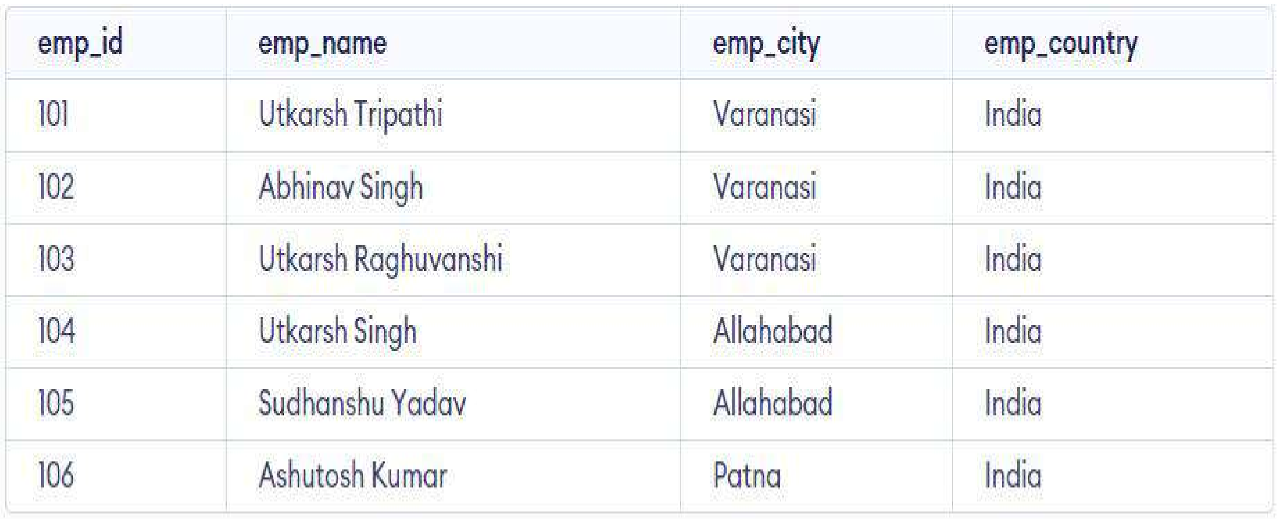
OR Operator

The OR operator is used to combines two or more conditions but if it is true when one of the conditions are satisfied.

Query

SELECT * FROM employee WHERE emp_city = 'Varanasi' OR emp_country = 'India';

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India
104	Utkarsh Singh	Allahabad	India
105	Sudhanshu Yadav	Allahabad	India
106	Ashutosh Kumar	Patna	India

LIKE Operator

In SQL, the LIKE operator is used in the WHERE clause to search for a specified pattern in a column.

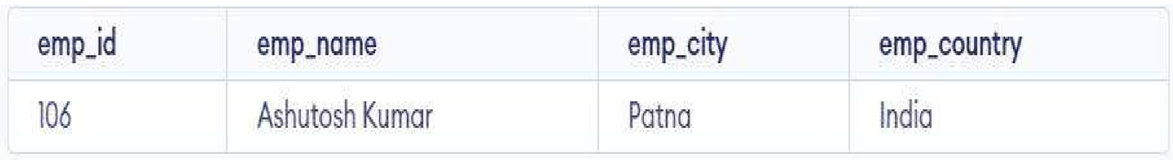
% – It is used for zero or more than one character.

_ – It is used for only one character means fixed length.

Query

SELECT * FROM employee WHERE emp_city LIKE 'P%';

Output



emp_id	emp_name	emp_city	emp_country
106	Ashutosh Kumar	Patna	India

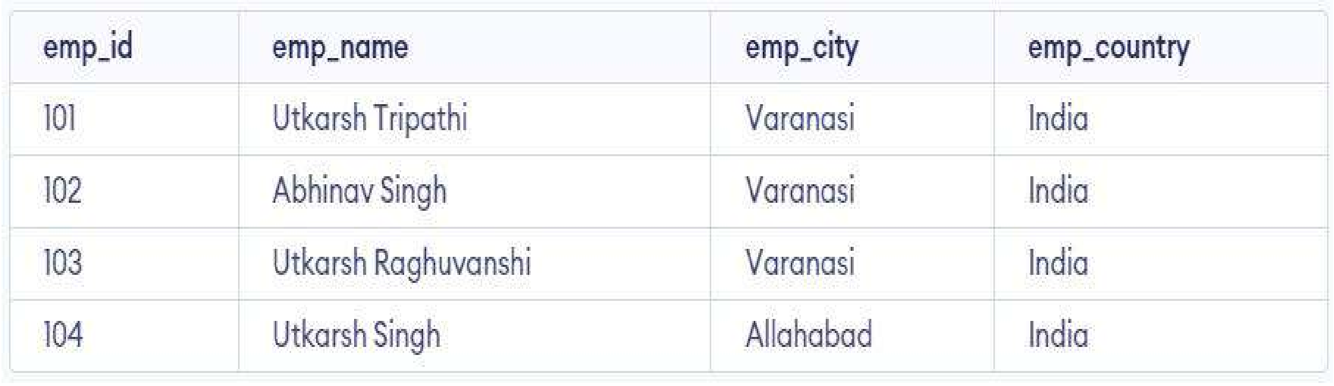
<u>BETWEEN Operator</u>

The SQL **BETWEEN** condition allows you to easily test if an expression is within a range of values (inclusive).

Query

```
SELECT * FROM employee WHERE emp_id BETWEEN 101 AND 104;
```

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India
104	Utkarsh Singh	Allahabad	India

<u>ALL Operator</u>

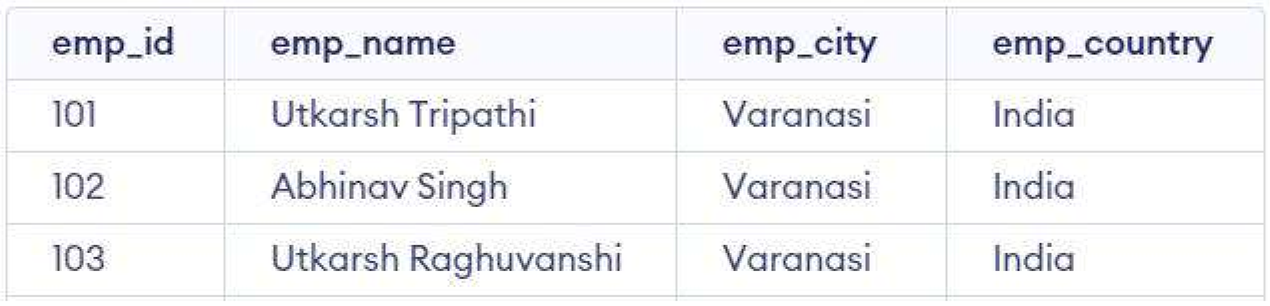
The ALL operator returns TRUE if all of the subqueries values matches the condition.

All operator is used with SELECT, WHERE, HAVING statement.

Query

```
SELECT * FROM employee WHERE emp_id = ALL
            (SELECT emp_id FROM employee WHERE emp_city = 'Varanasi');
```

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India

<u>ANY Operator</u>

The ANY operator:

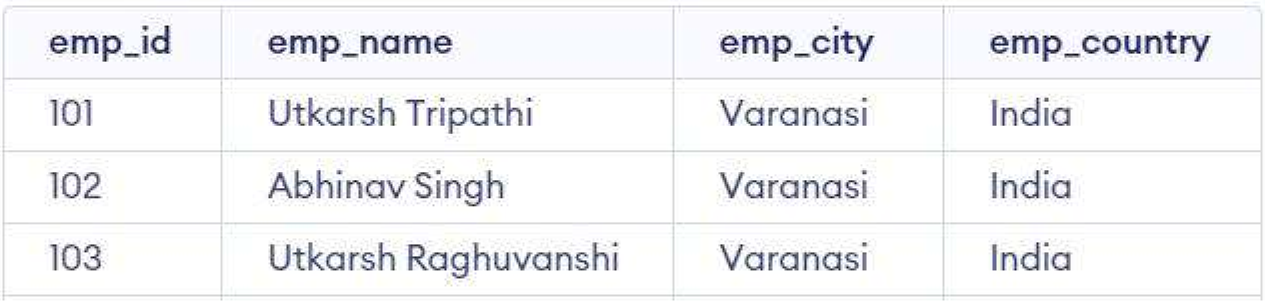
It returns a boolean value as a result

It returns TRUE if ANY of the subquery values match the condition

Query

```
SELECT * FROM employee WHERE emp_id = ANY
            (SELECT emp_id FROM employee WHERE emp_city = 'Varanasi');
```

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India

<u>EXISTS Operator</u>

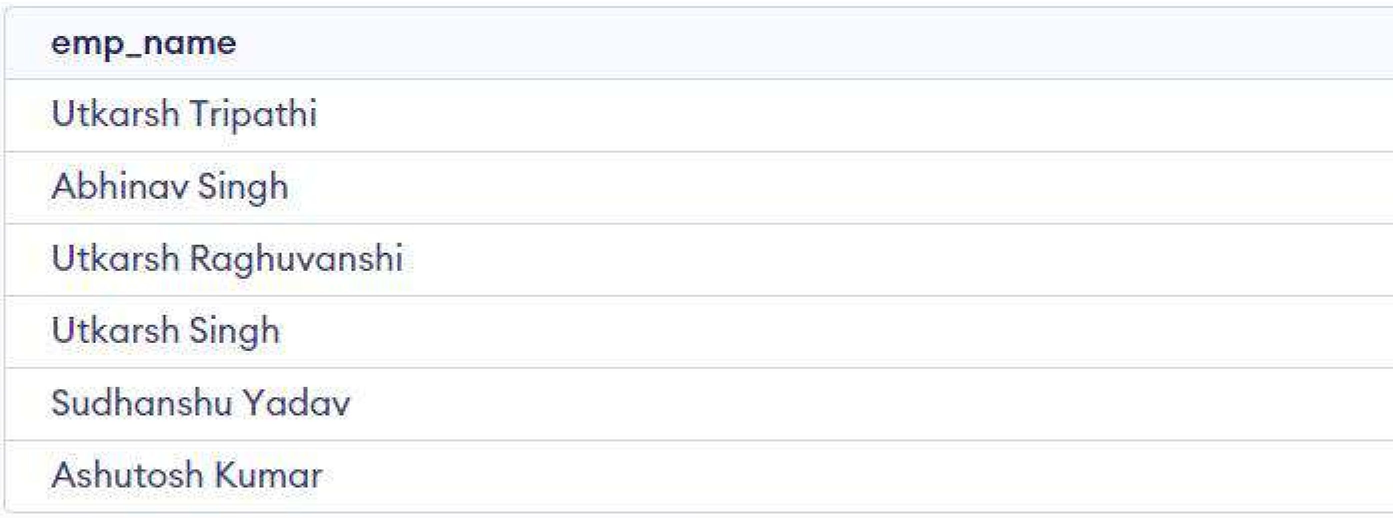
In SQL,Exists operator is used to check whether the result of a correlated nested query is empty or not.

Exists operator is used with SELECT, UPDATE, INSERT or DELETE statement.

Query

```
SELECT emp_name FROM employee WHERE EXISTS
            (SELECT emp_id FROM employee WHERE emp_city = 'Patna');
```

Output



emp_name
Utkarsh Tripathi
Abhinav Singh
Utkarsh Raghuvanshi
Utkarsh Singh
Sudhanshu Yadav
Ashutosh Kumar

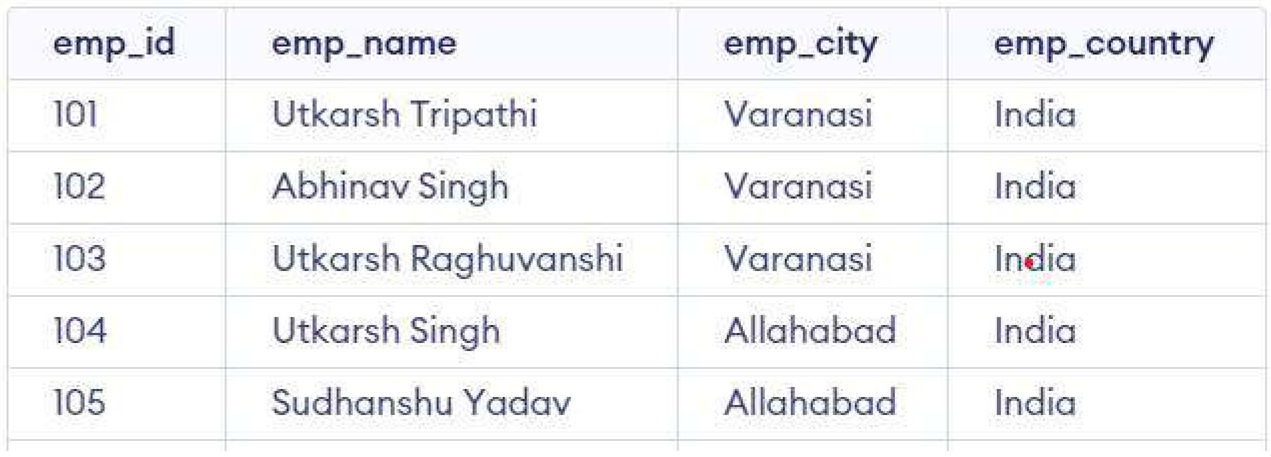
SOME Operator

In SQL, SOME operators are issued with comparison operators (<,>,=,<=, etc) to compare the value with the result of a subquery.

Query

```
SELECT * FROM employee WHERE emp_id < SOME
            (SELECT emp_id FROM employee WHERE emp_city = 'Patna');
```

Output



emp_id	emp_name	emp_city	emp_country
101	Utkarsh Tripathi	Varanasi	India
102	Abhinav Singh	Varanasi	India
103	Utkarsh Raghuvanshi	Varanasi	India
104	Utkarsh Singh	Allahabad	India
105	Sudhanshu Yadav	Allahabad	India

Functions in MySQL | Numeric, String and Date Time Functions in MySQL

In **MySQL**, functions play a crucial role in performing various operations on data, such as **calculations**, **string manipulations**, and **date handling.** These built-in functions simplify complex queries and data transformations, making it easier to manage and analyze data within a database.

In this article, we will look at these different categories of MySQL functions, and look at different MySQL functions with definitions and examples in each category.

Functions in MySQL

In **MySQL**, functions are a fundamental part of the **SQL** language, enabling us to perform calculations, manipulate data and retrieve information.

The **functions in MySQL** can edit rows and tables, alter strings, and help us to manage organized and easy-to-navigate **databases**.

A function is a special type of predefined command set that performs some operation and returns a value. Functions operate on zero, one, two, or more values that are provided to them.

The values that are provided to functions are called parameters or arguments.

The MySQL functions have been categorized into various categories, such as **String functions, Mathematical functions, Date and Time functions,** etc.

String Functions

Numeric Functions

Date and Time Functions

String functions

String functions are used to perform an operation on input string and return an output string. Following are the string functions defined in SQL:

1. **CONCAT**

Purpose: Combines two or more strings into a single string.

Syntax:

```
CONCAT(string1, string2, ...)
```

Example:

```
SELECT CONCAT('Hello', ' ', 'World') AS Result;
```

Result:

```
Result -------- Hello World
```

2. **LENGTH / LEN**

Purpose: Returns the length of the string.

Syntax:

```
LENGTH(string)
```

Example:

```
SELECT LENGTH('Database') AS Length;
```

Result:

```
Length -------- 8
```

3. **UPPER**

Purpose: Converts all characters in a string to uppercase.

Syntax:

```
UPPER(string)
```

Example:

```
SELECT UPPER('hello') AS UpperCase;
```

Result:

```
UpperCase -------- HELLO
```

4. **LOWER**

Purpose: Converts all characters in a string to lowercase.

Syntax:

```
LOWER(string)
```

Example:

```
SELECT LOWER('WORLD') AS LowerCase;
```

Result:

```
LowerCase -------- world
```

5. **SUBSTRING / SUBSTR**

Purpose: Extracts a part of the string.

Syntax:

```
SUBSTRING(string, start_position, length)
```

Example:

```
SELECT SUBSTRING('Database', 1, 4) AS Sub;
```

Result:

```
Sub -------- Data
```

6. **TRIM**

Purpose: Removes leading, trailing, or both spaces (or characters) from a string.

Syntax:

```
TRIM([BOTH | LEADING | TRAILING] 'char' FROM string)
```

Example:

```
SELECT TRIM(' SQL ') AS Trimmed;
```

Result:

```
Trimmed -------- SQL
```

7. **REPLACE**

Purpose: Replaces all occurrences of a substring with another substring.

Syntax:

```
REPLACE(string, old_substring, new_substring)
```

Example:

```
SELECT REPLACE('Learn SQL', 'SQL', 'DBMS') AS Replaced;
```

Result:

```
Replaced -------- Learn DBMS
```

8. **REVERSE**

Purpose: Reverses the characters of a string.

Syntax:

```
REVERSE(string)
```

Example:

```
SELECT REVERSE('DBMS') AS Reversed;
```

Result:

```
Reversed -------- SMBD
```

9. **INSTR / CHARINDEX**

Purpose: Finds the position of a substring in a string.

Syntax:

For MySQL: `INSTR(string, substring)`

For SQL Server: `CHARINDEX(substring, string)`

Example:

```
SELECT INSTR('Database', 'base') AS Position;
```

Result:

```
Position -------- 5
```

10. **LEFT**

Purpose: Returns the specified number of characters from the **left** of the string.

Syntax:

```
LEFT(string, number_of_characters)
```

Example:

```
SELECT LEFT('Database', 4) AS LeftPart;
```

Result:

```
LeftPart -------- Data
```

11. **RIGHT**

Purpose: Returns the specified number of characters from the **right** of the string.

Syntax:

```
RIGHT(string, number_of_characters)
```

Example:

```
SELECT RIGHT('Database', 4) AS RightPart;
```

Result:

```
RightPart -------- base
```

12. **LPAD**

Purpose: Pads a string on the left side with a specified character.

Syntax:

```
LPAD(string, length, pad_string)
```

Example:

```
SELECT LPAD('SQL', 5, '*') AS LeftPad;
```

Result:

LeftPad -------- **SQL

13. **RPAD**

Purpose: Pads a string on the right side with a specified character.

Syntax:

```
RPAD(string, length, pad_string)
```

Example:

```
SELECT RPAD('SQL', 5, '*') AS RightPad;
```

Result:

```
RightPad -------- SQL**
```

14. **CONCAT_WS**

Purpose: Combines strings with a specified separator.

Syntax:

```
CONCAT_WS(separator, string1, string2, ...)
```

Example:

```
SELECT CONCAT_WS('-', '2024', '06', '23') AS Date;
```

Result:

Date -------- 2024-06-23

15. **ASCII**

Purpose: Returns the ASCII value of the first character of a string.

Syntax:

ASCII(string)

Example:

SELECT ASCII('A') AS AsciiValue;

Result:

AsciiValue -------- 65

16. **CHAR**

Purpose: Converts an ASCII code to its corresponding character.

Syntax:

CHAR(ascii_value)

Example:

SELECT CHAR(65) AS Character;

Result:

Character -------- A

Summary Table of String Functions:

Function	**Purpose**	**Example**	**Result**
CONCAT	Combine strings	CONCAT('A', 'B')	AB
LENGTH	Find string length	LENGTH('SQL')	3
UPPER	Uppercase conversion	UPPER('sql')	SQL
LOWER	Lowercase conversion	LOWER('SQL')	sql
SUBSTRING	Extract part of string	SUBSTRING('Hello', 2, 3)	ell
REPLACE	Replace substring	REPLACE('SQL', 'S', 'P')	PQL
REVERSE	Reverse string	REVERSE('ABC')	CBA
TRIM	Remove spaces	TRIM(' SQL ')	SQL

Numeric Functions

Numeric Functions are used to perform operations on numbers and return numbers. Following are the numeric functions defined in SQL:

The table Products looks like this:

ProductID	**ProductName**	**Quantity**	**Price**	**Discount**
1	Laptop	10	50000.75	5.5
2	Phone	25	20000.50	10.0
3	Tablet	15	15000.25	7.5
4	Headphones	50	2000.00	15.0
5	Monitor	8	12000.00	0.0

1. **ABS()**

Purpose: Returns the absolute value of a number.

Syntax:

```
SELECT ABS(column_name) AS Result FROM table_name;
```

Query:

```
SELECT ProductName, ABS(Discount) AS AbsoluteDiscount FROM Products;
```

Result:

ProductName	**AbsoluteDiscount**
Laptop	5.5
Phone	10.0
Tablet	7.5
Headphones	15.0
Monitor	0.0

---

2. **CEIL()**

Purpose: Returns the smallest integer greater than or equal to a number.

Syntax:

SELECT CEIL(column_name) AS Result FROM table_name;

Query:

SELECT ProductName, CEIL(Price) AS CeilPrice FROM Products;

Result:

ProductName	**CeilPrice**
Laptop	50001
Phone	20001
Tablet	15001
Headphones	2000
Monitor	12000

---

3. **FLOOR()**

Purpose: Returns the largest integer less than or equal to a number.

Syntax:

SELECT FLOOR(column_name) AS Result FROM table_name;

Query:

SELECT ProductName, FLOOR(Price) AS FloorPrice FROM Products;

Result:

ProductName	**FloorPrice**
Laptop	50000
Phone	20000
Tablet	15000
Headphones	2000
Monitor	12000

---

4. **ROUND()**

Purpose: Rounds a number to the specified number of decimal places.

Syntax:

```
SELECT ROUND(column_name, decimal_places) AS Result FROM table_name;
```

Query:

sql

Copy code

```
SELECT ProductName, ROUND(Price, 1) AS RoundedPrice FROM Products;
```

Result:

ProductName	**RoundedPrice**
Laptop	50000.8
Phone	20000.5
Tablet	15000.3
Headphones	2000.0
Monitor	12000.0

---

5. **MOD()**

Purpose: Returns the remainder of a division.

Syntax:

```
SELECT MOD(column1, column2) AS Result FROM table_name;
```

Query:

```
SELECT ProductName, MOD(Quantity, 3) AS QuantityRemainder FROM Products;
```

Result:

ProductName	**QuantityRemainder**
Laptop	1
Phone	1
Tablet	0

ProductName	**QuantityRemainder**
Headphones	2
Monitor	2

---

6. **POWER()**

Purpose: Returns the value of a number raised to a power.

Syntax:

SELECT POWER(column_name, exponent) AS Result FROM table_name;

Query:

SELECT ProductName, POWER(2, 3) AS PowerResult FROM Products;

Result:

ProductName	**PowerResult**
Laptop	8
Phone	8
Tablet	8
Headphones	8
Monitor	8

---

7. **SQRT()**

Purpose: Returns the square root of a number.

Syntax:

SELECT SQRT(column_name) AS Result FROM table_name;

Query:

SELECT ProductName, SQRT(Price) AS PriceSquareRoot FROM Products;

Result:

ProductName	**PriceSquareRoot**
Laptop	223.61

ProductName	**PriceSquareRoot**
Phone	141.42
Tablet	122.47
Headphones	44.72
Monitor	109.54

Summary of Queries and Results

Function	**Purpose**	**Example Query**	**Key Result**
ABS()	Absolute value	ABS(Discount)	Returns positive value
CEIL()	Smallest integer ≥ number	CEIL(Price)	50001 for 50000.75
FLOOR()	Largest integer ≤ number	FLOOR(Price)	50000 for 50000.75
ROUND()	Round to specified decimals	ROUND(Price, 1)	50000.8
MOD()	Remainder of division	MOD(Quantity, 3)	1 or 2
POWER()	Exponentiation	POWER(2, 3)	8
SQRT()	Square root	SQRT(Price)	√50000.75 = 223.61

SQL Date and Time Functions

In SQL, dates are complicated for newbies, since while working with a database, the format of the data in the table must be matched with the input data to insert. In various scenarios instead of date, date-time (time is also involved with date) is used.

For storing a date or a date and time value in a database,**MySQL** offers the following data types:

DATE	format YYYY-MM-DD
DATETIME	format: YYYY-MM-DD HH:MI: SS
TIMESTAMP	format: YYYY-MM-DD HH:MI: SS

DATE	format YYYY-MM-DD
YEAR	format YYYY or YY

Now, come to some popular functions in SQL date functions.

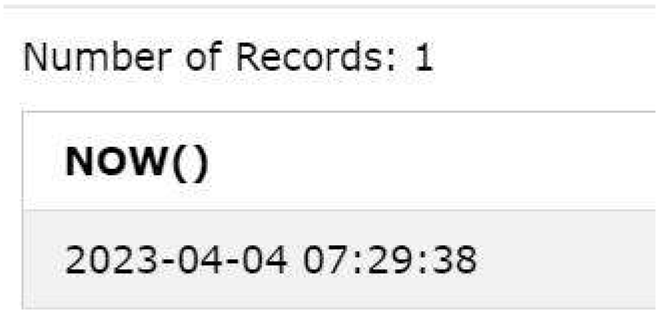
NOW()

Returns the current date and time.

Query:

SELECT NOW();

Output:



```
Number of Records: 1

NOW()
2023-04-04 07:29:38
```

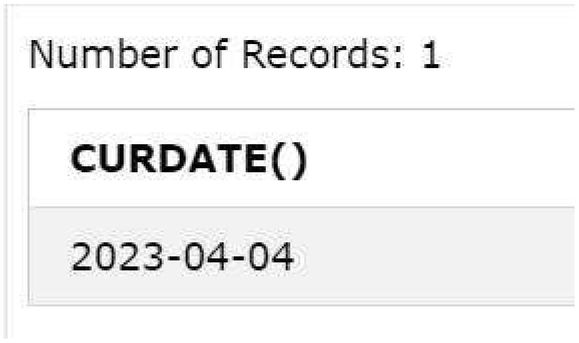
CURDATE()

Returns the current date.

Query:

SELECT CURDATE();

Output:



```
Number of Records: 1

CURDATE()
2023-04-04
```

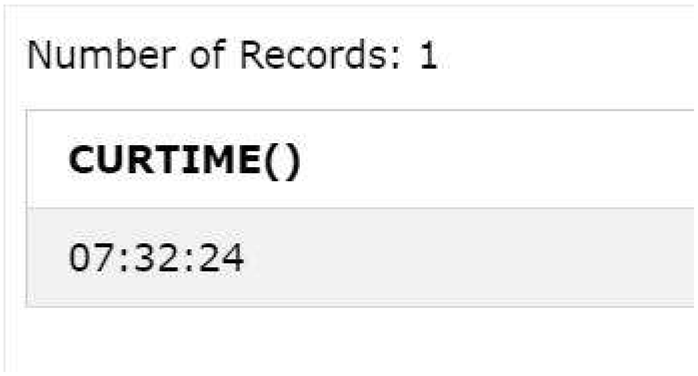
CURTIME()

Returns the current time.

Query:

SELECT CURTIME();

Output:



```
Number of Records: 1

CURTIME()
07:32:24
```

DATE()

Extracts the date part of a date or date/time expression. Example: For the below table named 'Test'

Id	Name	BirthTime
4120	Pratik	1996-09-26 16:44:15.581

Query:

SELECT Name, DATE(BirthTime)
AS BirthDate FROM Test;

Output:

Name	BirthDate
Pratik	1996-09-26

EXTRACT()

Returns a single part of a date/time.

Syntax

EXTRACT(unit FROM date);

Several units can be considered but only some are used such as **MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.** And 'date' is a valid date expression. Example: For the below table named 'Test'

Id	Name	BirthTime

Id	**Name**	**BirthTime**
4120	Pratik	1996-09-26 16:44:15.581

Query:

SELECT Name, Extract(DAY FROM BirthTime) AS BirthDay FROM Test;

Output:

Name	**Birthday**
Pratik	26

Query:

SELECT Name, Extract(YEAR FROM BirthTime) AS BirthYear FROM Test;

Output:

Name	**BirthYear**
Pratik	1996

Query:

SELECT Name, Extract(SECOND FROM BirthTime) AS BirthSecond FROM Test;

Output:

Name	**BirthSecond**
Pratik	581

DATE_ADD()

Adds a specified time interval to a date.

Syntax:

DATE_ADD(date, INTERVAL expr type);

Where, date – valid date expression, and expr is the number of intervals we want to add. and type can be one of the following: MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc. Example: For the below table named ‘Test’

Id	**Name**	**BirthTime**
4120	Pratik	1996-09-26 16:44:15.581

Query:

SELECT Name, DATE_ADD(BirthTime, INTERVAL
1 YEAR) AS BirthTimeModified FROM Test;

Output:

Name	**BirthTimeModified**
Pratik	1997-09-26 16:44:15.581

Query:

SELECT Name, DATE_ADD(BirthTime, INTERVAL 30 DAY) AS BirthDayModified FROM Test;

Output:

Name	**BirthDayModified**
Pratik	1996-10-26 16:44:15.581

Query:

SELECT Name, DATE_ADD(BirthTime, INTERVAL
4 HOUR) AS BirthHourModified FROM Test;

Output:

Name	**BirthSecond**
Pratik	1996-10-26 20:44:15.581

DATE_SUB()

Subtracts a specified time interval from a date. The syntax for DATE_SUB is the same as DATE_ADD just the difference is that DATE_SUB is used to subtract a given interval of date.

DATEDIFF()

Returns the number of days between two dates.

Syntax:

DATEDIFF(interval,date1, date2);

interval – minute/hour/month/year,etc

date1 & date2- date/time expression

Query:

SELECT DATEDIFF(day, '2017-01-13', '2017-01-03') AS DateDiff;

Output:

DateDiff
0

DATE_FORMAT()

Displays date/time data in different formats.

Syntax:

DATE_FORMAT(date,format);

the date is a valid date and the format specifies the output format for the date/time. The formats that can be used are:

%a-Abbreviated weekday name (Sun-Sat)

%b-Abbreviated month name (Jan-Dec)

%c-Month, numeric (0-12)

%D-Day of month with English suffix (0th, 1st, 2nd, 3rd)

%d-Day of the month, numeric (00-31)

%e-Day of the month, numeric (0-31)

%f-Microseconds (000000-999999)

%H-Hour (00-23)

%h-Hour (01-12)

%I-Hour (01-12)

%i-Minutes, numeric (00-59)

%j-Day of the year (001-366)

%k-Hour (0-23)

%l-Hour (1-12)

%M-Month name (January-December)

%m-Month, numeric (00-12)

%p-AM or PM

%r-Time, 12-hour (hh:mm: ss followed by AM or PM)

%S-Seconds (00-59)

%s-Seconds (00-59)

%T-Time, 24-hour (hh:mm: ss)

%U-Week (00-53) where Sunday is the first day of the week

%u-Week (00-53) where Monday is the first day of the week

%V-Week (01-53) where Sunday is the first day of the week, used with %X

%v-Week (01-53) where Monday is the first day of the week, used with %x

%W-Weekday name (Sunday-Saturday)

%w-Day of the week (0=Sunday, 6=Saturday)

%X-Year for the week where Sunday is the first day of the week, four digits, used with %V

%x-Year for the week where Monday is the first day of the week, four digits, used with %v

%Y-Year, numeric, four digits

%y-Year, numeric, two digits

Creating tables with relationship

In database design, understanding and implementing relationships between entities is crucial. These relationships, such as one-to-one, one-to-many, and many-to-many, establish connections between tables using key constraints.

Let's explore each type of relationship with examples and SQL implementation.

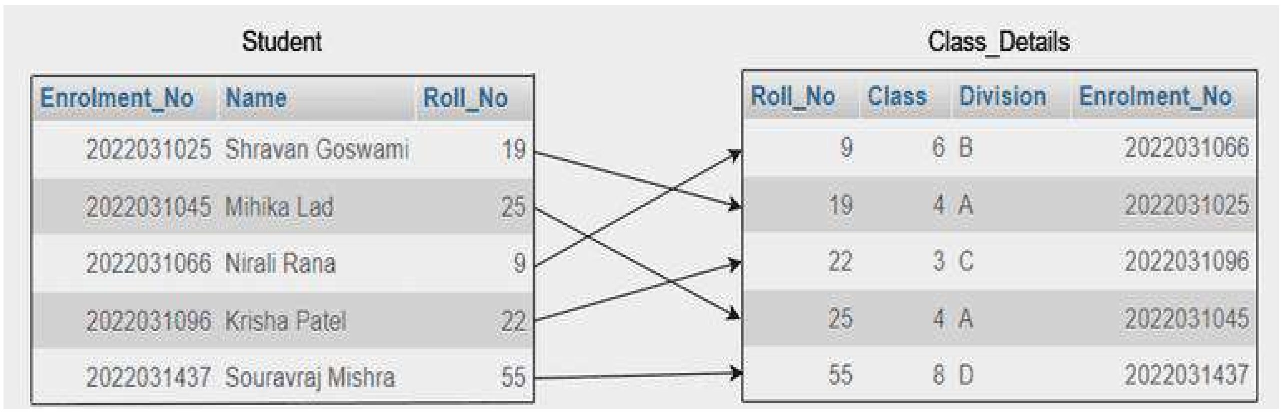
One-to-One Relationship

In this relationship, **each row/record of the** Let's **table is exactly related to one row in the second table** and **vice versa**.

Example: *Let's consider 2 tables "**Student**" and "**Class_Details**". Now "**Student**" has 2 columns - "**Enrolment_No**", "**Name**" and "**Class_Details**" has 4 columns "**Roll_No**", "**Class**", "**Division**", and "**Enrolment_No**".*

In the example, you can see that a student will have only one enrolment number allotted, only one roll number, and only one class and division. Then it may change year-wise or semester-wise but a student cannot have multiple classes or enrollment numbers or roll numbers simultaneously. Here, a one-to-one relationship occurs. Let's understand **One-to-One** relationships with the help of a table diagram and their output.

Create tables "**Student**" and "**Class_Details**" with required columns or attributes as mentioned in the above example.



Student

Enrolment_No	Name	Roll_No
2022031025	Shravan Goswami	19
2022031045	Mihika Lad	25
2022031066	Nirali Rana	9
2022031096	Krisha Patel	22
2022031437	Souravraj Mishra	55

Class_Details

Roll_No	Class	Division	Enrolment_No
9	6	B	2022031066
19	4	A	2022031025
22	3	C	2022031096
25	4	A	2022031045
55	8	D	2022031437

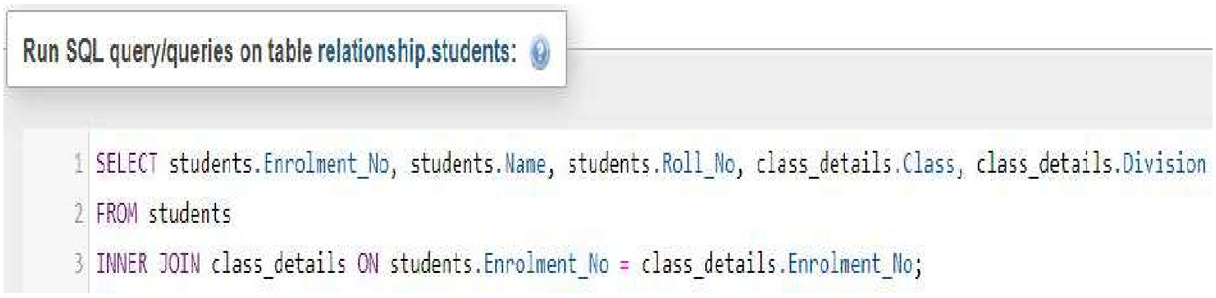
One-to-one relationship between student and class_details

Here, you can see that each student's table record is exactly related to one record in the class_details table. Each student has a unique Enrolment number and Roll number. It is not possible for a student to be enrolled in 2 classes or divisions at same time. Each student is related from its name details and its class uniquely.

SQL Query

Now, to retrieve data from the tables we write the following query to view the details of both the tables.

```
SELECT students.Enrolment_No, students.Name, students.Roll_No, class_details.Class,
class_details.Division
FROM students
JOIN class_details ON students.Enrolment_No = class_details.Enrolment_No;
```



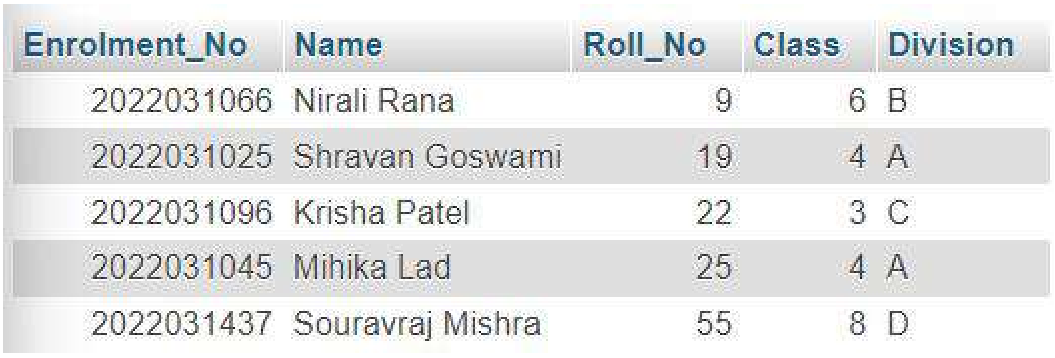
Run SQL query/queries on table relationship.students:

```
SELECT students.Enrolment_No, students.Name, students.Roll_No, class_details.Class, class_details.Division
FROM students
INNER JOIN class_details ON students.Enrolment_No = class_details.Enrolment_No;
```

Query for retrieving records from both the table

The above query retrieves records using the primary key and foreign key of another table which is in turn a primary key of the first table. The above query uses '**Enrolment_No**' for referencing the records from both tables. In this query, we retrieved data using the primary key of the Student table and as a foreign key of the class_details table.

Output:

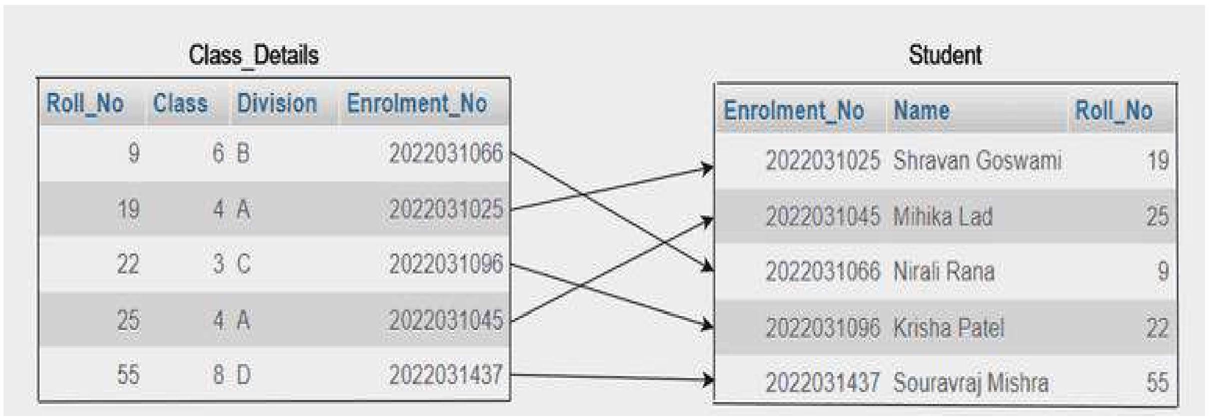


Enrolment_No	Name	Roll_No	Class	Division
2022031066	Nirali Rana	9	6	B
2022031025	Shravan Goswami	19	4	A
2022031096	Krisha Patel	22	3	C
2022031045	Mihika Lad	25	4	A
2022031437	Souravraj Mishra	55	8	D

Explanation: Through the above output or result we can see that each record of the first table is related exactly to one another record in the next table. You can also notice that in one-to-one relationship output, no multiple records are possible.

one-to-one relationship(vice-versa)

In the definition, we saw that the one-to-one relationship is also vice-versa, this means the **class_details** table is also related to exactly one record in students table.



Class_Details

Roll_No	Class	Division	Enrolment_No
9	6	B	2022031066
19	4	A	2022031025
22	3	C	2022031096
25	4	A	2022031045
55	8	D	2022031437

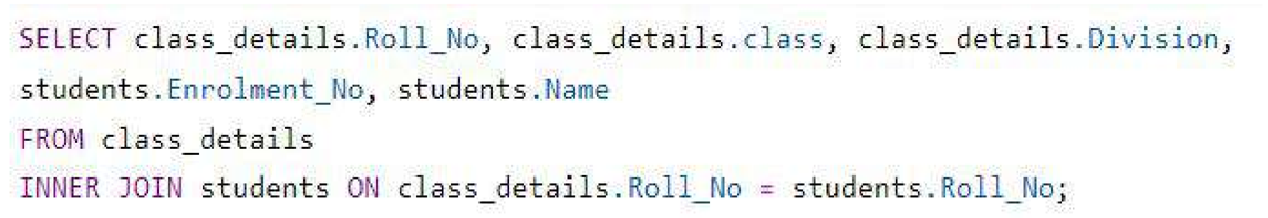
Student

Enrolment_No	Name	Roll_No
2022031025	Shravan Goswami	19
2022031045	Mihika Lad	25
2022031066	Nirali Rana	9
2022031096	Krisha Patel	22
2022031437	Souravraj Mishra	55

One-to-one relation from class_details to students entity representing vice-versa condition

Let's see the query for a one-to-one relationship from the class_details table to the student's table.

SELECT class_details.Roll_No, class_details.Class, class_details.Division,
students.Enrolment_No, students.Name
FROM class_details



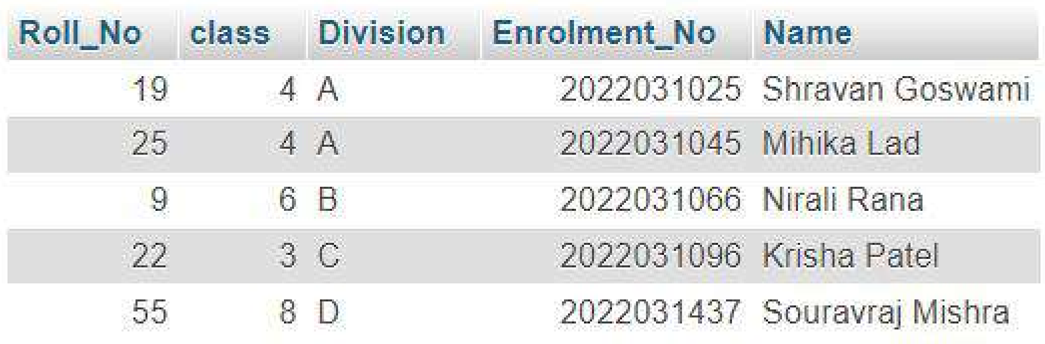
```sql
SELECT class_details.Roll_No, class_details.class, class_details.Division,
students.Enrolment_No, students.Name
FROM class_details
INNER JOIN students ON class_details.Roll_No = students.Roll_No;
```

JOIN students ON class_details.Roll_No = students.Roll_No;

The query for vice-versa condition i.e. one-to-one relationship from class_details to students

As you can see, in the above query, we used the class_details table primarily to retrieve the data using presenting one-to-one relation.

Output:



Roll_No	class	Division	Enrolment_No	Name
19	4	A	2022031025	Shravan Goswami
25	4	A	2022031045	Mihika Lad
9	6	B	2022031066	Nirali Rana
22	3	C	2022031096	Krisha Patel
55	8	D	2022031437	Souravraj Mishra

Explanation: You can see that we have the same number of records that we got in the output from the students table to the class_details table i.e. 5 and it is the same number of records while retrieving from class_details to the students table. You can also see that each record is associated with only one record in another table. We got 5 records from both way of retrieving the data using one-to-one relation from table students and **class_details** and **vice-versa**.

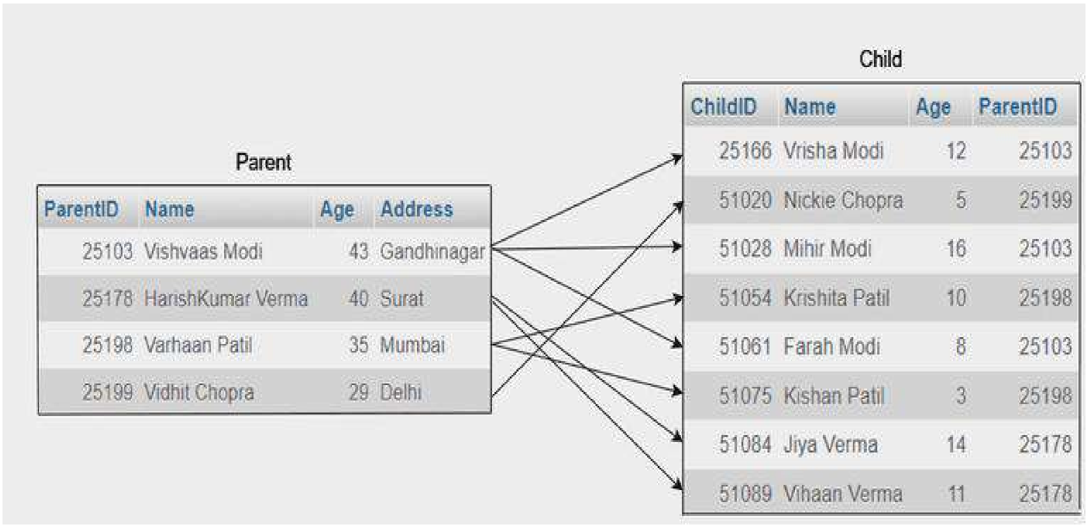
One-to-Many Relationship

In this relationship, **each row/record of the Let's first table can be related to multiple rows in the second table.**

Example:

Let's consider 2 tables "**Parent**" and "**Child**". Now the "Parent" table has columns "**ParentID**", "**Name**"... and so on, and "Child" has columns or attributes "**ChildID**", "**Name**", "**Age**", "**ParentID**" and so on.

In the above example we used the logic of parent and child and you can see that a parent



Parent

ParentID	Name	Age	Address
25103	Vishvaas Modi	43	Gandhinagar
25178	HarishKumar Verma	40	Surat
25198	Varhaan Patil	35	Mumbai
25199	Vidhit Chopra	29	Delhi

Child

ChildID	Name	Age	ParentID
25166	Vrisha Modi	12	25103
51020	Nickie Chopra	5	25199
51028	Mihir Modi	16	25103
51054	Krishita Patil	10	25198
51061	Farah Modi	8	25103
51075	Kishan Patil	3	25198
51084	Jiya Verma	14	25178
51089	Vihaan Verma	11	25178

can have many children, but a child cannot have many/multiple parents.

Create tables '**parent**' and '**child**' as mentioned in the above table and let's see **one-to-many** relation between these 2 tables.

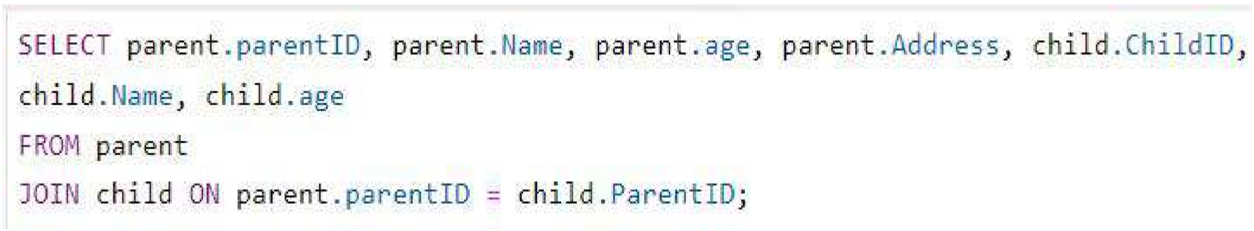
one-to-many relationship between parent and child table

We can see that each record from the table parent is associated with one or more than one record(s) in the child table representing one-to-many relationships. Here, a one-to-many relationship occurs when the vice-versa of the above condition should not be true. If the vice-versa or reverse condition for a one-to-many relationship becomes true then it is a many-to-many relationship.

SQL Query:

Let's see the query to retrieve data representing **one-to-many relationships** parent-to-child:

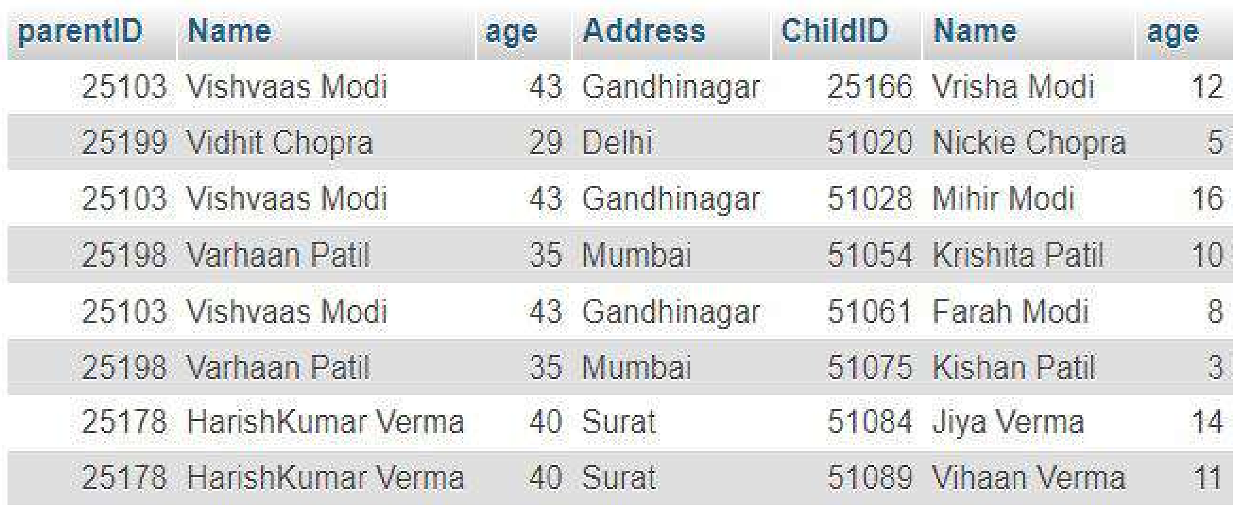
SELECT parent.parentID, parent.Name, parent.age, parent.Address, child.ChildID,
child.Name, child.age
FROM parent
JOIN child ON parent.parentID = child.ParentID



```
SELECT parent.parentID, parent.Name, parent.age, parent.Address, child.ChildID,
child.Name, child.age
FROM parent
JOIN child ON parent.parentID = child.ParentID;
```

Query for retrieving data from parent and child table

The above query retrieves all possible records from parent-to-child relationship entities.



parentID	Name	age	Address	ChildID	Name	age
25103	Vishvaas Modi	43	Gandhinagar	25166	Vrisha Modi	12
25199	Vidhit Chopra	29	Delhi	51020	Nickie Chopra	5
25103	Vishvaas Modi	43	Gandhinagar	51028	Mihir Modi	16
25198	Varhaan Patil	35	Mumbai	51054	Krishita Patil	10
25103	Vishvaas Modi	43	Gandhinagar	51061	Farah Modi	8
25198	Varhaan Patil	35	Mumbai	51075	Kishan Patil	3
25178	HarishKumar Verma	40	Surat	51084	Jiya Verma	14
25178	HarishKumar Verma	40	Surat	51089	Vihaan Verma	11

Output for the above query representing one-to-many relationships

Explanation: In the output of the above query, you can see that the parent's name is occurring more than one time with different child names representing a parent associated with one or more than one child. For example, we can see that **HarishKumar Verma** has two children, **Jiya Verma**, and **Vihaan Verma**, **Vishvaas Modi** has 3 children i.e. **Vrisha Modi**, **Mihir Modi**, and **Farah Modi**, and last **Vidhit Chopra** has only a single child i.e. Nickie Chopra representing one record/row associated with one or more than one child.

Each record from the child table is associated with only one record in the parent table and it also fits the logic that a parent may have one or more than one child but a child cannot have multiple parents. Similarly, you can also design the relation as per the business logic which fits into which relationship better.

Many-to-Many Relationship

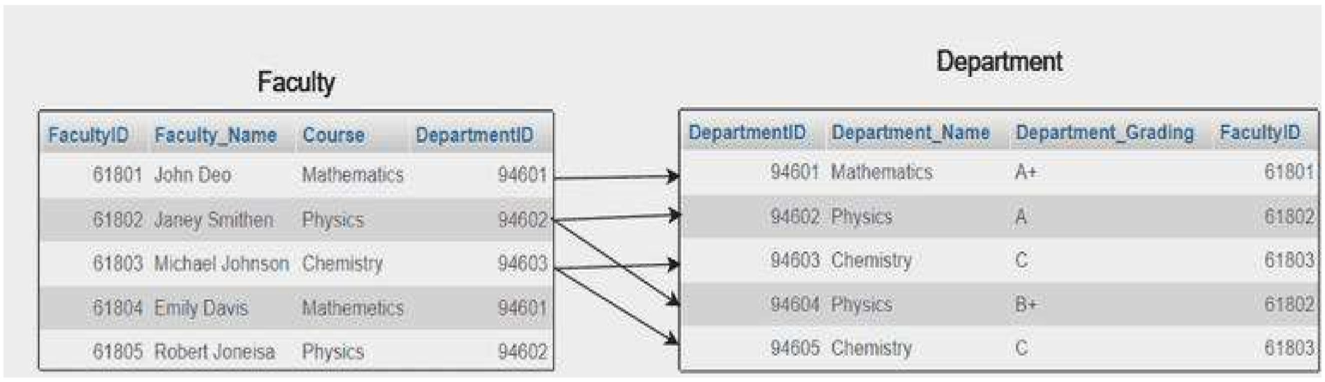
In this relationship, multiple rowsthe /record of first table can be related to multiple rows in the second table and vice versa.

Example: Consider tables "Faculty" and "Department". Now the "Faculty" table has columns "FacultyID", "Faculty_Name", "Course" and so on, and "Department" has columns "DepartmentID", "Department_Name" and so on.

Now, each faculty member can be associated with multiple departments, and each department can be associated with multiple faculty members establishing many-to-many relationships.

Create Table:

Example of many-to-many relationships from faculty to department entities:



Faculty

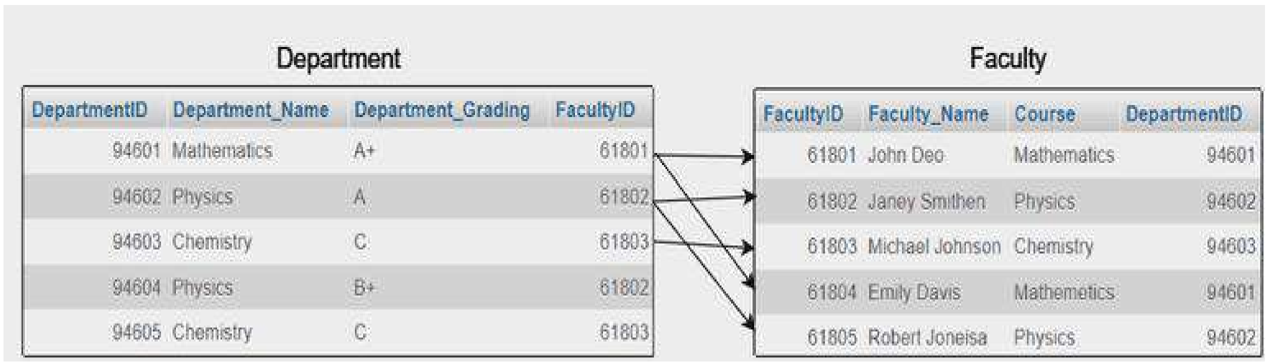
FacultyID	Faculty_Name	Course	DepartmentID
61801	John Deo	Mathematics	94601
61802	Janey Smithen	Physics	94602
61803	Michael Johnson	Chemistry	94603
61804	Emily Davis	Mathemetics	94601
61805	Robert Joneisa	Physics	94602

Department

DepartmentID	Department_Name	Department_Grading	FacultyID
94601	Mathematics	A+	61801
94602	Physics	A	61802
94603	Chemistry	C	61803
94604	Physics	B+	61802
94605	Chemistry	C	61803

Example of many-to-many relationships from faculty to department entities

Example of many-to-many relationship from department to faculty entities:



Department

DepartmentID	Department_Name	Department_Grading	FacultyID
94601	Mathematics	A+	61801
94602	Physics	A	61802
94603	Chemistry	C	61803
94604	Physics	B+	61802
94605	Chemistry	C	61803

Faculty

FacultyID	Faculty_Name	Course	DepartmentID
61801	John Deo	Mathematics	94601
61802	Janey Smithen	Physics	94602
61803	Michael Johnson	Chemistry	94603
61804	Emily Davis	Mathemetics	94601
61805	Robert Joneisa	Physics	94602

Example of many-to-many relationship from department to faculty entities

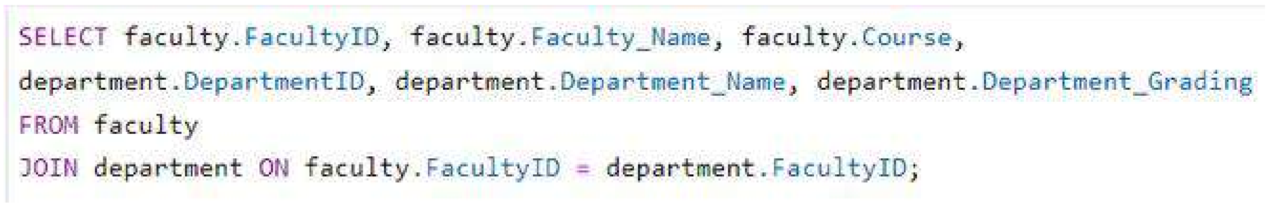
Now, in above example as per the business logic, you can see that faculty works in more than one department and vice-versa is also true that departments have multiple faculties working with them.

Now if we see technically then, records in the faculty table are associated with more than one record in the department table, and records from department table are associated with more than one faculties or records in the faculty entity.

SQL Query:

Now let's see the query for retrieving data using record association from faculty to department entity referencing using **FacultyID** as the primary key.

SELECT faculty.FacultyID, faculty.Faculty_Name, faculty.Course,
department.DepartmentID, department.Department_Name,
department.Department_Grading
FROM faculty
JOIN department ON faculty.FacultyID = department.FacultyID

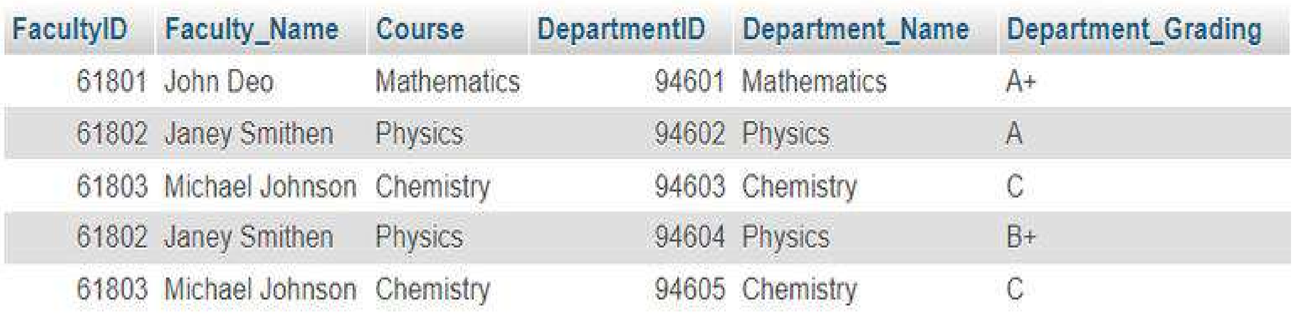


```
SELECT faculty.FacultyID, faculty.Faculty_Name, faculty.Course,
department.DepartmentID, department.Department_Name, department.Department_Grading
FROM faculty
JOIN department ON faculty.FacultyID = department.FacultyID;
```

faculty to department table

This query works like, the FacultyID matched or present in records of the department table will get associated with it. Let's understand by looking at the output

Output:



FacultyID	Faculty_Name	Course	DepartmentID	Department_Name	Department_Grading
61801	John Deo	Mathematics	94601	Mathematics	A+
61802	Janey Smithen	Physics	94602	Physics	A
61803	Michael Johnson	Chemistry	94603	Chemistry	C
61802	Janey Smithen	Physics	94604	Physics	B+
61803	Michael Johnson	Chemistry	94605	Chemistry	C

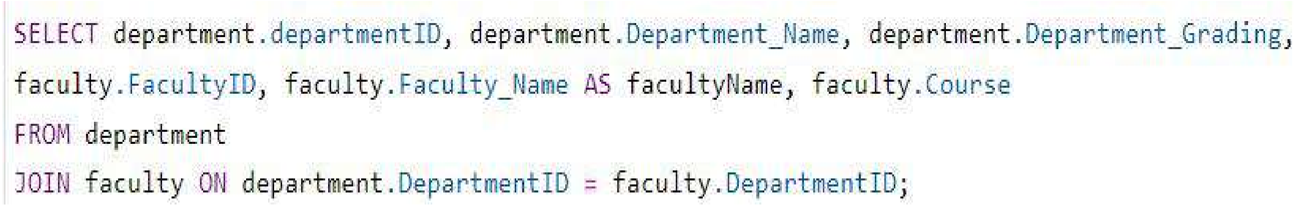
Output of above query from faculty to department table

Explanation: FacultyID present in the department table will be associated with those records in department which have the FacultyID as the same in the faculty table. For example, **FacultyID 61801**, **61802**, **61803** are associated with Departments so the FacultyID is repeating for representing multiple departments

Now let's see the query from record association from the department table to the faculty table to represent the many-to-many relationship.

<u>**SQL Query:**</u>

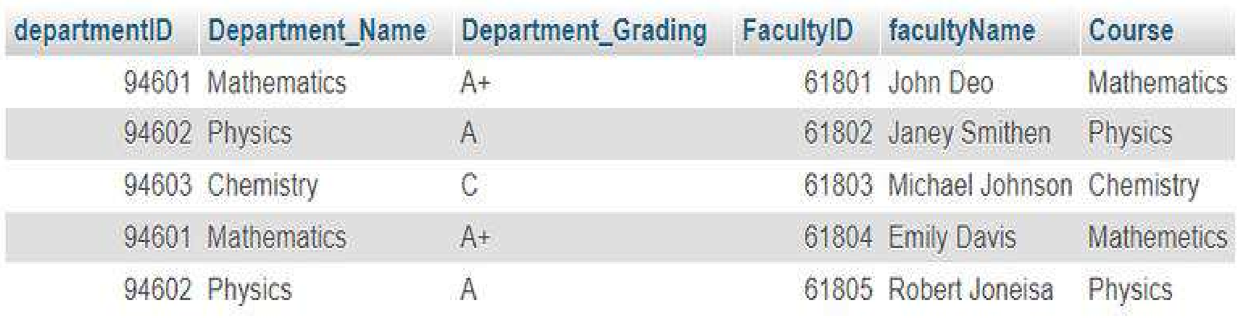
SELECT department.departmentID, department.Department_Name, department.Department_Grading, faculty.FacultyID, faculty.Faculty_Name AS facultyName, faculty.Course
FROM department
JOIN faculty ON department.DepartmentID = faculty.DepartmentID;



```
SELECT department.departmentID, department.Department_Name, department.Department_Grading,
faculty.FacultyID, faculty.Faculty_Name AS facultyName, faculty.Course
FROM department
JOIN faculty ON department.DepartmentID = faculty.DepartmentID;
```

Retrieving records/data using departmentID as a reference from department to faculty table

Output:



departmentID	Department_Name	Department_Grading	FacultyID	facultyName	Course
94601	Mathematics	A+	61801	John Deo	Mathematics
94602	Physics	A	61802	Janey Smithen	Physics
94603	Chemistry	C	61803	Michael Johnson	Chemistry
94601	Mathematics	A+	61804	Emily Davis	Mathemetics
94602	Physics	A	61805	Robert Joneisa	Physics

This is the output of the above query

Explanation: Now, in the above example you can see that the department name, and grading is repeating multiple times to represent multiple faculties.

Conclusion

We learned types of relationships i.e. **one-to-one**, **one-to-many**, **many-to-many** relationships between entities, or while designing tables using SQL. The implementation of these relationships and how to apply them as per the business logic.

DBMS Integrity Constraints

Integrity constraints are the set of predefined rules that are used to maintain the quality of information. Integrity constraints ensure that the data insertion, data updating, data deleting and other processes have to be performed in such a way that the data integrity is not affected. They act as guidelines ensuring that data in the database remain accurate and consistent. So, integrity constraints are used to protect databases. The various types of integrity constraints are

Types of Integrity Constraints:

Domain Constraints

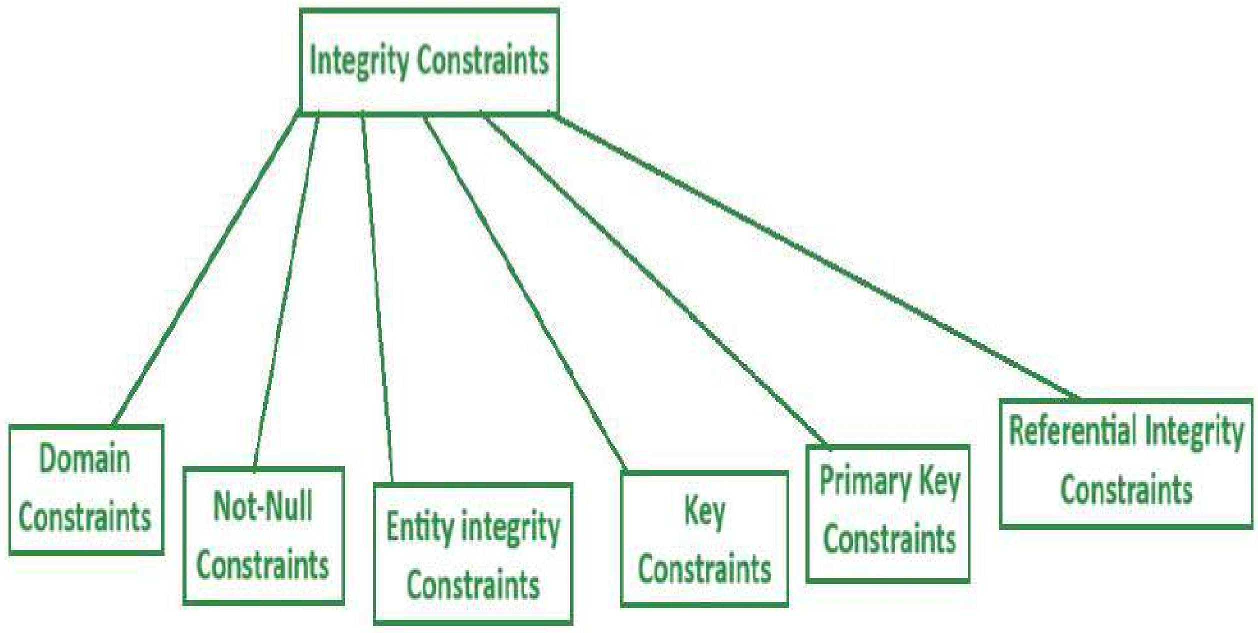
Not-Null Constraints

Entity integrity Constraints

Key Constraints

Primary Key Constrains

Referential integrity constraints



Domain Constraints

These are defined as the definition of valid set of values for an attribute. The data type of domain include string, char, time, integer, date, currency etc. The value of the attribute must be available in comparable domains.

Example:

Student_Id	**Name**	**Semester**	**Age**
21CSE100	Ramesh	5th	20
21CSE101	Kamlesh	5th	21
21CSE102	Aakash	5th	22
21CSE103	Mukesh	5th	20

Not-Null Constraints

It specifies that within a tuple, attributes overs which not-null constraint is specified must not contain any null value.

Example:

The not-null constraint be specified on the "Semester" attribute in the relation/table given below, then the data entry of 4th tuple will violate this integrity constraint, because the "Semester" attribute in this tuple contains null value. To make this database instance a legal instance, its entry must not be allowed by database management system.

Student_id	**Name**	**Semester**	**Age**
21CSE100	Ramesh	5th	20
21CSE101	Kamlesh	5th	21
21CSE102	Akash	5th	22
21CSE103	Mukesh		20

Entity Integrity Constraints

Entity integrity constraints state that primary key can never contain null value because primary key is used to determine individual rows in a relation uniquely, if primary key contains null value then we cannot identify those rows. A table can contain null value in it except primary key field.

Example:

It is not allowed because it is containing primary key as NULL value.

Student_id	**Name**	**Semester**	**Age**
21CSE101	Ramesh	5th	20
21CSE102	Kamlesh	5th	21

Student_id	**Name**	**Semester**	**Age**
21CSE103	Aakash	5th	22
	Mukesh	5th	20

Key Constraints

Keys are the entity set that are used to identify an entity within its entity set uniquely. An entity set can contain multiple keys, bit out of them one key will be primary key. A primary key is always unique, it does not contain any null value in table.

Example:

Student_id	**Name**	**Semester**	**Age**
21CSE101	Ramesh	5th	20
21CSE102	Kamlesh	5th	21
21CSE103	Aakash	5th	22
21CSE102	Mukesh	5th	20

It is now acceptable because all rows must be unique.

Primary Key Constraints

It states that the primary key attributes are required to be unique and not null. That is, primary key attributes of a relation must not have null values and primary key attributes of two tuples must never be same. This constraint is specified on database schema to the primary key attributes to ensure that no two tuples are same.

Example:

Here, in the below example the Student_id is the primary key attribute. The data entry of 4th tuple violates the primary key constraint that is specifies on the database schema and therefore this instance of database is not a legal instance.

Student_id	**Name**	**Semester**	**Age**
21CSE101	Ramesh	5th	20
21CSE102	Kamlesh	5th	21
21CSE103	Akash	5th	22
21CSE103	Mukesh	5th	20

Referential integrity constraints

It can be specified between two tables. In case of referential integrity constraints, if a Foreign key in Table 1 refers to Primary key of Table 2 then every value of the Foreign key in Table 1 must be null or available in Table 2.

Example:

Here, in below example Block_No 22 entry is not allowed because it is not present in 2nd table.

Student_id	**Name**	**Semester**	**Block_No**
22CSE101	Ramesh	5th	20
21CSE105	Kamlesh	6th	21
22CSE102	Aakash	5th	20
23CSE106	Mukesh	2nd	22

Block_No	Block Location
20	Chandigarh
21	Punjab
25	Delhi

Conclusion

Integrity constraints act as the backbone of reliable and robust database. They ensure that the data stored is reliable, worthy, consistent and accurate within the database. By implement integrity constraints we can improve the quality of the data stored in database. So, as the database continues to grow it will not become inconsistent and inaccurate.

Nested Queries in SQL

A **nested query** (also called a **subquery**) is a query embedded within another query. The subquery is executed first, and its result is used by the main query. Subqueries can be used in SELECT, FROM, or WHERE clauses.

Types of Nested Queries

Single Row Subquery: Returns a single value.

Multiple Row Subquery: Returns multiple rows.

Multiple Column Subquery: Returns multiple columns.

Correlated Subquery: Subquery depends on the outer query.

Syntax of a Nested Query

```
SELECT column_name(s) FROM table_name WHERE column_name OPERATOR
(SELECT column_name FROM another_table WHERE condition);
```

The subquery is executed first.

Its output is passed to the outer query.

Examples of Nested Queries

1. Single Row Subquery

Example: Find employees with a salary greater than the average salary.

Tables: **Employees**

EmployeeID	**Name**	**Salary**
1	Alice	4000
2	Bob	6000
3	Carol	8000
4	David	7000

Query:

```
SELECT Name, Salary FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Execution:

Subquery: (SELECT AVG(Salary) FROM Employees) calculates the average salary (6250).

Outer Query: Fetches employees with Salary > 6250.

Result:

Name	**Salary**
Carol	8000
David	7000

2. Multiple Row Subquery

Example: Find employees who work in departments located in New York.

Tables:

Departments

DeptID	**DeptName**	**Location**
1	HR	New York
2	IT	California

DeptID	DeptName	Location
3	Finance	New York

Employees

EmployeeID	Name	DeptID
1	Alice	1
2	Bob	2
3	Carol	3

Query:

SELECT Name FROM Employees WHERE DeptID IN (SELECT DeptID FROM Departments WHERE Location = 'New York');

Execution:

Subquery: Fetches DeptID of departments in New York → (1, 3).

Outer Query: Fetches employees with matching DeptID.

Result:

Name
Alice
Carol

---

3. Correlated Subquery

A **correlated subquery** depends on the outer query for its values.

Example: Find employees whose salary is greater than the average salary of their department.

Tables:

Employees

EmployeeID	Name	Salary	DeptID
1	Alice	4000	1
2	Bob	6000	2
3	Carol	7000	1

EmployeeID	**Name**	**Salary**	**DeptID**
4	David	8000	2

Query:

SELECT Name, Salary FROM Employees E1 WHERE Salary > (SELECT AVG(Salary) FROM Employees E2 WHERE E1.DeptID = E2.DeptID);

Execution:

The subquery calculates the **average salary** for each department (DeptID).

The outer query checks if the employee's salary is greater than their department's average salary.

Result:

Name	**Salary**
Carol	7000
David	8000

---

4. Nested Query in the FROM Clause

Subqueries in the FROM clause act as temporary tables.

Example: Find the average salary of employees by department.

Query:

SELECT DeptID, AVG_Salary FROM (SELECT DeptID, AVG(Salary) AS AVG_Salary FROM Employees GROUP BY DeptID) AS TempTable;

Execution:

Subquery calculates the AVG(Salary) for each DeptID and returns a table.

Outer query fetches the results from the temporary table.

Result:

DeptID	**AVG_Salary**
1	5500
2	7000

5. Nested Query with EXISTS

The EXISTS operator checks whether a subquery returns any rows.

Example: Find employees who belong to departments located in New York.

Query:

```
SELECT Name FROM Employees E WHERE EXISTS (SELECT 1 FROM Departments D
WHERE E.DeptID = D.DeptID AND D.Location = 'New York');
```

Execution:

The subquery checks if a matching DeptID exists with Location = 'New York'.

If true, the outer query includes the employee.

Result:

Name
Alice
Carol

Subqueries in SQL

A **subquery** is a query nested inside another query. The result of the subquery is used by the main query to perform further operations. Subqueries are also called **inner queries**, and the main query is called the **outer query**.

Syntax of Subqueries

```
SELECT column1, column2 FROM table1 WHERE column_name OPERATOR (SELECT
column_name FROM table2 WHERE condition);
```

The **subquery** is placed inside parentheses ().

The **result of the subquery** can be a single value, multiple rows, or even a table.

Types of Subqueries

Single-Row Subquery – Returns one row and one column.

Multi-Row Subquery – Returns multiple rows.

Multi-Column Subquery – Returns multiple columns.

Correlated Subquery – Subquery depends on the outer query.

Nested Subquery – Subqueries inside other subqueries.

Examples of Subqueries

1. Single-Row Subquery

Returns a single value and uses operators like =, >, <, etc.

Example: Find employees whose salary is greater than the average salary.

Table: **Employees**

EmployeeID	**Name**	**Salary**
1	Alice	4000
2	Bob	6000
3	Carol	8000
4	David	7000

Query:

SELECT Name, Salary FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);

Subquery: Calculates AVG(Salary).

Outer Query: Finds employees with salary greater than the average.

Result:

Name	**Salary**
Carol	8000
David	7000

---

2. Multi-Row Subquery

Returns multiple rows and is used with operators like IN, ANY, ALL.

Example: Find employees who work in departments located in **New York**.

Tables:

Departments

DeptID	DeptName	Location
1	HR	New York
2	IT	California
3	Finance	New York

Employees

EmployeeID	Name	DeptID
1	Alice	1
2	Bob	2
3	Carol	3

Query:

SELECT Name FROM Employees WHERE DeptID IN (SELECT DeptID FROM Departments WHERE Location = 'New York');

Subquery: Fetches DeptID values for departments located in New York.

Outer Query: Finds employees whose DeptID matches.

Result:

Name
Alice
Carol

---

3. Multi-Column Subquery

Returns multiple columns and is often used in comparison.

Example: Find employees with the same salary and department as 'Alice'.

Query:

SELECT Name, Salary, DeptID FROM Employees WHERE (Salary, DeptID) = (SELECT Salary, DeptID FROM Employees WHERE Name = 'Alice');

Subquery: Returns Salary and DeptID for Alice.

Outer Query: Finds employees with the same Salary and DeptID.

Result:

Name	**Salary**	**DeptID**
Alice	4000	1

---

4. Correlated Subquery

A **correlated subquery** depends on the outer query for its values. It is executed for each row in the outer query.

Example: Find employees whose salary is greater than the average salary of their department.

Query:

```
SELECT Name, Salary FROM Employees E1 WHERE Salary > (SELECT AVG(Salary) FROM Employees E2 WHERE E1.DeptID = E2.DeptID);
```

Subquery: Calculates the average salary for each department.

Outer Query: Checks if an employee's salary is greater than the average salary of their department.

---

5. Subquery in the FROM Clause

A subquery can act as a **derived table** in the FROM clause.

Example: Find the maximum salary by department.

Query:

```
SELECT DeptID, MAX_Salary FROM (SELECT DeptID, MAX(Salary) AS MAX_Salary FROM Employees GROUP BY DeptID) AS TempTable;
```

Subquery: Calculates MAX(Salary) for each department.

Outer Query: Fetches the result.

---

6. Subquery with EXISTS

EXISTS checks if the subquery returns any rows.

Example: Find departments that have employees.

Query:

```
SELECT DeptID, DeptName FROM Departments D WHERE EXISTS (SELECT 1 FROM Employees E WHERE E.DeptID = D.DeptID);
```

Subquery: Checks if a department has employees.

Outer Query: Returns departments where the subquery is true.

Grouping in SQL

Grouping in SQL is done using the GROUP BY clause. It is used to group rows that have the same values in specified columns into **summary rows** (like totals or averages). Often, it is used with **aggregate functions** such as SUM, AVG, COUNT, MIN, and MAX.

Syntax

SELECT column_name, aggregate_function(column_name) FROM table_name WHERE condition GROUP BY column_name;

column_name: The column(s) on which the grouping is performed.

aggregate_function: Functions like SUM, AVG, COUNT, etc., applied to the grouped data.

Key Points about **GROUP BY**

The GROUP BY clause groups rows with the same value in specified columns.

Aggregate functions operate on each group, not on individual rows.

The GROUP BY clause must appear after the WHERE clause but before the ORDER BY clause.

Columns in the SELECT statement must either be part of the GROUP BY clause or used within an aggregate function.

Example 1: Basic GROUP BY

Problem: Find the total salary for each department.

Table: Employees

EmployeeID	**Name**	**Department**	**Salary**
1	Alice	HR	4000
2	Bob	IT	6000
3	Carol	HR	5000
4	David	IT	7000

EmployeeID	**Name**	**Department**	**Salary**
5	Eve	Finance	8000

Query:

SELECT Department, SUM(Salary) AS TotalSalary FROM Employees GROUP BY Department;

Result:

Department	**TotalSalary**
HR	9000
IT	13000
Finance	8000

Example 2: GROUP BY with COUNT

Problem: Count the number of employees in each department.

Query:

SELECT Department, COUNT(EmployeeID) AS TotalEmployees FROM Employees GROUP BY Department;

Result:

Department	**TotalEmployees**
HR	2
IT	2
Finance	1

Example 3: GROUP BY with WHERE Clause

Problem: Find the total salary for each department where the salary is greater than 5000.

Query:

SELECT Department, SUM(Salary) AS TotalSalary FROM Employees WHERE Salary > 5000 GROUP BY Department;

Result:

Department	**TotalSalary**
IT	13000
Finance	8000

Example 4: GROUP BY with Multiple Columns

Problem: Find the total salary for each department and each job title.

Table: Employees

EmployeeID	**Name**	**Department**	**JobTitle**	**Salary**
1	Alice	HR	Manager	4000
2	Bob	IT	Developer	6000
3	Carol	HR	Assistant	5000
4	David	IT	Developer	7000
5	Eve	Finance	Analyst	8000

Query:

```
SELECT Department, JobTitle, SUM(Salary) AS TotalSalary FROM Employees GROUP
BY Department, JobTitle;
```

Result:

Department	**JobTitle**	**TotalSalary**
HR	Manager	4000
HR	Assistant	5000
IT	Developer	13000
Finance	Analyst	8000

Example 5: GROUP BY with HAVING Clause

The HAVING clause is used to filter groups after grouping, similar to the WHERE clause but for groups.

Problem: Find departments where the total salary is greater than 10000.

Query:

SELECT Department, SUM(Salary) AS TotalSalary FROM Employees GROUP BY Department HAVING SUM(Salary) > 10000;

Result:

Department	**TotalSalary**
IT	13000

GROUP BY with ORDER BY

To sort the grouped results, use the ORDER BY clause.

Example: Sort the total salary for each department in descending order.

Query:

SELECT Department, SUM(Salary) AS TotalSalary FROM Employees GROUP BY Department ORDER BY TotalSalary DESC;

Result:

Department	**TotalSalary**
IT	13000
HR	9000
Finance	8000

Summary

GROUP BY is used to group rows that have the same values into aggregated results.

It works with aggregate functions like SUM, AVG, COUNT, MIN, MAX.

Use the **HAVING clause** to filter groups based on aggregate results.

Columns in SELECT must be either in the GROUP BY clause or part of an aggregate function. use **ORDER BY** to sort grouped results.

Aggregation in SQL

Aggregation in SQL involves using aggregate functions to summarize or compute values over a group of rows. These functions allow you to perform calculations like totals, averages, counts, and other summary statistics.

Common Aggregate Functions

Function	**Description**
SUM()	Returns the sum of all values.
AVG()	Returns the average of the values.
COUNT()	Counts the number of rows or values.
MIN()	Returns the smallest value.
MAX()	Returns the largest value.

Syntax of Aggregate Functions

SELECT aggregate_function(column_name) FROM table_name WHERE condition;

aggregate_function: One of SUM, AVG, COUNT, etc.

column_name: The column to perform aggregation on.

WHERE: Filters rows before applying the aggregate function.

Examples of Aggregation

1. SUM() – Total of Values

Problem: Calculate the total salary of all employees.

Table: Employees

EmployeeID	**Name**	**Department**	**Salary**
1	Alice	HR	4000
2	Bob	IT	6000
3	Carol	HR	5000
4	David	IT	7000

Query:

```
SELECT SUM(Salary) AS TotalSalary FROM Employees;
```

Result: **TotalSalary**
22000

---

2. AVG() – Average of Values

Problem: Calculate the average salary of employees.

Query:

```
SELECT AVG(Salary) AS AverageSalary FROM Employees;
```

Result:

AverageSalary
5500

---

3. COUNT() – Count the Rows

Problem: Count the number of employees in the company.

Query:

```
SELECT COUNT(EmployeeID) AS TotalEmployees FROM Employees;
```

Result:

TotalEmployees
4

---

4. MIN() and MAX() – Smallest and Largest Value

Problem: Find the minimum and maximum salary of employees.

Query:

```
SELECT MIN(Salary) AS MinimumSalary, MAX(Salary) AS MaximumSalary FROM
Employees;
```

Result:

MinimumSalary	**MaximumSalary**
4000	7000

Ordering in SQL

Ordering in SQL allows you to sort the result set of a query in ascending or descending order based on one or more columns. The ORDER BY clause is used to specify the sorting order.

Syntax of **ORDER by**

SELECT column1, column2, ... FROM table_name WHERE condition ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;

column1, column2, ...: The columns by which you want to sort the result set.

ASC (default): Sorts the result in **ascending order** (from smallest to largest, alphabetically A to Z).

DESC: Sorts the result in **descending order** (from largest to smallest, alphabetically Z to A).

Examples of Ordering in SQL

1. Simple Ordering in Ascending Order

Problem: Retrieve all employees, ordered by their salaries in ascending order.

Table: Employees

EmployeeID	**Name**	**Salary**
1	Alice	4000
2	Bob	6000
3	Carol	5000
4	David	7000

Query:

SELECT EmployeeID, Name, Salary FROM Employees ORDER BY Salary ASC;

Result:

EmployeeID	**Name**	**Salary**
1	Alice	4000

EmployeeID	**Name**	**Salary**
3	Carol	5000
2	Bob	6000
4	David	7000

2. Ordering in Descending Order

Problem: Retrieve all employees, ordered by their salaries in descending order.

Query:

```
SELECT EmployeeID, Name, Salary FROM Employees ORDER BY Salary DESC;
```

Result:

EmployeeID	**Name**	**Salary**
4	David	7000
2	Bob	6000
3	Carol	5000
1	Alice	4000

3. Ordering by Multiple Columns

You can sort by multiple columns by separating them with commas.

Problem: Retrieve employees ordered by their department, and within each department, ordered by salary in descending order.

Table: Employees

EmployeeID	**Name**	**Department**	**Salary**
1	Alice	HR	4000
2	Bob	IT	6000
3	Carol	HR	5000
4	David	IT	7000

Query:

SELECT EmployeeID, Name, Department, Salary FROM Employees ORDER BY Department ASC, Salary DESC;

Result:

EmployeeID	**Name**	**Department**	**Salary**
1	Alice	HR	4000
3	Carol	HR	5000
2	Bob	IT	7000
4	David	IT	6000

---

4. Ordering with NULL Values

By default, **NULL values** are sorted as the **lowest** values when using ASC and as the **highest** values when using DESC.

Problem: Retrieve employees and sort by salary, putting employees with NULL salaries at the top.

Table: Employees

EmployeeID	**Name**	**Salary**
1	Alice	4000
2	Bob	NULL
3	Carol	5000
4	David	7000

Query:

SELECT EmployeeID, Name, Salary FROM Employees ORDER BY Salary ASC NULLS FIRST;

Result:

EmployeeID	**Name**	**Salary**
2	Bob	NULL

EmployeeID	Name	Salary
1	Alice	4000
3	Carol	5000
4	David	7000

5. Using ORDER BY with Other Clauses

You can use ORDER BY in combination with the LIMIT clause to restrict the number of rows returned.

Problem: Retrieve the top 3 highest-paid employees.

Query (for MySQL or PostgreSQL):

```
SELECT EmployeeID, Name, Salary FROM Employees ORDER BY Salary DESC LIMIT 3;
```

Result:

EmployeeID	Name	Salary
4	David	7000
2	Bob	6000
3	Carol	5000

Important Points About **ORDER BY**

By default, ORDER BY sorts in **ascending** order (ASC), but you can explicitly use ASC or DESC to specify the sort order.

Sorting is case-sensitive in most SQL implementations. Generally, uppercase letters are sorted before lowercase letters.

ORDER BY is applied after filtering by WHERE but before limiting the results with LIMIT (if used).

You can sort by multiple columns by specifying them in the ORDER BY clause, and you can use different sort orders for each column.

SQL Views

Views in SQL are a type of **virtual table** that simplifies how users interact with data across one or more tables. Unlike **traditional tables**, a view in **SQL** does not store data on disk; instead, it dynamically retrieves data based on a pre-defined query each time it's accessed.

SQL views are particularly useful for managing complex queries, enhancing security, and presenting data in a simplified format. In this guide, we will cover the **SQL** create view statement, updating and deleting views, and using the WITH CHECK OPTION clause.

What is a View in SQL?

A view in SQL is a saved SQL query that acts as a virtual table. It can fetch data from one or more tables and present it in a customized format, allowing developers to:

Simplify Complex Queries: Encapsulate complex joins and conditions into a single object.

Enhance Security: Restrict access to specific columns or rows.

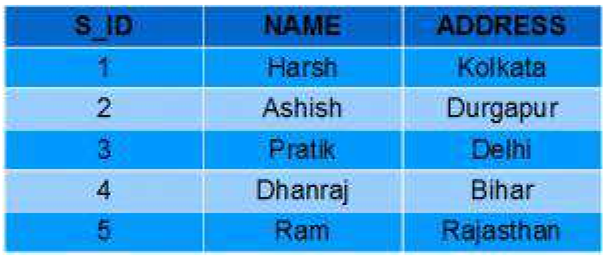
Present Data Flexibly: Provide tailored data views for different users.

Demo SQL Database

We will be using these **two SQL tables** for examples.

StudentDetails

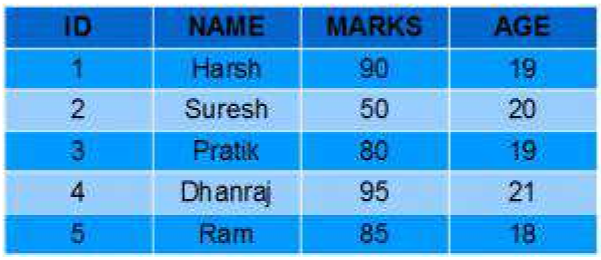
```
-- Create StudentDetails table
CREATE TABLE StudentDetails (
   S_ID INT PRIMARY KEY,
   NAME VARCHAR(255),
   ADDRESS VARCHAR(255)
);

INSERT INTO StudentDetails (S_ID, NAME, ADDRESS)
VALUES
   (1, 'Harsh', 'Kolkata'),
   (2, 'Ashish', 'Durgapur'),
   (3, 'Pratik', 'Delhi'),
   (4, 'Dhanraj', 'Bihar'),
   (5, 'Ram', 'Rajasthan');
```



S_ID	NAME	ADDRESS
1	Harsh	Kolkata
2	Ashish	Durgapur
3	Pratik	Delhi
4	Dhanraj	Bihar
5	Ram	Rajasthan

StudentMarks

```
-- Create StudentMarks table
CREATE TABLE StudentMarks (
    ID INT PRIMARY KEY,
    NAME VARCHAR(255),
    Marks INT,
    Age INT
);

INSERT INTO StudentMarks (ID, NAME, Marks, Age)
VALUES
    (1, 'Harsh', 90, 19),
    (2, 'Suresh', 50, 20),
    (3, 'Pratik', 80, 19),
    (4, 'Dhanraj', 95, 21),
    (5, 'Ram', 85, 18);
```



ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATE VIEWS in SQL

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

Syntax:

CREATE VIEW view_name AS

SELECT column1, column2.....

FROM table_name

WHERE condition;

Parameters:

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

SQL CREATE VIEW Statement Examples

Let's look at some examples of CREATE VIEW Statement in SQL to get a better understanding of how to create views in SQL.

Example 1: Creating View From a Single Table

In this example, we will create a View named DetailsView from the table StudentDetails. Query:

CREATE VIEW DetailsView **AS**

SELECT NAME, ADDRESS

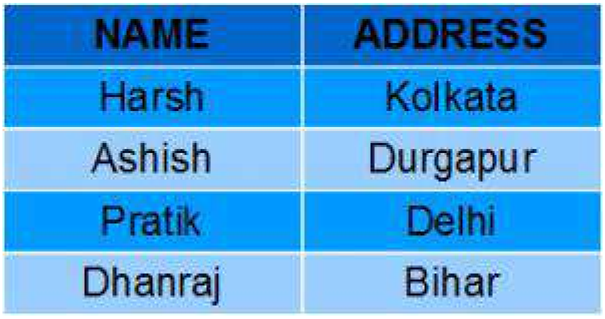
FROM StudentDetails

WHERE S_ID < 5;

To see the data in the View, we can query the view in the same manner as we query a table.

SELECT * FROM DetailsView;

Output:



NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Example 2: Create View From Table

In this example, we will create a view named StudentNames from the table StudentDetails. Query:

CREATE VIEW StudentNames **AS**

SELECT S_ID, NAME

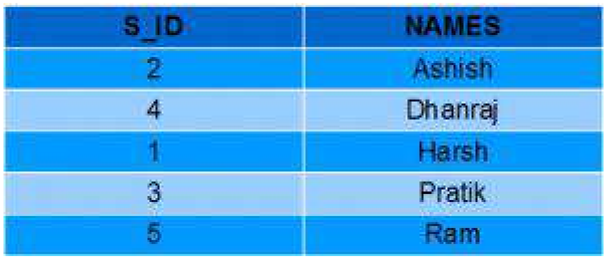
FROM StudentDetails

ORDER BY NAME;

If we now query the view as,

```
SELECT * FROM StudentNames;
```

Output:



S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

Example 3: Creating View From Multiple Tables

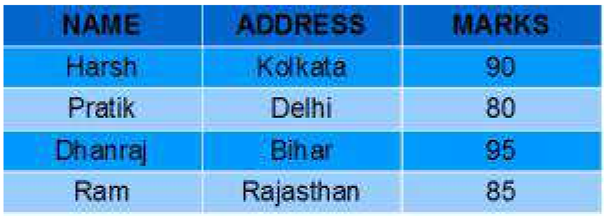
In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:

```
CREATE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:



NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

Listing all Views in a Database

We can list View using the **SHOW FULL TABLES** statement or using the **information_schema table**. A View can be created from a single table or multiple tables.

Syntax:

```
USE "database_name";

SHOW FULL TABLES WHERE table_type LIKE "%VIEW";
```

Using information_schema

```
SELECT table_name
```

FROM information_schema.views

WHERE table_schema = 'database_name';

OR

SELECT table_schema, table_name, view_definition

FROM information_schema.views

WHERE table_schema = 'database_name';

DELETE VIEWS in SQL

SQL allows us to delete an existing View. We can delete or drop View using the **DROP statement**.

Syntax:

DROP VIEW view_name;

Example

In this example, we are deleting the View **MarksView.**

DROP VIEW MarksView;

UPDATE VIEW in SQL

If you want to update the existing data within the view, use the **UPDATE** statement.

Syntax:

UPDATE view_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

Note: Not all views can be updated using the UPDATE statement.

If you want to update the view definition without affecting the data, use the **CREATE OR REPLACE VIEW** statement. you can use this syntax

CREATE OR REPLACE VIEW view_name **AS**

SELECT column1, column2, ...

FROM table_name

WHERE condition;

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.

The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.

The SELECT statement should not have the DISTINCT keyword.

The View should have all NOT NULL values.

The view should not be created using nested queries or complex queries.

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.

Examples

Let's look at different use cases for updating a view in SQL. We will cover these use cases with examples to get a better understanding.

Example 1: Update View to Add or Replace a View Field

We can use the **CREATE OR REPLACE VIEW** statement to add or replace fields from a view.

If we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table, we can do this by:

CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

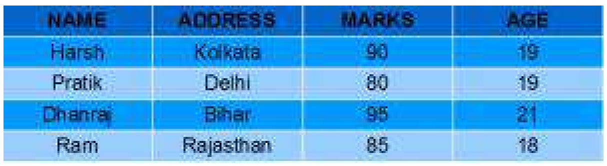
FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;

If we fetch all the data from MarksView now as:

SELECT * FROM MarksView;

Output:



NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Example 2: Update View to Insert a row in a view

We can insert a row in a View in the same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

In the below example, we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

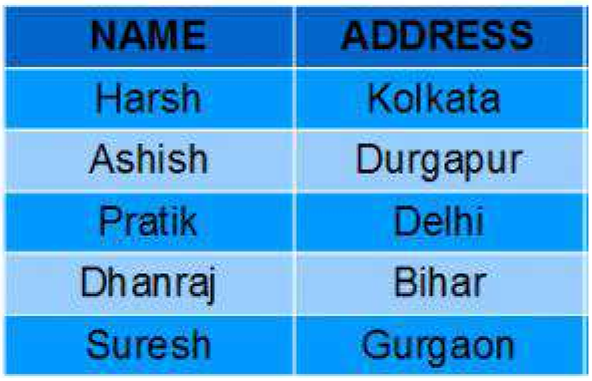
INSERT INTO DetailsView(NAME, ADDRESS)

VALUES("Suresh","Gurgaon");

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:



NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Example 3: Deleting a row from a View

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first deletes the row from the actual table and the change is then reflected in the view.

In this example, we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

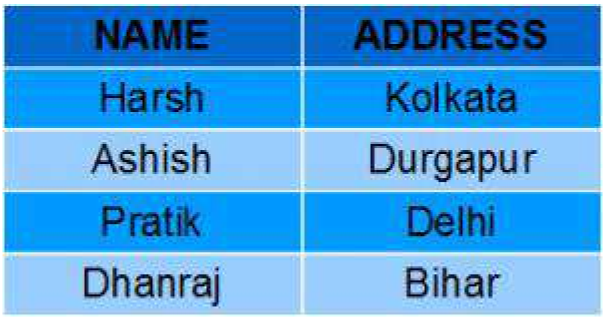
DELETE FROM DetailsView

WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

SELECT * FROM DetailsView;

Output:



NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

WITH CHECK OPTION Clause

The **WITH CHECK OPTION** clause in SQL is a very useful clause for views. It applies to an updatable view.

The WITH CHECK OPTION clause is used to prevent data modification (using INSERT or UPDATE) if the condition in the WHERE clause in the CREATE VIEW statement is not satisfied.

If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

WITH CHECK OPTION Clause Example:

In the below example, we are creating a View SampleView from the StudentDetails Table with a WITH CHECK OPTION clause.

CREATE VIEW SampleView AS

SELECT S_ID, NAME

FROM StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

In this view, if we now try to insert a new row with a null value in the NAME column then it will give an error because the view is created with the condition for the NAME column as NOT NULL. For example, though the View is updatable then also the below query for this View is not valid:

INSERT INTO SampleView(S_ID)

VALUES(6);

***NOTE**: The default value of NAME column is null.*

Uses of a View

A good database should contain views for the given reasons:

Restricting data access – Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.

Hiding data complexity – A view can hide the complexity that exists in multiple joined tables.

Simplify commands for the user – Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.

Store complex queries – Views can be used to store complex queries.

Rename Columns – Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in a select statement. Thus, renaming helps to hide the names of the columns of the base tables.

Multiple view facility – Different views can be created on the same table for different users.

Relational Set Operations in SQL

Relational **set operations** in SQL are used to combine the results of two or more SELECT queries. The results are treated as sets, and set operations allow you to perform tasks like finding the union, intersection, or difference of two result sets. The primary relational set operations in SQL are:

1. UNION

2. INTERSECT

3. EXCEPT (or MINUS)

4. UNION ALL

Each of these operations works on two result sets and follows the principles of set theory. These operations require that the result sets have the **same number of columns** and the corresponding columns must have compatible data types.

1. UNION

The UNION operator combines the results of two SELECT statements, removing duplicates. If you want to include duplicate rows, use UNION ALL.

Syntax:

```
SELECT column1, column2, ... FROM table1 UNION SELECT column1, column2, ... FROM table2;
```

Key points:

The columns in the SELECT statements must match in number and data type.

By default, UNION removes duplicate rows from the result.

To include duplicates, use UNION ALL.

Example of UNION

Consider two tables: Employees_1 and Employees_2.

Table 1: Employees_1

EmployeeID	**Name**
1	Alice
2	Bob
3	Carol

Table 2: Employees_2

EmployeeID	**Name**
2	Bob
4	David
5	Eve

Query to get the unique names from both tables:

```
SELECT Name FROM Employees_1 UNION SELECT Name FROM Employees_2;
```

Result:

Name
Alice
Bob
Carol
David
Eve

Here, the duplicate Bob is removed from the result set.

---

2. INTERSECT

The INTERSECT operator returns only the rows that are common between two SELECT statements. The result includes only the rows that exist in both tables.

Syntax:

```
SELECT column1, column2, ... FROM table1 INTERSECT SELECT column1, column2, ... FROM table2;
```

Key points:

The result set will only contain rows that exist in **both** tables.

Like UNION, the columns must match in number and data type.

Example of INTERSECT

Using the same Employees_1 and Employees_2 tables:

Query to get the common names between the two tables:

```
SELECT Name FROM Employees_1 INTERSECT SELECT Name FROM Employees_2;
```

Result:

Name
Bob

Here, only Bob is returned because it is the only common name in both tables.

3. EXCEPT (or MINUS in some databases)

The EXCEPT operator returns the rows that are in the first SELECT statement but not in the second. Some databases use MINUS instead of EXCEPT (e.g., Oracle).

Syntax:

```
SELECT column1, column2, ... FROM table1 EXCEPT SELECT column1, column2, ...
FROM table2;
```

Key points:

The result includes rows that appear in the first table but not in the second.

The columns in both SELECT statements must match in number and data type.

Example of EXCEPT

Using the same Employees_1 and Employees_2 tables:

Query to get the names in Employees_1 but not in Employees_2:

```
SELECT Name FROM Employees_1 EXCEPT SELECT Name FROM Employees_2;
```

Result:

Name
Alice
Carol

Here, Alice and Carol are returned because they are in Employees_1 but not in Employees_2.

4. UNION ALL

While not strictly a set operation in the mathematical sense, UNION ALL is an important extension. It combines the result sets of two SELECT statements without removing duplicates.

Syntax:

```
SELECT column1, column2, ... FROM table1 UNION ALL SELECT column1, column2, ... FROM table2;
```

Key points:

UNION ALL does **not** remove duplicates, so all rows from both tables are included, even if they are identical.

Example of UNION ALL

Using the same Employees_1 and Employees_2 tables:

Query to get all names, including duplicates:

```
SELECT Name FROM Employees_1 UNION ALL SELECT Name FROM Employees_2;
```

Result:

Name
Alice
Bob
Carol
Bob
David
Eve

Here, Bob appears twice because UNION ALL does not eliminate duplicates.

Key Points About Relational Set Operations

Matching Columns: The columns in the SELECT statements must match in **number** and **data type** for all set operations.

Null Handling: NULL values are treated as distinct, meaning they are not considered equal to each other in some cases (especially in EXCEPT and INTERSECT).

Performance: UNION ALL is generally faster than UNION because it does not perform the duplicate removal operation.

Database Compatibility: Not all databases support EXCEPT. Some use MINUS instead (e.g., Oracle).

<u>Use Cases for Set Operations</u>

UNION is used when you want to combine the results of two queries and eliminate duplicates.

INTERSECT is useful when you want to find common data between two queries.

EXCEPT helps you identify data in the first query that doesn't appear in the second.

UNION ALL is used when you want to combine results, including duplicates.

UNIT 4 – SCHEMA REFINEMENT (NORMALIZATION)

What is Data Normalization and Why Is It Important? Normalization is the process of reducing data redundancy in a table and improving data integrity. Data normalization is a technique used in databases to organize data efficiently. Have you ever faced a situation where data redundancy and anomalies affected the accuracy of your database? Data normalization ensures that your data remains clean, consistent, and error-free by breaking it into smaller tables and linking them through relationships. This process reduces redundancy, improves data integrity, and optimizes database performance. Then why do you need it? If there is no normalization in SQL, there will be many problems, such as:

- Insert Anomaly: This happens when we cannot insert data into the table without another.
- Update Anomaly: This is due to data inconsistency caused by data redundancy and data update.
- Delete exception: Occurs when some attributes are lost due to the deletion of other attributes.

<u>What is Normalization in DBMS?</u>

So, normalization is a way of organizing data in a database. Normalization involves organizing the columns and tables in the database to ensure that their dependencies are correctly implemented using database constraints. Normalization is the process of organizing data properly. It is used to minimize the duplication of various relationships in the database. It is also used to troubleshoot exceptions such as inserts, deletes, and updates in the table. It helps to split a large table into several small normalized tables. Relational links and links are used to reduce redundancy. Normalization, also known as database normalization or data normalization, is an important part of relational database design because it helps to improve the speed, accuracy, and efficiency of the database.

Now the question arises what is the relationship between SQL and normalization? Well, SQL is the language used to interact with the database. Normalization in SQL improves data distribution. To initiate interaction, the data in the database must be normalized. Otherwise, we cannot continue because it will cause an exception. Normalization can also make it easier to design the database to have the best structure for atomic elements (that is, elements that cannot be broken down into smaller parts). Usually, we break large tables into small tables to improve efficiency. Edgar F. Codd defined the first paradigm in 1970, and finally other paradigms. When normalizing a database, organize data into tables and columns. Make sure that each table contains only relevant data. If the data is not directly related, create a new table for that data. Normalization is necessary to ensure that the table only contains data directly related to the primary key, each data field contains only one data element, and to remove redundant (duplicated and unnecessary) data.

The process of refining the structure of a database to minimize redundancy and improve integrity of database is known as Normalization. When a database has been normalized, it is said to be in normal form.

Types of Normalization

Normalization usually occurs in phases where every phase is assigned its equivalent 'Normal form'. As we progress upwards the phases, the data gets more orderly and hence less permissible to redundancy, and more consistent. The commonly used normal forms include:

1. **First Normal Form (1NF):** In the 1NF stage, each column in a table is unique, with no repetition of groups of data. Here, each entry (or tuple) has a unique identifier known as a primary key.

2. **Second Normal Form (2NF):** Building upon 1NF, at this stage, all non- key attributes are fully functionally dependent on the primary key. In other words, the non-key columns in the table should rely entirely on each candidate key.

3. **Third Normal Form (3NF):** This stage takes care of transitive functional dependencies. In the 3NF stage, every non-principal column should be non-transitively dependent on each key within the table.

4. **Boyce-Codd Normal Form (BCNF):** BCNF is the next level of 3NF that guarantees the validity of data dependencies. The dependencies of any attributes on non-key attributes are removed under the third level of normalization . For that reason, it ensures that each determinant be a candidate key and no dependent can fail to possess an independent attribute as its candidate key.

5. **Fourth Normal Form (4NF):** 4NF follows that data redundancy is reduced to another level with the treatment of multi-valued facts. Simply put, the table is in normal form when it does not result in any update anomalies and when a table consists of multiple attributes, each is independent. In other words, it collapses the dependencies into single vs.

multi-valued and eliminates the root of any data redundancy concerned with the multi-valued one.

Need For Normalization

- It eliminates redundant data.
- It reduces chances of data error.
- The normalization is important because it allows database to take up less disk space.
- It also helps in increasing the performance.
- It improves the data integrity and consistency.

Advantages

There are many benefits to normalizing a database. Some of the main advantages are:

- By using normalization redundancy of database or data duplication can be resolved.
- We can minimize null values by using normalization.
- Results in a more compact database (due to less data redundancy/zero).
- Minimize/avoid data modification problems.
- It simplifies the query.is
- The database structure is clearer and easier to understand.
- The database can be expanded without affecting existing data.
- Finding, sorting, and indexing can be faster because the table is small and more rows can be accommodated on the data page.

We can now see that the concepts of denormalization, normalization, and denormalization are technologies used in databases and are differentiable terms. Normalization is a method of minimizing insertion, elimination and update exceptions by eliminating redundant data. The reverse normalization process, which adds redundancy to the data to improve application-specific performance and data integrity.

Parameters	**Normalization**	**Denormalization**
Concept	Normalization is the process of creating a general scheme for storing non-redundant and consistent data.	A process of combining the data so that it can be queried speedily is known as denormalization.
Goal	Reduce data redundancy and inconsistency.	Execute queries faster through introducing redundancy.
Used in	OLTP system and its focus is to speed up the insertion, deletion, and update of abnormalities and the preservation of quality data.	OLAP system, focusing on better search and analysis.
Data integrity	Here data integrity is well maintained.	Here data integrity may not retain.
Redundancy	Normalization eliminated redundancy.	Denormalization added redundancy.

Number of tables	In normalization number of tables increases.	Whereas denormalization decreases tables.
Disk space	Optimized usage of disk space is possible.	Whereas in denormalization optimal use of disk space is not possible.

When to normalize data Normalization is particularly important for OLTP systems, where insert, update and delete operations are fast and are usually initiated by the end-user. On the other hand, normalization is not always seen as important for OLAP systems and data warehouses. Data is usually denormalized to improve the performance of queries that need to be run in that context.

When to denormalize data It is best to denormalize a database in several situations. Many data warehouses and OLAP applications use denormalized databases. The main reason for this is performance. These applications are often used to perform complex queries. Joining many tables usually returns very large records. There may be other reasons for database denormalization, for example, to enforce certain restrictions that may not be enforced.

Here are some common reasons you might want to denormalize your database:

- The most common queries require access to the entire concatenated data set.
- Most applications perform a table scan when joining tables.
- The computational complexity of the derived column requires an overly complex temporary table or query.
- You can implement constraints (based on DBMS) that could not otherwise be achieved

Although normalization is generally considered mandatory for OLTP and other transactional databases, it is not always appropriate for some analytical applications.

Why is Normalization Important?

Normalization is crucial as it helps eliminate redundant data and inconsistencies, ensuring more accurate, lean, and efficient databases. It also simplifies data management and enhances the speed and performance of the overall database system, thereby proving to be advantageous.

Example

Let us assume the library database that maintains the required details of books and borrowers. In an unnormalized database, the library records in one table the book details and the member who borrowed it, as well as the member's detail. This would result in repetitive information every time a member borrows a book.

Normalization splits the data into different tables ‘Books’, “Members” and “Borrowed” and connects “Books” and “Members” with “Borrowed” through a biunique key. This removes redundancy, which means data is well managed, and there is less space utilization.

Conclusion

The concepts of normalization, and the ability to put this theory into practice, are key to building and maintaining comprehensive databases which are both strong and impervious to data anomalies and redundancy. Properly applied and employed at the right times, normalization boosts database quality, making it structured, small, and easily manageable.

Types of Functional dependencies in DBMS

In relational database management, functional dependency is a concept that specifies the relationship between two sets of attributes where one attribute determines the value of another attribute. It is denoted as **X → Y**, where the attribute set on the left side of the arrow, **X** is called **Determinant**, and **Y** is called the **Dependent**.

What is Functional Dependency?

A functional dependency occurs when one attribute uniquely determines another attribute within a relation. It is a constraint that describes how attributes in a table relate to each other. If attribute A functionally determines attribute B, we write this as the **A→B**.

Functional dependencies are used to mathematically express relations among database entities and are very important to understanding advanced concepts in Relational Database Systems.

Example:

roll_no	**name**	**dept_name**	**dept_building**
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

- roll_no → {name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, dept_name and dept_building, hence a valid Functional dependency

- roll_no → dept_name , Since, roll_no can determine whole set of {name, dept_name, dept_building}, it can determine its subset dept_name also.
- dept_name → dept_building , Dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies: roll_no → name, {roll_no, name} ⇢ {dept_name, dept_building}, etc.

Here are some invalid functional dependencies:

- name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name There can be multiple departments in the same building. Example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.
- More invalid functional dependencies: name → roll_no, {name, dept_name} → roll_no, dept_building → roll_no, etc.

Armstrong's axioms/properties of functional dependencies:

1. **Reflexivity:** If Y is a subset of X, then X→Y holds by reflexivity rule
   Example, {roll_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
   Example, {roll_no, name} → dept_building is valid, hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
   Example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

Types of Functional Dependencies in DBMS

1. Trivial functional dependency
2. Non-Trivial functional dependency
3. Multivalued functional dependency
4. Transitive functional dependency

1. Trivial Functional Dependency

In **Trivial Functional Dependency**, a dependent is always a subset of the determinant. i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency

Example:

roll_no	**name**	**age**
42	abc	17
43	pqr	18
44	xyz	18

Here, **{roll_no, name} → name** is a trivial functional dependency, since the dependent **name** is a subset of determinant set **{roll_no, name}.** Similarly, **roll_no → roll_no** is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In **non-trivial functional dependency**, the dependent is strictly not a subset of the determinant. i.e. If **X → Y** and **Y is not a subset of X**, then it is called non-trivial functional dependency.

Example:

roll_no	**name**	**age**
42	abc	17
43	pqr	18
44	xyz	18

Here, **roll_no → name** is a non-trivial functional dependency, since the dependent **name** is **not a subset of** determinant **roll_no.** Similarly,
{roll_no, name} → age is also a non-trivial functional dependency, since
age is **not a subset of {roll_no, name}**

3. Multivalued Functional Dependency

In **Multivalued functional dependency**, entities of the dependent set are **not dependent on each other.** i.e. If **a → {b, c}** and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency.**

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, **roll_no → {name, age}** is a multivalued functional dependency, since the dependents **name** & **age** are **not dependent** on each other (i.e. **name → age** or **age → name doesn't exist!**)

4. Transitive Functional Dependency: -

In transitive functional dependency, dependent is indirectly dependent on determinant. i.e. If **a → b** & **b → c**, then according to axiom of transitivity, **a → c**. This is a **transitive functional dependency.**

For example,

enrol_no	name	dept	building_no
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, **enrol_no → dept** and **dept → building_no.** Hence, according to the axiom of transitivity, **enrol_no → building_no** is a valid functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

5. Fully Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies X->Y and X->Z which states that those dependencies are fully functional.

6. Partial Functional Dependency

In partial functional dependency a non-key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then X->Z is a partial functional dependency in RBDMS.

Advantages of Functional Dependencies

Functional dependencies having numerous applications in the field of database management system. Here are some applications listed below:

1. Data Normalization

Data normalization is the process of organizing data in a database in order to minimize redundancy and increase data integrity. Functional dependencies play an important part in data normalization. With the help of functional dependencies, we are able to identify the primary key, candidate key in a table which in turns helps in normalization.

2. Query Optimization

With the help of functional dependencies, we are able to decide the connectivity between the tables and the necessary attributes need to be projected to retrieve the required data from the tables. This helps in query optimization and improves performance.

3. Consistency of Data

Functional dependencies ensure the consistency of the data by removing any redundancies or inconsistencies that may exist in the data. Functional dependency ensures that the changes made in one attribute does not affect inconsistency in another set of attributes thus it maintains the consistency of the data in database.

4. Data Quality Improvement

Functional dependencies ensure that the data in the database to be accurate, complete and updated. This helps to improve the overall quality of the data, as well as it eliminates errors and inaccuracies that might occur during data analysis and decision making, thus functional dependency helps in improving the quality of data in database.

Conclusion: -

Functional dependency is very important concept in database management system for ensuring the data consistency and accuracy. In this article we have discuss what is the concept behind functional dependencies and why they are important. The valid and invalid functional dependencies and the types of most important functional dependencies in RDBMS. We have also discussed about the advantages of FDs.

Lossless Join and Dependency Preserving Decomposition

Decomposition of a relation is done when a relation in a relational model is not in appropriate normal form. Relation R is decomposed into two or more relations if decomposition is lossless join as well as dependency preserving.

Lossless Join Decomposition

If we decompose a relation R into relations R1 and R2,

Decomposition is lossy if R1 ⋈ R2 ⊃ R Decomposition is lossless if R1 ⋈ R2 = R

To check for lossless join decomposition using the FD set, the following conditions must hold:

1. The Union of Attributes of R1 and R2 must be equal to the attribute of R. Each attribute of R must be either in R1 or in R2.

Att(R1) U Att(R2) = Att(R)

2. The intersection of Attributes of R1 and R2 must not be NULL.

Att(R1) ∩ Att(R2) ≠ Φ

3. The common attribute must be a key for at least one relation (R1 or R2)

Att(R1) ∩ Att(R2) -> Att(R1) or Att(R1) ∩ Att(R2) -> Att(R2)

For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is a lossless join decomposition as:

1. First condition holds true as Att(R1) U Att(R2) = (ABC) U (AD) = (ABCD) = Att(R).
2. Second condition holds true as Att(R1) ∩ Att(R2) = (ABC) ∩ (AD) ≠ Φ
3. The third condition holds as Att(R1) ∩ Att(R2) = A is a key of R1(ABC) because A->BC is given.

Dependency Preserving Decomposition

If we decompose a relation R into relations R1 and R2, all dependencies of R either must be a part of R1 or R2 or must be derivable from a combination of functional dependency of R1 and R2. For Example, A relation R (A, B, C, D) with FD set{A->BC} is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of R1(ABC).

Advantages of Lossless Join and Dependency Preserving Decomposition

- **Improved Data Integrity:** Lossless join and dependency preserving decomposition help to maintain the data integrity of the original relation by ensuring that all dependencies are preserved.
- **Reduced Data Redundancy:** These techniques help to reduce data redundancy by breaking down a relation into smaller, more manageable relations.
- **Improved Query Performance:** By breaking down a relation into smaller, more focused relations, query performance can be improved.
- **Easier Maintenance and Updates:** The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.
- **Better Flexibility:** Lossless join and dependency preserving decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

Disadvantages of Lossless Join and Dependency Preserving Decomposition

- **Increased Complexity:** Lossless join and dependency- preserving decomposition can increase the complexity of the database system, making it harder to understand and manage.
- **Costly:** Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.
- **Reduced Performance:** Although query performance can be improved in some cases, in others, lossless join and dependency- preserving decomposition can result in reduced query performance due to the need for additional join operations.
- **Limited Scalability:** These techniques may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

Schema Refinement: -

The schema Refinement refers to refine the schema by using Some technique.
Normalization: - Normalization means split the tables. into small tables which will contain less number of attributes. Normalization or schema Refinement in a technique of organizing the data in the database It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like insertion, Update and deletion Anomalies.

Types of Normalization: -

Example: -

ID	Name	Course	Skills	C-Fees
501	A	CSE	C, C++	80k
502	B	CSM	C, python	60k
503	C	CSM	C, java	60k

1. 1NF
2. 2NF
3. 3NF
4. BCNF
5. 4NF
6. 5NF

1NF [First Normal Form]: - A relation is said to in the 1NF if it is already in un- normalized form.

* Each attribute name must be unique
* Each attribute Value must be single or atomic i.e., Single Value attribute

Example: -

ID	Name	Course	Skills	C-Fees
501	A	CSE	C	80k
501	A	CSE	C++	80k
502	B	CSM	C	60k
502	B	CSM	python	60k
503	C	CSM	C	60k
503	C	CSM	java	60k

2NF: - A relation is said to be in 2NF. If it satisfies 1Nf and it has no partial dependency. [primary key should be unique].

Ex: -

id	name	Course	C-fee
501	A	CSE	80k
502	B	CSM	60k
503	C	CSM	60k

3NF: - A database is in 3NF if it satisfies 2NF and there is no transitive functional dependency.

* A non- key attribute is depending on a non- key attribute.

id	name	course
501	A	CSE
502	B	CSM
503	C	CSM

course	C-fee
CSE	80k
CSM	60k
CSM	60k

4NF: - A relation is said to be in 4NF if it is in Boyce Codd normal form and should have no multi. Valued dependency

id	name
501	A
502	B
503	C

id	C-fee
501	80k
502	60k
503	60k

5NF: -

* It is a database normalization technique that ensures data consistency and reduces data de redundancy. It is an extension of the fourth Normal form (4NF) and is considered a stronger normal form.

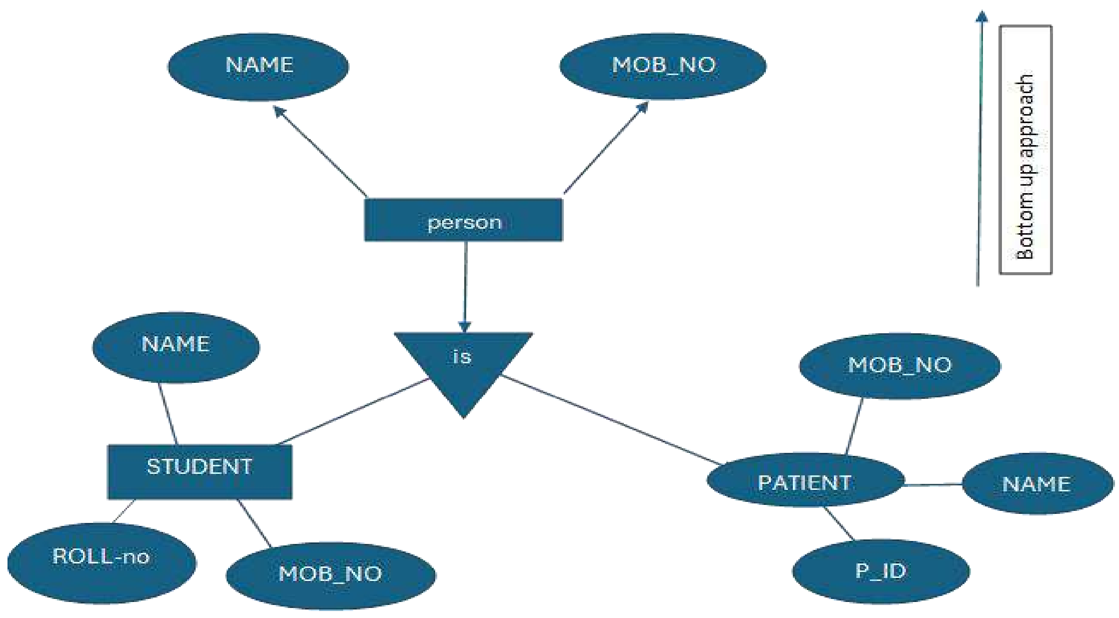
Faculty	year
A	2024
B	2024

Faculty	Branch
A	CSE
B	AI&DS

Branch	year
CSE	2024
AI&DS	2024

Generalization: -

It works on the principle of bottom-up approach. In Generalization lower-level functions are combined to form highest level function which is called as entities.
In generalization process properties are drawn from particular entities as it combines subclasses to form superclass



Specialization: -

Specialization is opposite of Generalization. In specialization things a broken down into smaller things to simplify it further. We can say that in specialization a particular entity gets divided into sub entities. Also, in specialization Inheritance takes place.

Surrogate Key in DBMS

A key is a column, or group of columns, in a database management system (DBMS) that uniquely identifies every row in a table. Natural keys and surrogate keys are the two categories of keys.

- **Natural Key:** A column, or group of columns, that is generated from the table's data is known as a natural key. For instance, since it uniquely identifies every client in the table, the customer ID column in a customer table serves as a natural key.
- **Surrogate key**: A column that is not generated from the data in the database is known as a surrogate key. Rather, the DBMS generates a unique identifier for you. In database tables, surrogate keys are frequently utilized as primary keys.

Surrogate Key

A surrogate key also called a synthetic primary key, is generated when a new record is inserted into a table automatically by a database that can be declared as the primary key of that table. It is the sequential number outside of the database that is made available to the user and the application or it acts as an object that is present in the database but is not visible to the user or application.

We can say that, in case we do not have a natural primary key in a table, then we need to artificially create one in order to uniquely identify a row in the table, this key is called the surrogate key or synthetic primary key of the table. However, the surrogate key is not always the primary key. Suppose we have multiple objects in a database that are connected to the surrogate key, then we will have a many-to-one association between the primary keys and the surrogate key and the surrogate key cannot be used as the primary key.

Features of the Surrogate Key

- It is automatically generated by the system.
- It holds an anonymous integer.
- It contains a unique value for all records of the table.
- The value can never be modified by the user or application.

- The surrogate key is called the fact less key as it is added just for our ease of identification of unique values and contains no relevant fact (or information) that is useful for the table.

Consider an example: Suppose we have two tables of two different schools having the same column registration_no, name, and percentage, each table having its own natural primary key, that is registration_no.

Table of school A:

registration_no	**name**	**percentage**
210101	Harry	90
210102	Maxwell	65
210103	Lee	87
210104	Chris	76

Table of school B:

registration_no	**name**	**percentage**
CS107	Taylor	49
CS108	Simon	86
CS109	Sam	96
CS110	Andy	58

Now, suppose we want to merge the details of both the schools in a single table.
Resulting table will be:

surr_no	**registration_no**	**name**	**percentage**
1	210101	Harry	90
2	210102	Maxwell	65
3	210103	Lee	87
4	210104	Chris	76
5	CS107	Taylor	49
6	CS108	Simon	86
7	CS109	Sam	96
8	CS110	Andy	58

As we can observe the above table and see that registration_no cannot be the primary key of the table as it does not match with all the records of the table though it is holding all unique values of the table. Now, in this case, we have to artificially create one primary key for this table. We can do this by adding a column surr_no in the table that contains anonymous integers and has no direct relation with other columns. This additional column of surr_no is the surrogate key of the table.

Why use Surrogate Key in DBMS?

There are several reasons to use surrogate keys in database tables:

1. **Uniqueness:** Data integrity is improved by the guaranteed uniqueness of surrogate keys.
2. **Stability:** Since surrogate keys do not depend on any business rules or data value, they have a lower chance of changing over time.
3. **Efficiency:** Compared to natural keys, surrogate keys are frequently smaller and process more quickly.
4. **Flexibility:** In the event that the natural key changes, rows can still be uniquely identified using surrogate keys.

Advantages of the Surrogate Key

- As there is no direct information related with the table, so the changes are only based on the requirements of the application.
- Performance is enhanced as the value of the key is relatively smaller.
- The key value is guaranteed to contain unique information.
- As it holds smaller constant values, this makes integration of the table easy.
- Enables us to run fast queries (as compared to the natural primary key)

Disadvantages of the Surrogate Key

- The surrogate key value can never be used as a search key.
- As the key value has no relation to the data of the table, so third normal form is violated.
- The extra column for surrogate key will require extra disk space.
- We will need extra IO when we have to insert or update data of the table.

Examples of Surrogate Key

- System date & time stamp
- Random alphanumeric string

Conclusion

Surrogate keys are an important tool for designing and implementing databases. They can be applied to enhance database systems, flexibility, stability, and performance.

Boyce-Codd Normal Form (BCNF)

Although, 3NF is an adequate normal form for relational databases, this (3NF) normal form may not remove 100% redundancy because of X−>Y functional dependency if X is not a candidate key of the given relation. This can be solved by the Boyce-Codd Normal Form (BCNF).

Application of the general definitions of 2NF and 3NF may identify additional redundancy caused by dependencies that violate one or more candidate keys. However, despite these additional constraints, dependencies can still exist that will cause redundancy in 3NF relations. This weakness in 3NF resulted in the presentation of a stronger normal form called the **Boyce-Codd Normal Form (Codd, 1974)**.

Boyce-Codd Normal Form (BCNF)

Boyce-Codd Normal Form (BCNF) An advanced version of the third normal form, which is a little more strict. This is a tougher criterion that helps eliminate redundancy and anomalies from your Database. BCNF (Boyce-Codd Normal Form) is a further refinement of the third normal form which simplifies our database design. More precisely, this enforces that every non-trivial functional dependency is determined by superkeys. When is a relation in BCNF: A formula for FD is present, where the left- hand side is super key this solves potential problems with candidate keys as well. Why it enforces no redundancy more strongly than 3NF?

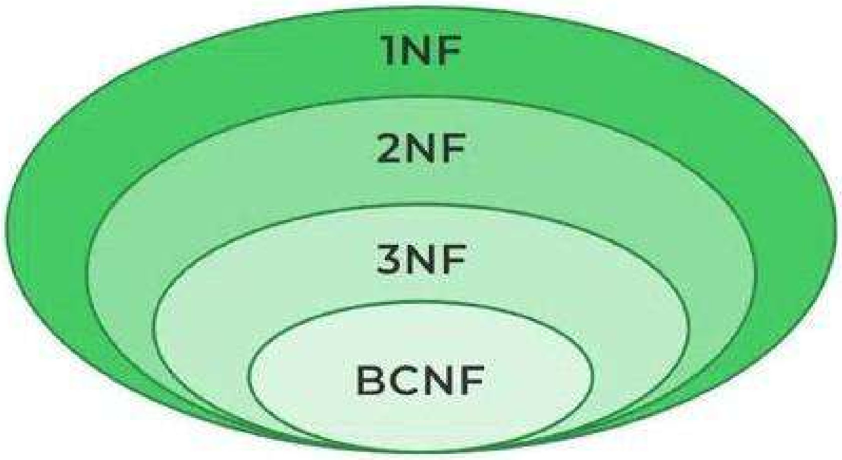
BCNF is essential for good database schema design in higher-level systems where consistency and efficiency are important, particularly when there are many candidate keys (as one often finds with a delivery system).

Rules for BCNF

Rule 1: *The table should be in the 3rd Normal Form.*

Rule 2: *X should be a super key for every functional dependency (FD) X−>Y in a given relation.*

Note: To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys.



You came across a similar hierarchy known as the **Chomsky Normal Form** in the Theory of Computation. Now, carefully study the hierarchy above. It can be inferred that **every relation in BCNF is also in 3NF**. To put it another way, a relation in 3NF need not be in BCNF. Ponder over this statement for a while.

To determine the highest normal form of a given relation R with functional dependencies, the first step is to check whether the BCNF condition holds. If R is found to be in BCNF, it can be safely deduced that the relation is also in 3NF, 2NF, and 1NF as the hierarchy shows. The 1NF has the least restrictive constraint – it only requires a relation R to have atomic values in each tuple. The 2NF has a slightly more restrictive constraint.

The 3NF has a more restrictive constraint than the first two normal forms but is less restrictive than the BCNF. In this manner, the restriction increases as we traverse down the hierarchy.

Examples

Here, we are going to discuss some basic examples which let you understand the properties of BCNF. We will discuss multiple examples here.

Example 1

Let us consider the student database, in which data of the student are mentioned.

Stu_ID	**Stu_Branch**	**Stu_Course**	**Branch_Number**	**Stu_Course_No**
101	Computer Science & Engineering	DBMS	B_001	201
101	Computer Science & Engineering	Computer Networks	B_001	202
102	Electronics & Communication Engineering	VLSI Technology	B_003	401
102	Electronics & Communication Engineering	Mobile Communication	B_003	402

Functional Dependency of the above is as mentioned:

Stu_ID –> Stu_Branch
Stu_Course –> {Branch_Number, Stu_Course_No}

Candidate Keys of the above table are: **{Stu_ID, Stu_Course}**

Why this Table is Not in BCNF?

The table present above is not in BCNF, because as we can see that neither Stu_ID nor Stu_Course is a Super Key. As the rules mentioned above clearly tell that for a table to be in BCNF, it must follow the property that for functional dependency X–>Y, X must be in Super Key and here this property fails, that's why this table is not in BCNF.

How to Satisfy BCNF?

For satisfying this table in BCNF, we have to decompose it into further tables. Here is the full procedure through which we transform this table into BCNF. Let us first divide this main table into two tables **Stu_Branch** and **Stu_Course** Table.

Stu_Branch Table

Stu_ID	**Stu_Branch**
101	Computer Science & Engineering
102	Electronics & Communication Engineering

Candidate Key for this table: **Stu_ID**.

Stu_Course Table

Stu_Course	**Branch_Number**	**Stu_Course_No**
DBMS	B_001	201
Computer Networks	B_001	202
VLSI Technology	B_003	401
Mobile Communication	B_003	402

Candidate Key for this table: **Stu_Course**.

Stu_ID to Stu_Course_No Table

Stu_ID	**Stu_Course_No**
101	201
101	202
102	401
102	402

Candidate Key for this table: **{Stu_ID, Stu_Course_No}.**

After decomposing into further tables, now it is in BCNF, as it is passing the condition of Super Key, that in functional dependency X−>Y, X is a **Super Key.**

Example 2

Find the highest normal form of a relation R (A, B, C, D, E) with FD set as:

{BC->D, AC->BE, B->E}

Explanation:

- **Step-1:** As we can see, (AC)+ = {A, C, B, E, D} but none of its subsets can determine all attributes of the relation, So AC will be the candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
- **Step-2:** Prime attributes are those attributes that are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example.
- **Step-3:** The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attributes.

The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).

The relation is **not** in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be a prime attribute. So the highest normal form of relation will be the 2nd Normal form.

Note: A prime attribute cannot be transitively dependent on a key in BCNF relation.

Consider these functional dependencies of some relation R

AB ->C C
->B AB -
>B

From the above functional dependency, we get that the candidate key of R is AB and AC. A careful observation is required to conclude that the above dependency is a Transitive Dependency as the prime attribute B transitively depends on the key AB through C. Now, the first and the third FD are in BCNF as they both contain the candidate key (or simply KEY) on their left sides. The second dependency, however, is not in BCNF but is definitely in 3NF due to the presence of the prime attribute on the right side. So, the highest normal form of R is 3NF as all three FDs satisfy the necessary conditions to be in 3NF.

Example 3

For example, consider relation R (A, B, C)

A -> BC,
B -> A

A and B both are super keys so the above relation is in BCNF.

Note: BCNF decomposition may always not be possible with dependency preserving, however, it always satisfies the lossless join condition. For example, relation R (V, W, X, Y, Z), with functional dependencies:

V, W -> X
Y, Z -> X W
-> Y

It would not satisfy dependency preserving BCNF decomposition.

Note: Redundancies are sometimes still present in a BCNF relation as it is not always possible to eliminate them completely.

There are also some higher-order normal forms, like the 4th Normal Form and the 5th Normal Form.

For more, refer to the 4th and 5th Normal Forms.

Conclusion: -
In conclusion, we can say that Boyce-Codd Normal Form (BCNF) is very much essential as far as database normalization are concerned which help us in doing normalizing beyond the limits of 3NF. By making sure all functional dependencies depend on super keys, this is how BCNF helps us avoid redundancy and update anomalies. This makes the BCNF a highly desirable property and helps in achieving Data Integrity which is number one concern for any Database Designer.

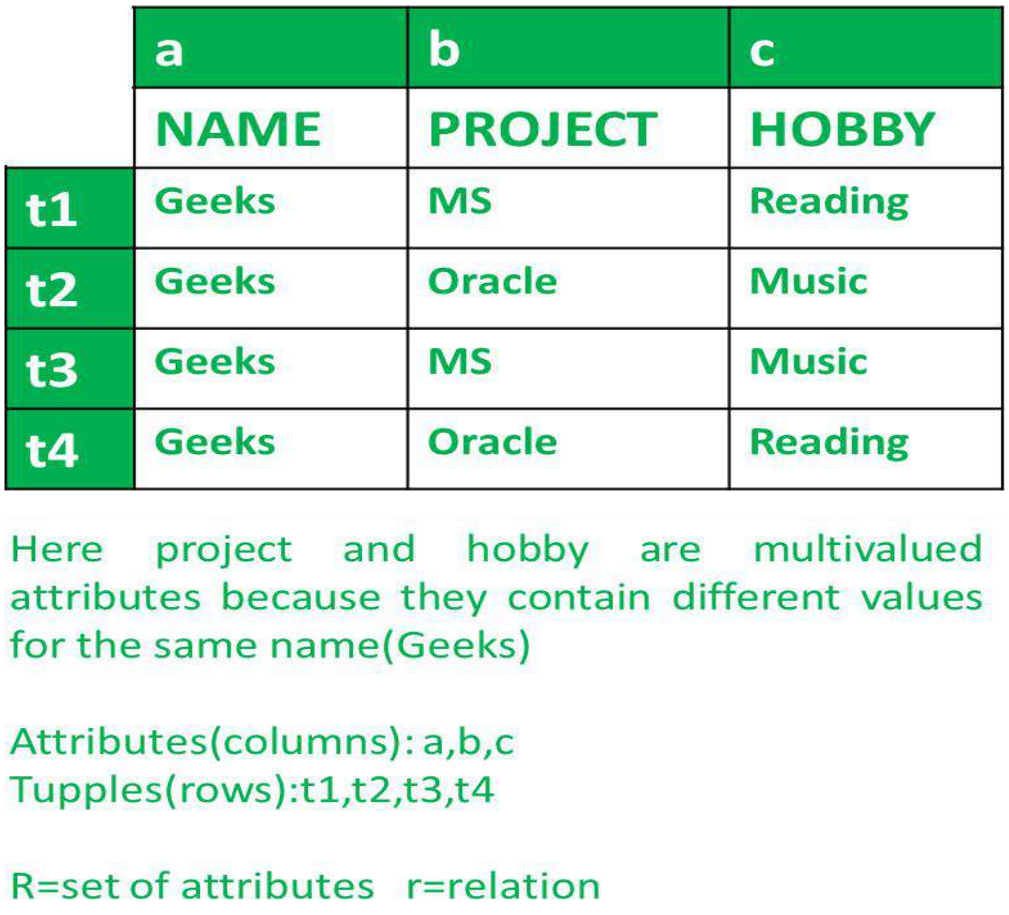
Multivalued Dependency (MVD) in DBMS

In **Database Management Systems (DBMS),** multivalued dependency (MVD) deals with complex attribute relationships in which an attribute may have many independent values while yet depending on another attribute or group of attributes. It improves database structure and consistency and is essential for data integrity and database normalization.

MVD or multivalued dependency means that for a single value of attribute ‘a’ multiple values of attribute ‘b’ exist. We write it as,

a --> --> b

It is read as a is multi-valued dependent on b. Suppose a person named Geeks is working on 2 projects Microsoft and Oracle and has 2 hobbies namely Reading and Music. This can be expressed in a tabular format in the following way.



	a	b	c
	NAME	PROJECT	HOBBY
t1	Geeks	MS	Reading
t2	Geeks	Oracle	Music
t3	Geeks	MS	Music
t4	Geeks	Oracle	Reading

Here project and hobby are multivalued attributes because they contain different values for the same name(Geeks)

Attributes(columns): a,b,c
Tupples(rows):t1,t2,t3,t4

R=set of attributes r=relation

Project and Hobby are multivalued attributes as they have more than one value for a single person i.e., Geeks.

<u>What is Multivalued Dependency?</u>

When one attribute in a database depends on another attribute and has many independent values, it is said to have multivalued dependency (MVD). It supports maintaining data accuracy and managing intricate data interactions.

Multi Valued Dependency (MVD)

We can say that multivalued dependency exists if the following conditions are met.

Conditions for MVD

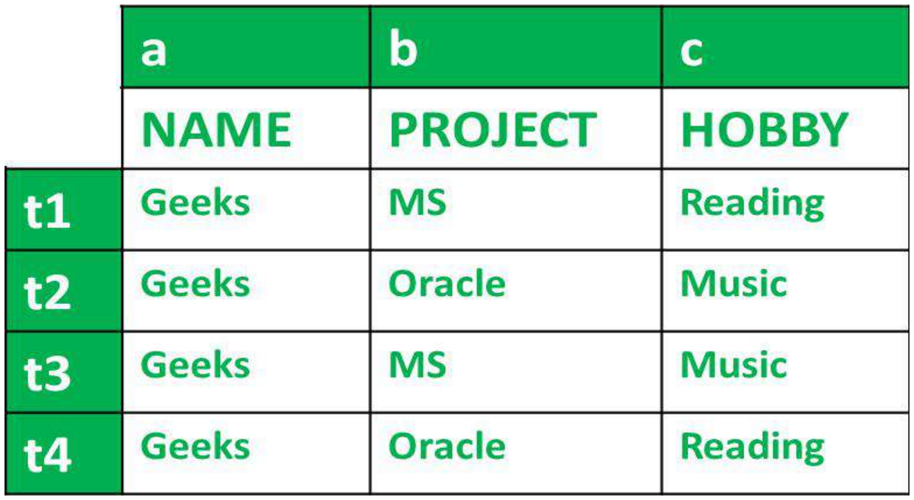
Any attribute says **a** multiple define another attribute b; if any legal relation r(R), for all pairs of tuples t1 and t2 in r, such that,

t1[a] = t2[a]

Then there exists t3 and t4 in r such that.

t1[a] = t2[a] = t3[a] = t4[a]
t1[b] = t3[b]; t2[b] = t4[b] t1 = t4; t2 =
t3

Then multivalued (MVD) dependency exists. To check the MVD in given table, we apply the conditions stated above and we check it with the values in the given table.



	a	b	c
	NAME	PROJECT	HOBBY
t1	Geeks	MS	Reading
t2	Geeks	Oracle	Music
t3	Geeks	MS	Music
t4	Geeks	Oracle	Reading

Condition-1 for MVD
t1[a] = t2[a] = t3[a] = t4[a]

Finding from table,

t1[a] = t2[a] = t3[a] = t4[a] = Geeks

So, condition 1 is Satisfied.

Condition-2 for MVD
t1[b] = t3[b]
And
t2[b] = t4[b]

Finding from table,

t1[b] = t3[b] = MS
And
t2[b] = t4[b] = Oracle

So, condition 2 is Satisfied.

Condition-3 for MVD

∃c ∈ R- (a ∪ b) where R is the set of attributes in the relational table.
t1 = t4
And **t2=t3**

Finding from table,

t1 = t4 = Reading
And
t2 = t3 = Music

So, condition 3 is Satisfied. All conditions are satisfied, therefore,

a --> --> b

According to table we have got,

name --> --> project

And for,

a --> --> C
We get,

name --> --> hobby

Hence, we know that MVD exists in the above table and it can be stated by,

name --> --> project name --> -->
hobby

Conclusion

- Multivalued Dependency (MVD) is a form of data dependency where two or more attributes, other than the key attribute, are functionally independent on each other, but these attributes depend on the key.
- Data errors and redundancies may result from Multivalued Dependency.
- We can normalize the database to 4NF in order to get rid of Multivalued Dependency.

4NF: - A relation is said to be in 4NF if it is in Boyce Codd normal form and should have no multi. Valued dependency

id	name
501	A
502	B
503	C

id	C-fee
501	80k
502	60k
503	60k

5NF: -

* It is a database normalization technique that ensures data consistency and reduces data de redundancy. It is an extension of the fourth Normal form (4NF) and is considered a stronger normal form.

Faculty	year
A	2024
B	2024

Faculty	Branch
A	CSE
B	AI&DS

Branch	year
CSE	2024
AI&DS	2024

UNIT 5 – TRANSACTION CONCEPT

Transaction:-collection of operations that from a single logical unit of work are called transactions.

Example of a transaction:-

Consider a bank transfer where 100/- is transferred from account A to Account B. The transaction consists of two operation.

1. Deduct 100/- from Account A
2. Add 100/- to Account B

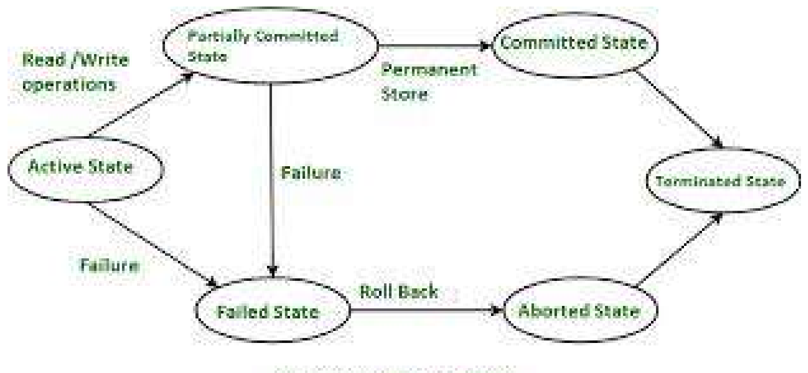
This transaction is atomic; either both operations are complete successfully or neither occurs. If the first operation (deducting 100/-) succeeds but the second (adding to Account B) fails, the system will roll back the transaction to ensure consistency.

Transaction life cycle:-

- Begin Transaction:-The transaction starts.
- Execution of Operation:-Various read/write operations are performed.
- Commit:-If all operations are successful, the transaction is committed, and the changes are saved permanently.
- Rollback:-If any error occurs, the transaction is rolled back, and the change made during the transaction all undone.

Transaction State:-

In DBMS, a transaction is a sequence of one or more operations performed on a database. These operations must follow the ACID properties to ensure the database remains in a valid state. A transaction can be in one of several states during its life cycle, which are described below:



Transaction States in DBMS

1. **Active state:-**
   - ➢ The transaction is in the active state when it is being executed.
   - ➢ Operations like reading and writing data are performed during this phase.
   - ➢ It can remain in the state until the transaction is complete or aborted.
2. **Partially Committed State:-**
   - ➢ After the transaction has executed its final operation (e.g.., an update or insert), it enters the partially committed state.
   - ➢ At this point, the DBMS must ensure that all changes mode by the transaction are saved permanent. However the changes are not yet mode permanent in the database until. Successful completion (commit).
3. **Failed state:-**
   - ➢ If a transaction fails during execution (due to an error system crash), it enters the failed state.
   - ➢ In this state, any changes mode to the database by the transaction are rolled back (undone).
4. **Aborted State:-**
   - ➢ If a transaction cannot proceed due to failure, it is rolled back and all the changes mode by the transaction are undone.
   - ➢ After rolled, the transaction can either be restarted or completely terminated.
5. **Committed State:**
   - ➢ If a transaction completes all its operations successfully and the database change are saved permanently, it enters the committed state.
   - ➢ This is the final state, where the transaction's changes become visible to other transactions.

- After the system enter the aborted state. At this point, the system has two options:
  - ➢ It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
  - ➢ It can kill the transaction. It usually does so because of some internal logical error that can be corrected
  - ➢ Only by rewriting the application programme, or because the input was bad, or because the during data were not found in the database

Example:-

We have two Accounts in a bank:

- o Account A has a balance of 500/-
- o Account B has a balance of 300/-

The goal is to transfer 100/- from Account A to B.

Transaction states:-

1.Active state:-

- The transaction begins with a series of operation.
- First, the system read the balance of Account A to check if there are sufficient funds.
- Next, it deducts 100/- from Account A.

- Then, the system reads the balance of Account B and prepares to add 100/- to Account B.

<u>Operations during Active state:-</u>

- Read balance of Account A:500/-
- Deducted 100/-from Account A:new balance of Account A:400/-
- Read balance of Account B:300/-
- Prepare to add 100/- to Account B (but not bone yet).

2. <u>Partially committed state</u>:-

- Once all operations have been executed successfully, the transaction enters the partially committed state.
- At this point, the system has successfully deducted 100/- from Account A and is about to add 100/- to account B.
- However, the transaction has not been successfully committed to db. The system performs final checks, such as ensuring no errors occurred during execution.

<u>Current status:-</u>

- Balance of Account A is 400/- (deducted 100/-)
- Balance of Account B is about to be updated to 400/- (adding 100/-)
- N0 error so far, but changes have not been permanently saved

<u>3.Committed state:-</u>

- If everything goes smoothly, the transaction moves into the committed state.
- Now, the 100/- has been permanently added to account B, making its new balance 400/-
- The changes to both account A&B are written to the database, and the transaction is now complete.

<u>Final status after commit:-</u>

- Account A:400/-
- Account B:400/-
- The transaction is fully complete, and all changes are durable (i.e.., they will survive after a crash).

<u>4.Failed state:-</u>

- Let's assume that after the system deducts 100/- from Account A but before it adds 100/- to Account B, a power failure occurs or a system crash happens.
- The transaction cannot proceed, so it enters the failed state.

Since part of the transaction completed (the deduction from Account A), but it was not fully committed, the system recognizes that this failure requires recovery to rollback.

<u>5.Aborted state:-</u>

- Since the transaction has failed, the DBMS rolls back the entire transaction, undoing all changes models during the transaction to maintain the database consistency.

- In this case 100/- is restored to account A, So its balance returns to 500/- , and Account B remains at 300/-
- The transaction then enters the aborted state. The DBMS ensures that no partial updates remain in the database, and everything is restored to its original state.

Final status after abort

- Account A: 500/-(rolled back to original state)
- Account B: 300/-(unchanged)

Possible restart or kill:-

Restart:-If the failure was due to a temporary issue (like a deadlock), the dbms could restart the transaction. The transaction would again try to deduct 100/- from Account A and add it to Account B.
Kill:-If the transaction fails repeatedly due to logic error, the dbms could permanently abort (kill) the transaction and not and restart it.

ACID Properties:-

In DBMS ACID properties are a set of four essential principles.

The acronym ACID stands for A- Atomicity
C- Consistency
I – Isolation
D- Durability

1. **Atomicity:-**Atomicity ensures that a transaction is treated as a single invisible unit of work. It means that either all the operations within the transaction are completed successfully or none of them completed.
   - If any part of the transaction fails, the entire transaction is rolled back, and no changes are mode to the db.
2. **Consistency:-**Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database must always remain in a valid state.
   - After the transaction is completed the database must satisfy all integrity rules and constraints. If a transaction validates consistency, it is aborted.
3. **Isolation:-**Isolation ensures that the operations of a transaction are hidden from other concurrent transactions. Even if multiple transaction are executed concurrently, each transaction must behave as if it were running in isolation (i.e..., no interference of other transactions).
   - This prevents “dirty reads”, “uncommitted data read”, or any inconsistencies caused by simultaneous transaction execution.

4. **Durability:-**Durability ensures that once a transaction has been committed, its changes are permanent. Even if the system crashes after a transaction commits, the committed data will not be lost & will remain in the database.
   - ➢ This is typically achieved by writing the changes to non-volatile storage, such as disk or SSD, so that the data can be recovered even in case of system failure.

✓ Transaction access data using two operations

1. Read (x)
2. Write(x)

Read(x):-It transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

Write(x):-It transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Example:-

Account	Balance
A	1000
B	500

Let Ti be a transaction that transfer 200/- from account A to B. This transaction can be defined as:

```
Ti : read (A)
  A=A-200;
Write (A)
Read (B)
  B=B+200;
Write (B)
```

1.Atomicity: During the transfer process, the system first deducts 200/- from Account A.

- If the system crashes before adding the 200/- to Account B, Atomicity guarantees that the entire transaction will be rolled back, meaning the 200/- will be added back to Account A and balance will still be 1000/-

Final balance (after failure):

Account	Balance
A	1000
B	500

Without Atomicity, the system could end up in an inconsistent state where Account A is missing 200/-, but Account B has not received the amount.

2.Consistency: Suppose the banking system enforces a rules that an account cannot have a negative balance. Before the transaction, Account A has 1000/- and deducting 200/- still leaves it with 800/-, which is valid.

- After the transaction, the total sum of balances should be preserved (i.e.., the sum of balances in Account A&B should still be 1500/-).

Final balances (if the transaction completes):

Accounts	Balance
A	800
B	700

The database remain consistent because the sum of the two account balances before and after the transaction is the same (1500/-), and no integrity constraints (like negative balances) are violated.

3.Isolation: while the transaction is deducting 200/- from Account A and adding it to Account B, another transaction tries to check the balance of Account A

➲ Isolation is enforced, the second transaction (Checking the balance of Account A) will not see the intermediate state where 200/- has been deducted from Account A but not yet added to Account B. It will only see the balance as 1000/- (before the transaction) or 800/-(After the transaction), but not the immediate state.

Without isolation:

If isolation is not properly maintained, the second transaction might see an inconsistent state where Account A has 800/-and Account B still has 500/-, leading to an incorrect understanding of the account balances.

4.Durability:-Once the transaction has successfully committed (i.e..,200/- has been deducted from Account A and added to Account B), the change must be permanently stored in the database.

➲ Even if the system crashed immediately after the commit, the change will persist, and when the system recovers, the balance of account A&B will still reflect the successfully transfer.

Final balance (after the commit & system crash):

Account	Balance
A	800
B	700

If Durability is not guaranteed, the changes might be lost after a system crash, leading to inconsistencies in the database.

Concurrent Execution:-

Concurrent Execution transaction means multiple transaction executed/run concurrently in DBMS with each transaction doing its atomic unit of work.

❖ Helps in reducing waiting time

- ❖ Improved resource utilization

Problem of Concurrent Execution

1. **Lost update problem(write-write conflict)**
   When two transactions simultaneously read and update the same data, one update might be lost if one transaction overwrites the changes made by the other.

Example:-

- ➲ (A=500)

T1	T2
Read(A)	
A=A-100	Read(A)
	A=A-200
W(A)	
	W(A)

Let's assume we have an balance of 500/- for Account A. Two transaction T1&T2 are executed concurrently. Although two withdraw is were made, the final balance will be incorrectly updated to 300/-instead of 200/-. In this case the update made by T1 is lost because T2 overwrote it. This is called lost update problem.

2.Dirty read(write-read conflict)

A transaction might read uncommitted data from another transaction, which can later be rolled back, leading to inconsistent results.

Example:-

- ➲ (A=500)

T1	T2
Read(A)	
A=A+100	
Write(A)	Read(A)

Let's assume the T1 is modifying Account A from 500/- to 600/- But it has not committed yet. At the sometime, T2 reads this uncommitted balance, which is 600/- (though the transaction is uncommitted). T1 encounters an error and rolls back the transaction, restoring the balance of Account A to 500/-.this is called dirty read.

3.unrepeatable read(Read-write conflict)

A transaction reads the same data twice but gets different results because another transaction has modified the data in between.

Example:-

➲ (A=1000)

T1	T2
↙ Read(A)	
1000	Read(A)
	A=A+200
	Write(A)
↙ Read(A)	
300	

T1 reads the balance of account A, which is 1000/-. T2 modifies the balance of Account A by adding 2000/-, making the new balance 3000/-, this transaction commits. T1 re-reads the balance of Account A and new sees 3000/-. This is called unrepeatable read.

4.Incorrect Summary read(write-write conflict):

This problem occurs when a transaction computer an aggregate function over a set data while other transaction are updating the data simultaneously.

Example:-

_T1	T2
	Sum=0
	R(A)
	Sum=Sum+ A
	R(y)
	Sum=Sum+ y
R(A)	
Y=y+100	
W(y)	

Here as we didn't commit the t2 transaction we have to apply the sum operation again in t1 to get the correct value which is 300/-. This is called incorrect summary read.

Serializability:-

Serializability is a key concept in dbms that ensures the correctness of transaction executed simultaneously. When multiple users or applications perform transaction concurrently on database, the system must ensure that the integrity & consistency of the data are maintained.

Example:-

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)

This schedule is said to be Serializable because it is executing simultaneously (i.e.., first T1 and T2).

Types of Serializability:-

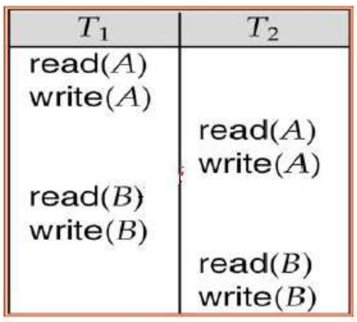
1. Conflict serializability
2. View serializability

1. **Conflict serializability:-**

Two transaction are said to be conflict-serializable if their execution can be rearranged by swapping non-conflicting operations to from a serial schedule without affecting the final result. Two operations conflict if:

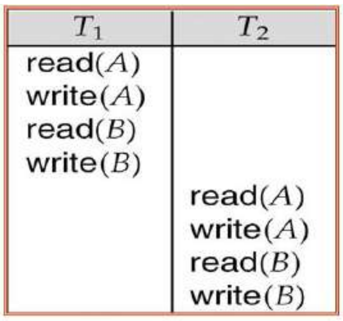
- ❖ Belong different transaction
- ❖ Access the some data item and
- ❖ At least one of them is a write operation

Example:



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Before swapping



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

After swapping

Hence s1 is a serial schedule that we got after swapping non- conflicting operations.

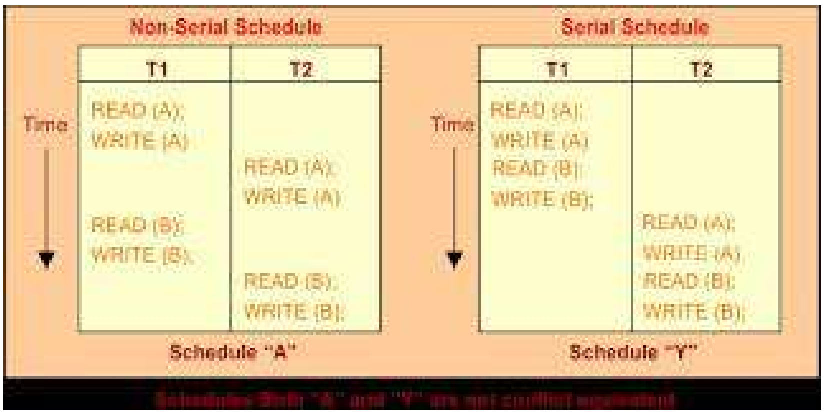
2. <u>**View serializability:-**</u>

A schedule is view serializable if it produces the same outcome as a serial schedule, even though the order of the operations might differ. Two schedules S1 a non-serial schedule and S2 a serial schedule are said to be view equivalent, if they satisfy all the following conditions.

I.e., if in both the schedules S1 & S2, the transaction that performs

- ♣ Initial read
- ♣ Final write and
- ♣ Update read on each data item are same.

Example:-



Non-Serial Schedule

T1	T2
READ (A);	
WRITE (A)	
	READ (A);
	WRITE (A)
READ (B);	
WRITE (B);	
	READ (B);
	WRITE (B);

Schedule "A"

Serial Schedule

T1	T2
READ (A);	
WRITE (A)	
READ (B)	
WRITE (B);	
	READ (A);
	WRITE (A)
	READ (B);
	WRITE (B);

Schedule "Y"

Schedules Both "A" and "Y" are not conflict equivalent

S2 is the serial schedule of S1.If we can prove that they are view equivalent then we can say that given schedule S1 is view serializable.

Let's check the three condition of view serializability.

Initial read

In schedule S1, T1 first reads the data item X. In S2 also T1 first reads the data item X. Let's check for y, In S1, t1 first reads the data item Y. In S2 also, t1 first read the data item y. In S2 also, t1 first reads the data item y.

We checked for both data items x &y and. the initial read condition is satisfied in S1 & S2.

Final write

In schedule S1, the final write operation on x is done by T2. In S2 also T2 performs the final write on X.

Let's check for y. In S1, the final write operation on y is done by T2. In S2 final write on y is done by T2.

We checked for bot data items x & y and the final write condition is satisfied in S1 & S2.

Update read

- In S1, transaction T2 reads the value of x, written by T1.In s2, the same transaction T2 reads the x after it is written by T1.

- In S1, transaction T2 reads the value of y, written by T1.In S2, the same transaction T2 reads the value of y after it is updated by T1.
- The update read condition is also satisfied for both the schedules.

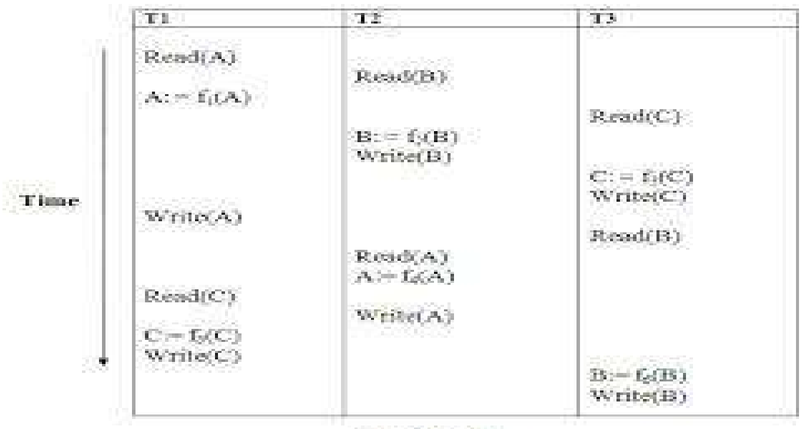
Therefore, since all three condition that checks whether the two means S1, and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

<u>Testing for serializability:-</u>

Serializability process is used to ensure whether the given schedule is serializable or not.

- A precedence graph is used to the test the serializability of a non -serial schedule.
- Precedence graph is a directed graph consist of the pair G=(V,E)
  Where V=all the transaction n in the schedule and
  E=set of edges Ti → Tj if Ti first perform a conflict operation before Tj perform. For which one of three conditions holds:
  1. Ti executes write (Q) before Tj executes read (Q).
  2. Ti executes reads (Q) before Tj executes write (Q).
  3. Ti executes write (Q) before Tj executes write (Q).
- If precedence graph contains no cycle /acyclic that means
  Schedule's' is serializable schedule.

Example:-



T1	T2	T3
Read(A)		
	Read(B)	
A:= f_1(A)		
		Read(C)

	B:= f_2(B)
	Write(B)	

		C:= f_3(C)
		Write(C)

Write(A)		
		Read(B)

	Read(A)
	A:= f_2(A)	

Read(C)		
	Write(A)	

C:= f_1(C)
Write(C)		

		B:= f_3(B)
		Write(B)

Schedule S1

Steps for creating precedence graph

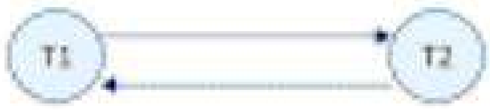
1. Final vertices

2. Draw edges by finding conflicting pairs

3. If there is no cycle, it's a serializable schedule and given consistent result.

4. If not we have to perform topological sorting to determine serializability order.

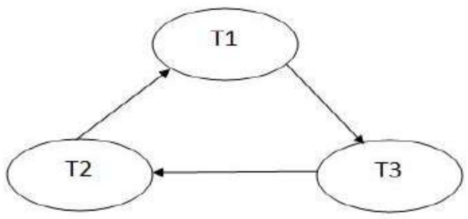
Precedence graph for above transaction is



Since it is a non-conflict serializable schedule it gives inconsistent results so we perform topological sorting

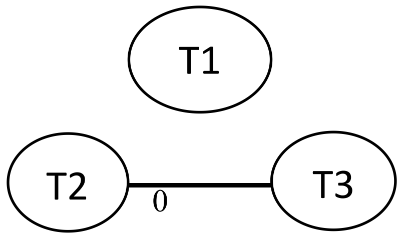
Topological sorting:

✓ No. of edges pointing towards a single transaction is called In degree.

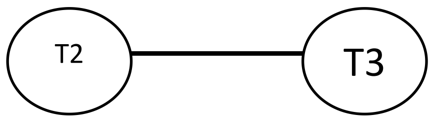


Steps:-

1. Choose the vertex with in degree 0.
2. Remove the outgoing edges from the vertex.
3. Repeat step1 & step2 until no vertex with in degree 0 exists.

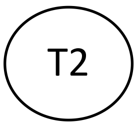
- 

Remove the outgoing edges from T1

- 

Remove the outgoing edges from T3 we get.

- 1 0



So the serialiability order of its equivalent serial schedule is: t1→t3→t2.

Recoverability:-

Recoverability in DBMS refers to the ability of a system to ensure that a database can be restored to a consistent state after a failure, such as system crashes, power outages, or transaction failure.

A schedule is recoverable if it ensure that:

- A transaction only commits after ensuring that all transactions whose changes it depends on have also committed.
- This prevents a situation where a transaction depends on an uncommitted transaction, which might be rolled back later, leading to inconsistencies.

Example:

- If transition T2 reads a value written by transaction t1, then T2 should only commit if T1 has committed.

Data base backup:

A data base backup is an exact copy positive (O+) your database kept in a separate location.

Data base recovery: data recovery refers to the process of resting a database to a consistent state after a failure or correction.

- we can recover data in DBMS by using some recovery techniques like rollback, redo undo operation, Log based recover shadow paying and Implementation of isolation :

1. Isolation is the one of the Acid property use to ensure data consistency and integrity

2. It also ensure that the transaction are executed independently without inference, preventing issues like dirty reads, non-repeatable reads and phantoms reads

3. Isolation is one of the core Acid properties of a database transactions, it ensure that the operations of one transaction hidden from no two transaction until completion. It means that no two transaction should inference with each other and affect the others intermediate state.

Levels of isolation:

- Isolation levels defines the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- 4 levels of isolation to balance between data consistency and system performance

1. Serializable
2. Repeated read
3. Read committed
4. Read uncommitted

1. **Serializable:**

❖ The highest isolation level. Transactions are executed in such a way that the outcome is the same as if the Transactions were executed serially (one after another). This prevents dirty reads, non-repeatable reads and phantom reads.

❖ Use case suitable for applications where strict consistency is required, such as financial application thought it can impact performance due to high contention and blocking.

2. **Repeatable Read:-**

- ♣ This is the most restrictive isolation level.
- ♣ It allows transactions to the change mode by other transaction
- ♣ The transaction holds read locks on all rows, it references.
- ♣ It holds write locks on all rows it inserts, updates, or deletes.
- ♣ It avoids non-repeatable read.
- ♣ Use case: Ideal for situation where you need, consistency in reading data across the same transaction.

3. **Read committed:-**

Transactions can only read committed data. They cannot see uncommitted changes from other transactions, preventing dirty reads.

- ♣ Use case: commonly used in many applications since it provides a balance between performance and consistency.

4. **Read uncommitted:-**

Transactions can read data that other transactions have written but not yet committed (dirty reads).

This is the least restrictive level.

- ♣ No locks are enforced.
- ♣ Dirty reads are allowed, meaning no transaction waits for other to complete.
- ♣ Use case: suitable for scenarios where data consistency is important (example:- reporting or log processing).

Isolation can be performed though locking-mechanism and Time stamp-based protocols.

1. **Locking Mechanisms:**
   It ensures exclusive access to a data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access the item.
   Types of lock 2 types of locks.
   i) Shared lock: (s-lock) are used for data that the transaction reads. It is also called as read lock.
   ii) Exclusive lock: (X-lock) are used for those if writes. This is also called as write lock.
2. **Time stamp-based protocol:**
   Timestamp is a value assigned to each transaction that indicates the order of it's another transaction.
   - ♣ Whereas time stamp-based protocol does not use locks but instead. It uses a concept of time-stamp, whenever a new transaction start it will be associated

with new time stamp, wherever a new time stamp which is increasing in order, generally it can be a system clock or anything

- It determines the order of execution of the transaction that will be ensure older transaction get priority.

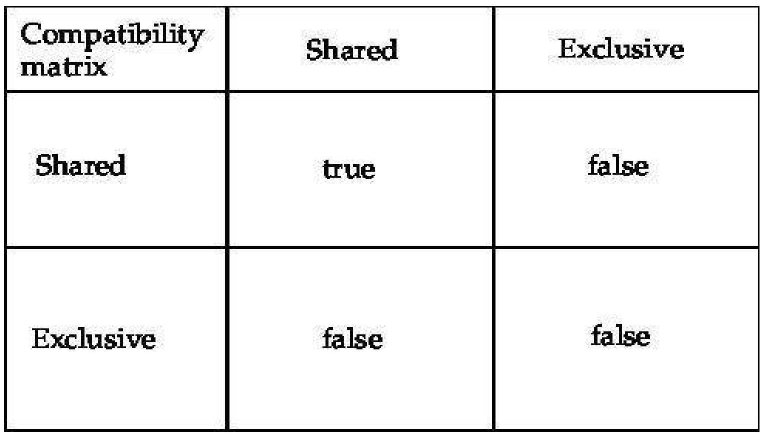
Lock-Based Protocol in a Database Management System:-

Lock based protocols in Database Management System are used to ensure concurrency control, which is crucial for maintaining data integrity when multiple transactions are being processed simultaneously. Lock regulate access to data items, ensuring that only one transaction can access a data item in a particular mode (e.g.., read or write) at a time.

Types of locks:-

1. Shared Lock (s-Lock):-A Shared lock is applied when a transaction wants to read a data item. Multiple transactions can hold a shared lock on the same data item, allowing them to read concurrently. However, no transaction can modify the data while it's locked in shared mode.

2. Exclusive lock(x-lock):-An exclusive lock is applied when a transaction wants to write or modify a data item. Only one transaction can hold an exclusive lock on a particular data item, and no other transaction can read or write item until the lock is released.



Compatibility matrix	Shared	Exclusive
Shared	true	false
Exclusive	false	false

Lock-Compatibility matrix comp.

We require that every transaction request a lock in an appropriate mode on data item, depending on the types of operations that it will perform on the transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants that lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Example:

Consider the banking example. Let A and B be two accounts that are accessed by transaction T1

T1: lock-x (B);

```
Read (B);
B: B-50;
Write (B);
Unlock (B);
Lock-x (A);
Read (A)
A: =A+50;
Write (A);
Unlock (A);
```

Figure:-Transaction T1.

And T2. Transaction T1 transfer ₹50 from account B to account a. transaction T2 display the total amount of money in accounts A and B- that is, the sum A+B.

Suppose that the values of accounts A and B are ₹100 and ₹200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value ₹300.

Two-phase locking protocol

One protocol that ensures serializability is the two phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

1) **Growing phase:** A transaction may obtain locks, but may not release any lock.
2) **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

- The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction.
- Two phase locking is of two types
    - Strict two phase locking
    - Rigorous two phase locking

Strict two phase locking: All exclusive locks taken by a transaction beheld until that transaction commits. (Only exclusive lock)

Rigorous two phase locking: All locks taken by a transaction be hid until the transaction commits. (Both)

- Converting shared lock to exclusive lock is called lock upgrade.
- Converting exclusive lock to shared lock is called lock downgrade.

Lock Conversion cannot be allowed arbitarrly. Rather, Upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Example:

Lock-x (A);   (Growing phase)

```
  Lock-x (B);    (Growing phase)
  Read (A);
  A: =A-50;
  Write (A);
  Read (B);
  B: =B+50;
 Write (B);
 Unlock (A);     (shrinking phase)
 Unlock (B);     (shrinking phase)
```

Deadlock Handling:-

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another in the set.

Let T0, T1……..,Tn be a set of transactions such that T0 is waiting for a data item that T1 holds and T1 is waiting for a data item that T2 holds and……, and Tn-1 is waiting for a data item Tn holds and Tn is waiting for a data item that T0 holds. None of the transaction can make progress and this situation is called deadlock.

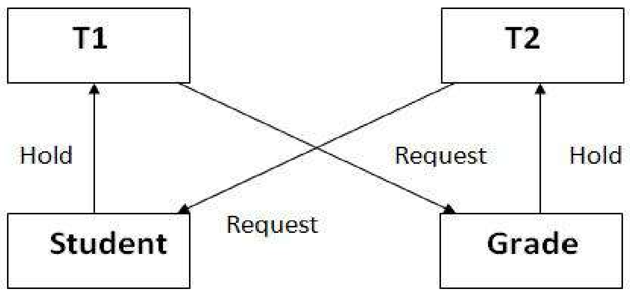


Figure: Deadlock in DBMS

There are two principal methods for dealing with the deadlock problem. We can use a deadlock prevention protocol to ensure that the System will never enter a deadlock state.

Alternatively, we can allow the System to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery.

Deadlock Detection

Deadlock can be described in items of a directed graph called a wait-for graph.

G= (V, E)

V=The set of transactions {T1, T2…..Tn}

E=The set of Edges {(Ti→Tj) (Tj→Tk….}

An edge Ti→Tj exists if transaction Ti waits for a data item being by transaction Tj

A deadlock exists in the system if and only if the wait. For graph contains a cycle.

Deadlock Recovery

The most common solution for deadlock is to rollback one or more transactions to break the deadlock

Three actions need to be taken

1. Selection of a victim
2. Rollback
3. Starvation

1. **Selection of a victim:-**
   Given a set of deadlocked transaction, we must determine which transaction to rollback to break the deadlock
   Many factors may determine the cost of a rollback including
   a. How long the transaction has computed and how much longer the transaction will computer before it completes its designated task.
   b. How many data item the transaction has used
   c. How many more data items the transaction needs for it to complete.
   d. How many transaction will be involved in the rollback
2. **Rollback:-**
   Once we have decided that a particular transaction must be rolled back, we must determine how for this transaction should be rolled back.
   There are two options for rollback
   a) Total Rollback
   b) Partial Rollback

Total rollback:-The total transaction is rolled back and then restarted.

Partial rollback:-The transaction must be rolled back to the point to break the deadlock. It requires the system to maintain additional information about the state of all the running transactions. The system should decide which lacks the selected transaction needs to release in order to break the deadlock.

3. **Starvation:-**
   In a system where the selection of victims is based primarily on cost factors. It may happen that the same transaction is always picked as a victim. As a result the transaction never completes and it is known as starvation.
   The most common solution is to include the no. of rollback in the cost factor.

Deadlock prevention:-

It includes mechanisms that ensures that the system will never enter a deadlock stage.
Two deadlock prevention schemes using time stamps have been proposed.

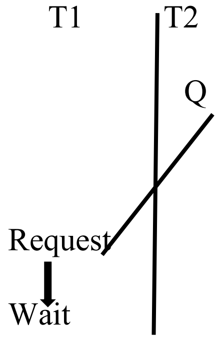
1. Wait-die schema

2. Wound-wait schema

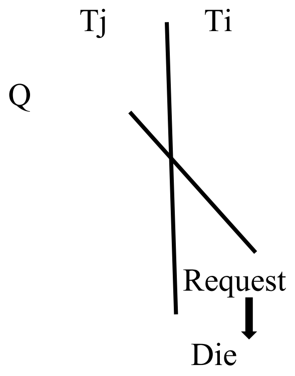
Wait-die scheme:-

↔ It is a non-preemptive technique.

↔ When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if Ti has a timestamp smaller than that of Tj. Otherwise Ti is rolled back.



Ts(Ti)<Ts(Tj)

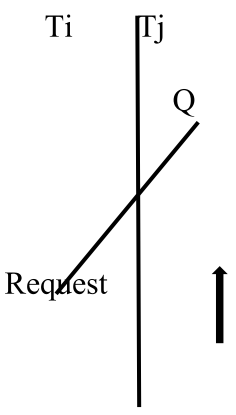


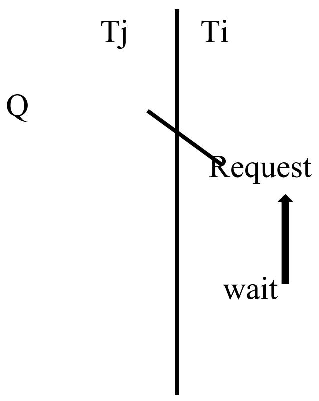
Ts(Ti)>Ts(Tj)

Wound-wait Scheme:-

↔ It is a preemptive technique.

↔ When transaction ti request a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp larger than that of Tj. Otherwise Tj is rolled back





Wound

Ts (Ti) <Ts (Tj)   Ts (Ti) >Ts (Tj)

Timestamp-based protocol:-

A protocol which is based on the ordering of the transaction is known as timestamp-based protocol.

Timestamp: A unique number assigned to each transaction before the transaction starts its execution is known a timestamp

The timestamp of a transaction Ti is denoted by Ts (Ti). There are two methods for assigning timestamp.

1. The value of the system clock is assigned as the timestamp of the transaction.
2. The value of the logical counter is assigned as the timestamp of the transaction. The value of the logical counter is incremented after a new timestamp has been assigned.

To implement this scheme, we associate each data item Q two timestamp values:

W-timestamp (Q):-It denotes the largest timestamp of any transaction that executed write (Q) Successfully

R-timestamp (Q):-It denotes the largest timestamp of any transaction that executed read (Q) Successfully.

≈ These timestamps are updated wherever a new read (Q) or write (Q) instruction is executed.

≈ The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

This protocol operates as follows:

1. Suppose that transaction Ti issues read (Q):
   a) If Ts (Ti) <W-timestamp (Q), then the read operation is rejected and Ti is rolled back.
   b) If Ts (Ti) >W-timestamp (Q), then the read operation is executed and R-timestamp (Q) is set to the maximum R-timestamp (Q) and Ts (Ti).
2. Suppose that transaction Ti issues write (Q)
   a) If Ts (Ti) <R-timestamp (Q), then the write operation is rejected and Ti is rolled back.
   b) If Ts (Ti) <W-timestamp (Q), then the write operation is rejected and Ti is rolled back.
   c) Otherwise, the system executes the write operation and sets W-timestamp (Q) to Ts (Ti).

Thomas write Rule:

Suppose that transaction Ti issues write (Q).

1. If Ts (Ti) <R-timestamp (Q), then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
2. If Ts (Ti) <w-timestamp (Q), then Ti is attempting to write on obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets w-timestamp (Q) to Ts(Ti).

Recovery Algorithm:-

A recovery algorithm in a dbms is a set of actions that a system uses to recover from a failure and restore the database to a consistent state.

Transaction Rollback-

First consider transaction rollback during normal operation (that is, not during recovery from a system crash). Rollback of transaction Ti is performed as follows:

1. The log is scanned backward, and for each log record of Ti of the form <Ti, Xj, V1, V2>that is found:
   a) The value V1 is written to data item Xj and
   b) A special redo-any log record<Ti, Xj, V1>is written to the log, where V1 is the value being restored to data item Xj during the rollback. These log records are sometimes called compensation log records. Such records do not need undo information, since we never need to undo such as undo such an undo operation.
2. Once the log record<Ti start> is found the backward scan is stopped, and a log record<Ti abort> is written to the log.

Recovery After a system crash

Recovery actions, when the database system is restarted after a crash, takes place in two phases:

1. In the redo phase, the system replays updates of all transaction by scanning the log forward from the last checkpoint.

The specific steps takes while scanning the log are as follows:

a. The list of transaction to rolled back, undo-list, is initially set to the list "L" in the <checkpoint L> log record.
b. Whenever a normal log record of the form <Ti, Xj, V1, V2> or a redo-only log record of the form<Ti, Xj, V2> is encountered, the operation is redone; that is the value V2 is written to data item Xj.
c. Whenever a log record of the form<Ti start> is found, Ti is added to undo-list.
d. Whenever a log record of the form<Ti abort> or<Ti commit> is found, Ti is removed from undo-list.

At the end of the redo phase, undo-list contains the list of all transaction that are incomplete, that is, they neither committed nor completed rollback before the crash.

2. In the undo phase, the system roll back all transaction in the undo list. It performs rollback by scanning the log backward from the end.
   a. Whenever it finds a log record belonging to a transaction in the undo list, it performs undo actions just as if the log record had been found during the rollback of a failed transaction.
   b. When the system finds a<Ti start> log record for a transaction Ti in undo-list, it writes a<Ti abort > log record to the log, and removes Ti from undo-list.
   c. The undo phase terminates once undo-list becomes empty, that is, the system has found<Ti start> log records for all transaction that were initially in undo-list.

After the undo phase of recovery terminates, normal transaction processing can resume.

- ❖ Recovery algorithm contains log-based recovery, shadow paging, checkpoints to maintain consistency of database.

Introduction to Indexing Techniques:

Introduction of B+ tree:-

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a **B+ tree structure** of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree
Features of B+ Trees:-
- **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
- **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
- **Ordered:** B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
- **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.
- **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.
- **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

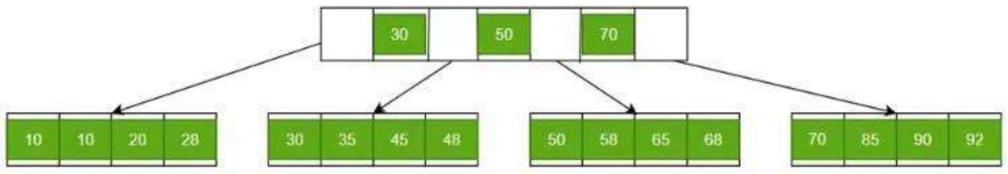
Why Use B+ Tree?
- B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
- B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Implementation of B+ Tree:

In order, to implement dynamic multilevel indexing, B-tree and B+ tree are generally employed. The drawback of the B-tree used for indexing, however, is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree. This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record. B+ tree eliminates the above drawback by storing data pointers only at the leaf nodes of the tree. Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record. From the above discussion, it is apparent that a B+ tree, unlike a B-tree, has two orders, 'a' and 'b', one for the internal nodes and the other for the external (or leaf) nodes

Structure of B+ trees:-



B+ Trees contain two types of nodes:

- **Internal Nodes:** Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- **Leaf Nodes:** Leaf Nodes are the nodes that have n pointers.

Insertion in B+ Trees:

Insertion in B+ Trees is done via the following steps.

- Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
- Insert the key into the leaf node in increasing order if there is no overflow.

Deletion in B+ Trees:

Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

Advantages of B+ Trees:

- A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.

- Data stored in a B+ tree can be accessed both sequentially and directly.
- It takes an equal number of disk accesses to fetch records.
- B+ trees have redundant search keys, and storing search keys repeatedly is not possible.

Disadvantages of B+ Trees:

- The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

Application of B+ Trees:

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)

Introduction:-

The B+ Tree is a type of balanced binary search tree. It uses multilevel indexing, with leaf nodes holding actual data references. An important feature is that all leaf nodes in a B+ tree are kept at the same height.

Search is made efficient as the leaf nodes in a **B+ Tree** are linked in the form of singly-linked lists. You can see the difference between B trees and B+ tree structures, respectively, where records in the latter are stored as linked lists.

B+ Trees in DBMS store a huge amount of data that cannot be stored in the limited main memory. The leaf nodes are stored in secondary memory, while only the internal nodes are stored in the main memory. This is called multilevel indexing.

B++ Trees are a type of self-balancing tree data structure that are used to store and retrieve large amounts of data efficiently. They are similar to B-Trees, but with some additional features that make them better suited for use in Database systems. B++ Trees maintain a separate leaf node structure that is connected by a linked list, which allows for efficient range queries and sequential access to data. Additionally, B++ Trees have a higher fanout than B-trees.

Why Use a B+ Tree?

A **B+ tree** is a self-balancing tree data structure primarily used in **database and file systems**. It provides an efficient way to store and retrieve data by maintaining balance across its nodes. B+ trees are an enhancement of **B-trees**, a balanced tree structure, but with a few distinct features that make them particularly well-suited for disk-based storage systems.

Here's why B+ trees are commonly used:

- **Efficient Range Queries**: Unlike binary search trees, B+ trees store all keys in leaf nodes in a sorted manner. Leaf nodes are linked, allowing sequential access, which makes range queries efficient (e.g., retrieving all entries in a certain range).
- **Fast Access to Disk-Based Storage**: Since B+ trees minimize the depth of the tree, they reduce the number of disk I/O operations required to access data. Non-leaf nodes only contain keys (not actual records), which allows more keys to be stored per node, minimizing disk reads and reducing the search time.
- **Improved Memory Utilization**: B+ trees use a large branching factor, which minimizes the height of the tree and enables more efficient use of memory, allowing them to store more data without increasing the tree's depth significantly.
- **Balanced Structure**: Like other balanced trees, B+ trees keep all leaf nodes at the same depth, ensuring uniform data retrieval times. This balance is maintained automatically with insertions and deletions.
- **High Insertion and Deletion Efficiency**: In B+ trees, insertions and deletions require minimal reorganization, and since keys are sorted only in the leaf nodes, only those nodes (and sometimes their parents) need to be adjusted when the structure changes.

Implementation of B+ Tree in DBMS:-

In DBMS, a B+ tree is used for indexing large datasets. The key concepts involved in a B+ tree implementation include **nodes, branching factor, and balancing** mechanisms. Here's an overview of how a B+ tree works in a DBMS context:

Nodes and Order:

- A B+ tree has a defined **order**, which dictates the maximum number of children a node can have. For instance, an order-4 B+ tree allows a maximum of 4 children per node.
- Internal nodes (non-leaf nodes) contain only keys for navigation, while **leaf nodes** contain actual data records or pointers to the records.

Insertion:

- Inserting a new key follows a search for the appropriate leaf node. If the leaf node has space, the new key is added.
- If the node is full, it splits, promoting the middle key to the parent node and creating a new leaf. This process continues up the tree as necessary, ensuring balance.

Deletion:

- When a key is deleted, it's removed from the leaf node. If this causes underflow (fewer keys than the minimum), neighboring nodes may lend keys. If that's not possible, nodes may merge, and balancing continues up the tree if needed.

Range Queries:

- Since all data records are stored in sorted order at the leaf level and leaf nodes are linked, range queries can be done quickly by traversing through the leaf nodes, making it very efficient.

Splitting and Merging:

- Splits and merges happen as part of maintaining balance. Splitting involves dividing a full node and pushing one key up to maintain order. Merging occurs when nodes have too few keys, keeping the tree balanced without redundancy.

Search Operation:

- Searching in a B+ tree involves traversing from the root to the leaf level, following keys in the internal nodes. Since B+ trees are balanced and have a shallow depth, this search process is efficient.

Properties of B+ Tree in DBMS:-

A B+ tree is a type of self-balancing search tree data structure that maintains sorted data and allows searches, insertions, and deletions in logarithmic time. Here are some key properties of B+ trees:

- **Balanced Structure:** B+ trees are self-balancing, meaning that after every insertion or deletion operation, the tree automatically reorganizes itself to maintain balance. The balance ensures that the depth of the tree remains logarithmic, leading to efficient search, insert, and delete operations.
- **Node Structure:** The tree is composed of nodes, where each node represents a range or interval of keys. Internal nodes contain pointers to child nodes and key values that act as separators for the ranges represented by their children. Leaf nodes store actual data and are linked together to support range queries efficiently.
- **Keys and Values:** Each internal node has a set of keys that act as separators for the ranges represented by its child nodes. Leaf nodes store key-value pairs, and keys in leaf nodes are usually sorted.
- **Order of the Tree:** The order of a B+ tree is a parameter that determines the maximum number of children a node can have. In a B+ tree of order m, an internal node can have

at most m-1 keys and m children, and a leaf node can have at most m-1 key-value pairs. Higher-order trees generally lead to shallower trees but may require more memory.

Structure of B+ Tree in DBMS:-

Before we study the structure of B+ Trees, there are a few terms we need to understand:

(a) Indexes

They are references to actual records in a database. Indexing helps the database search any given key in correspondence to the tree.

(b) Leaf nodes

The leaf nodes or leaves are the bottommost nodes marking the end of a tree. We must note that all the leaf nodes are on the same level.

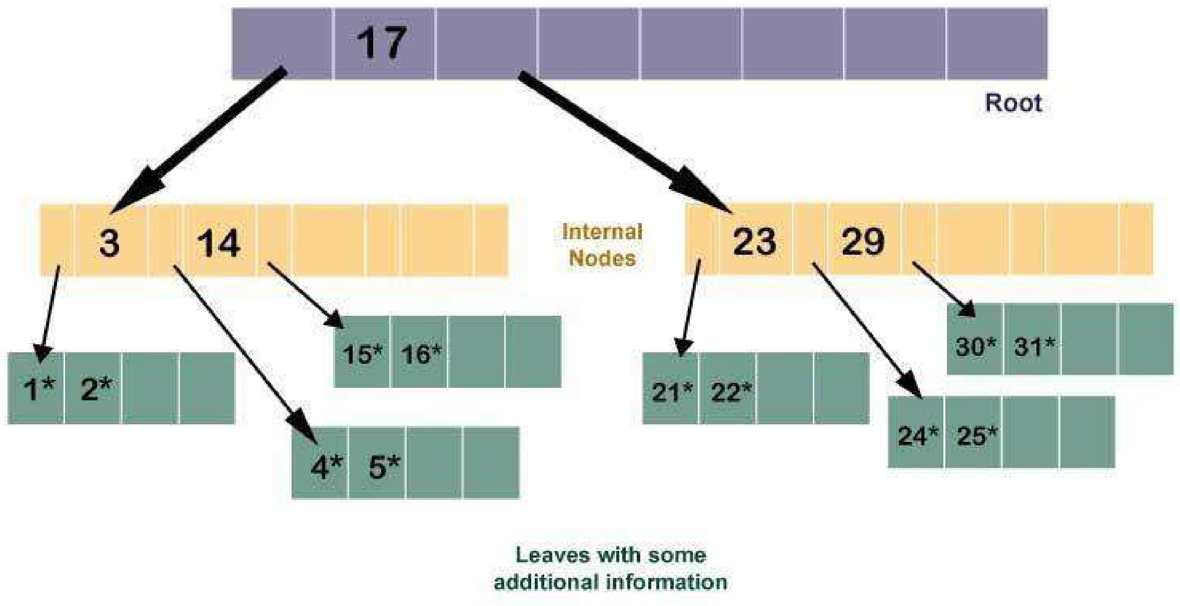
(c) Depth of tree

It refers to the number of levels of nodes from the root to the leaves. The depth of a tree keeps increasing as more and more elements are added to the tree.

(d) Order of B+ tree

This is the maximum number of keys in a record in a B+ Tree.

All the leaves are the same depth. All leaves in this tree are depth 2, assuming the root is at depth 0. A B+ tree is also a search tree, but in a binary search tree, there is only one search key and two child pointers at each node. On the other hand, in a B+ tree with order 'd', there are between 'd' and 2*d keys (meaning between 2 and 4 keys in our example tree) at each node (except probably the root). There are between d+1 and 2(d+1) child pointers.



Now in our **B+ tree of order 2**, the yellow internal nodes have between **2-4 keys**; they each have at least a minimum of two. Each of the leaves in green has between 2-4 keys, the key values of which are shown with an asterisk (to indicate other information associated with a search key). There are pointers at the leaves, which point to the additional information associated with keys.

Must Recommended Topic, Introduction to DBMS.

Searching a record in B+ Tree

Searching a record in a B+ tree involves the following steps:

1. Start at the root node of the tree.

2. Compare the search key with the keys in the current node.

3. If the search key is less than the smallest key in the node, follow the leftmost pointer to the child node.

4. If the search key is greater than or equal to the largest key in the node, follow the rightmost pointer to the child node.

5. If the search key is between two keys in the node, follow the pointer to the child node corresponding to the key that is just greater than the search key.

6. Repeat steps 2-5 until a leaf node is reached.

7. Search for the record in the leaf node using the search key to locate the corresponding entry.

8. If the record is found, return it. If not, return a "record not found" message.

Because B+ trees are balanced and all leaf nodes are at the same level, the search time is O(log n), where n is the number of records in the tree.

Let's take an example.

Assume we have the following B+ tree, where each node can hold up to 3 keys:

```
                  [23, 34, 45]
                 /  |   |  \
         [5, 7, 10]   [23] [34]   [45, 56, 67]
         /  |  \               |   |  \
   [1, 2, 3] [5] [7]              [45] [56] [67, 89]
```

To search for a record with a key of 45, we would start at the root node, which contains the keys 23, 34, and 45. Because 45 is greater than or equal to the smallest key in the node (23) and less than the largest key in the node (45), we would follow the middle pointer to the second child node.

In the second child node, we would find the key value of 45 in the leaf node. We would then use this key to locate the corresponding record in the leaf node and return the record to the user. If the key value was not found, we would return a "record not found" message.

Operations on B+ Tree in DBMS:-

(e) Insertion in B+ Tree

Inserting a record in a B+ tree involves the following steps:

1. Search the tree to determine the appropriate leaf node for the new record.

2. If the leaf node has room for the new record (i.e., it has fewer than m-1 records, where m is the order of the tree), insert the new record in the correct position and update the pointers as necessary.

3. If the leaf node is full, split the node into two halves and promote the median key to the parent node.

4. Repeat step 2 for the appropriate leaf node in the new subtree.

Here's an example of inserting a record with a key value of 30 in a B+ tree with an order of 3:

```
           [10, 20, 30, 40, 50]
          /   |     |      \
   [1, 2, 3]  [10] [20]  [30, 40] [50, 60, 70]
```

1. Search the tree to determine the appropriate leaf node for the new record. In this case, the appropriate leaf node is the one that contains the key values 20 and 30.

2. Because the leaf node has room for the new record, we can simply insert it in the correct position:

3. If the leaf node were full, we would split it into two halves and promote the median key to the parent node.

4. We would then update the pointers in the parent node to reflect the new child nodes. If the parent node were full, we would repeat the split process until the tree was balanced.

(f) Deletion in B+ Tree

Deleting a record from a B+ tree involves the following steps:

1. Search the tree to find the leaf node containing the record to be deleted.

2. Delete the record from the leaf node.

3. If the leaf node has more than the minimum number of records (i.e., at least ceil((m-1)/2) records, where m is the order of the tree), we are done.

4. If the leaf node has fewer than the minimum number of records, try to redistribute records from neighboring nodes. If redistribution is not possible, merge the node with a neighboring node.

5. Update the parent node keys and pointers as necessary.

Here's an example of deleting a record with a key value of 20 from a B+ tree with an order of 3:

```
           [10, 20, 30, 40, 50]
          /      |     |    \
  [1, 2, 3]  [10] [20, 30]  [40] [50, 60, 70]
```

1. Search the tree to find the leaf node containing the record to be deleted. In this case, the leaf node is the one that contains the key values 20 and 30.

2. Delete the record from the leaf node:

3. Because the leaf node has more than the minimum number of records, we are done.

4. If the leaf node had fewer than the minimum number of records, we would try to redistribute records from neighboring nodes. If redistribution was not possible, we would merge the node with a neighboring node. For example, if the node containing the key values 10 and 30 had been deleted instead, we would have to redistribute or

5. We would then update the parent node keys and pointers as necessary.

Properties of B+ Tree:-

B+ tree is a specialized Data Structure used for indexing and searching large amounts of data. Here are the key properties of a B+ tree:

- All keys are stored in the leaves, and each leaf node is linked to its neighbouring nodes.
- Non-leaf nodes contain only keys and pointers to child nodes.
- Each node (except for the root) has at least ceil(m/2) keys, where m is the order of the tree.
- All leaf nodes are at the same level, which ensures efficient range queries.
- Each node (except for the root) has at least ceil(m/2) child pointers.
- The root node may have fewer keys and child pointers than other nodes.
- The tree is balanced, which means that the path from the root to any leaf node has the same length.

To understand B+ Trees better, we first need to understand multilevel indexing in more detail.
You can also read about - Specialization and Generalization in DBMS
Let us get started now:

Features of B+ Tree

The following are some key features of B+ Trees:-

- Balanced Tree Structure
- Quick insertion and search operation
- Variable node capacity
- Disk-based storage
- Minimized disk I/O
- Effective range queries.

Advantages of using B+ Trees in DBMS

By now, we have learned how B+ Trees make a very convenient option for database users. Some of the advantages are listed right below:

- The height of B+ Trees is balanced as compared to the B-Trees.

Keys are used for indexing in B+ Trees

- Users can access the data in B+ Trees directly and sequentially.
- Since the data is stored only in leaf nodes, search queries are much faster.

Disadvantages of B+ Trees in DBMS

≈ The following are some disadvantages of B+ trees:-
Increased complexity in insertion and deletion operations

≈ Higher overhead in terms of memory usage
≈ Slower tree traversal compared to binary trees
≈ Less cache-friendly due to wide nodes

Application of B+ Trees

The applications of B+ trees are:

Database Indexing: B+ trees are widely used in database management systems for indexing. They provide efficient retrieval of records based on the values of the indexed columns.

File Systems: B+ trees are employed in file systems to organize and manage large amounts of data efficiently. They help in quick retrieval of file blocks and support sequential access.

Information Retrieval: B+ trees are used in information retrieval systems, such as search engines, to index and quickly locate relevant documents based on keywords.

Geographic Information Systems (GIS): GIS applications use B+ trees to index spatial data efficiently, facilitating spatial queries and range searches.

DNS Servers: Domain Name System (DNS) servers often use B+ trees to store and manage domain names, enabling fast lookups and updates.

File Databases: B+ trees are utilized in file databases, where they help organize and manage large volumes of data with efficient search and retrieval operations.

Caching Mechanisms: B+ trees can be employed in caching mechanisms to efficiently store and retrieve frequently accessed data, improving overall system performance.

Memory Management: B+ trees are used in memory management systems to efficiently organize and locate data in virtual memory.

Practice Problems on B+ Tree

The practice problems on B+ tree are:

Insertion Operation: Given a B+ tree of a certain order, practice inserting key-value pairs into the tree. Perform the necessary split operations to maintain the balance.

Deletion Operation: Practice deleting keys from a B+ tree. Perform the necessary merge and redistribution operations to ensure the tree remains balanced.

Search Operation: Given a B+ tree and a set of keys, practice performing search operations to locate specific keys within the tree.

Range Queries: Create a B+ tree and practice performing range queries. Identify and retrieve all key-value pairs within a specified range.

Sequential Access: Design a B+ tree and simulate sequential access operations. Traverse the tree in a way that efficiently retrieves keys in sorted order.

Duplicate Keys: Modify a B+ tree to handle duplicate keys. Practice inserting, deleting, and searching for key-value pairs with the same key.

Hashing in DBMS:-

Hashing in DBMS is a technique to quickly locate a data record in a database irrespective of the size of the database. For larger databases containing thousands and millions of records, the indexing data structure technique becomes very inefficient because searching a specific record through indexing will consume more time. This doesn't align with the goals of DBMS, especially when performance and data retrieval time are minimized. So, to counter this problem hashing technique is used. In this article, we will learn about various hashing techniques.

What is Hashing?

The hashing technique utilizes an auxiliary hash table to store the data records using a hash function. There are 2 key components in hashing:

- **Hash Table:** A hash table is an array or data structure and its size is determined by the total volume of data records present in the database. Each memory location in a hash table is called a '***bucket***' or hash indice and stores a data record's exact location and can be accessed through a hash function.
- **Bucket:** A bucket is a memory location (index) in the hash table that stores the data record. These buckets generally store a disk block which further stores multiple records. It is also known as the hash index.
- **Hash Function:** A hash function is a mathematical equation or algorithm that takes one data record's primary key as input and computes the hash index as output.

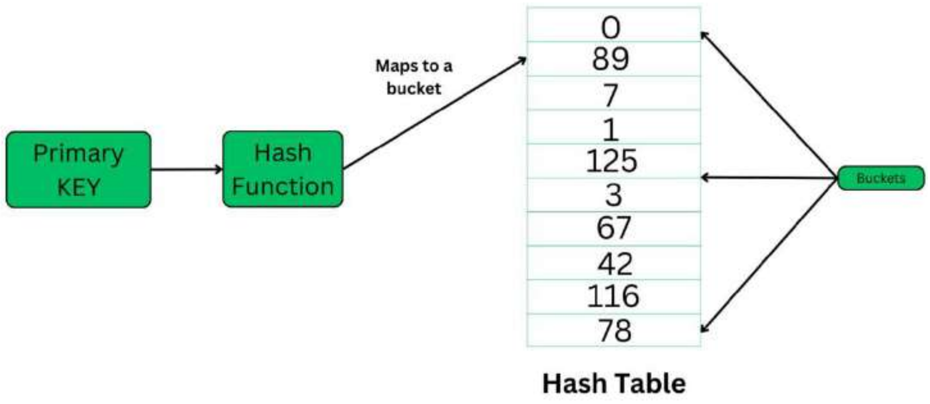
Hash Function:-

A hash function is a mathematical algorithm that computes the index or the location where the current data record is to be stored in the hash table so that it can be accessed efficiently later. This hash function is the most crucial component that determines the speed of fetching data.

Working of Hash Function:-

The hash function generates a hash index through the primary key of the data record.
Now, there are 2 possibilities**:**
1. The hash index generated isn't already occupied by any other value. So, the address of the data record will be stored here.
2. The hash index generated is already occupied by some other value. This is called collision so to counter this, a collision resolution technique will be applied.
3. Now whenever we query a specific record, the hash function will be applied and returns the data record comparatively faster than indexing because we can directly reach the exact location of the data record through the hash function rather than searching through indices one by one.

Example:



Types of Hashing in DBMS

There are two primary hashing techniques in DBMS.

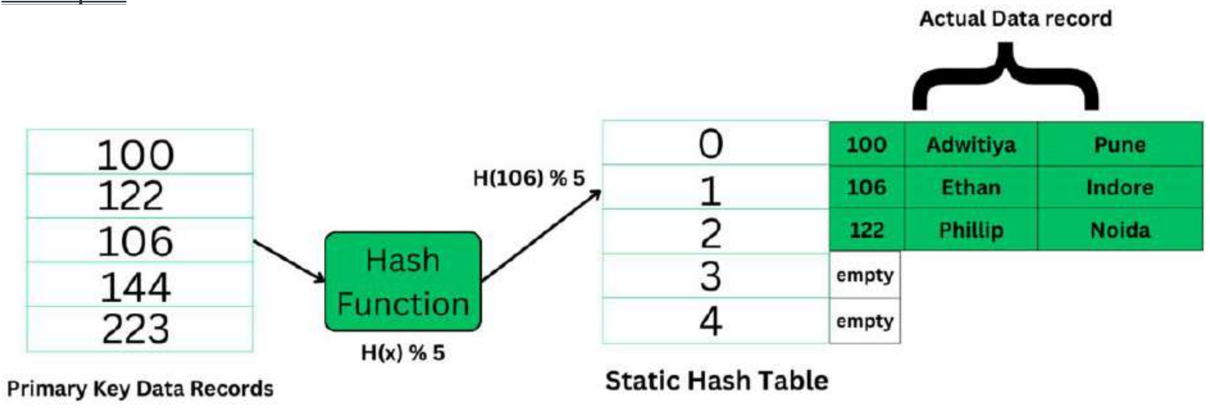
1. Static Hashing

In static hashing, the hash function always generates the same bucket's address. For example, if we have a data record for employee id = 107, the hash function is mod-5 which is - H(x) % 5, where x = id. Then the operation will take place like this:

H(106) % 5 = 1.

This indicates that the data record should be placed or searched in the 1st bucket (or 1st hash index) in the hash table.

Example:



The primary key is used as the input to the hash function and the hash function generates the output as the hash index (bucket's address) which contains the address of the actual data record on the disk block.

Static Hashing has the following Properties:

- **Data Buckets:** The number of buckets in memory remains constant. The size of the hash table is decided initially and it may also implement chaining that will allow handling some collision issues though, it's only a slight optimization and may not prove worthy if the database size keeps fluctuating.
- **Hash function:** It uses the simplest hash function to map the data records to its appropriate bucket. It is generally modulo-hash function
- **Efficient for known data size:** It's very efficient in terms when we know the data size and its distribution in the database.
- It is inefficient and inaccurate when the data size dynamically varies because we have limited space and the hash function always generates the same value for every specific

input. When the data size fluctuates very often it's not at all useful because collision will keep happening and it will result in problems like - bucket skew, insufficient buckets etc.

To resolve this problem of bucket overflow, techniques such as - chaining and open addressing are used. Here's a brief info on both:

1. Chaining:-

Chaining is a mechanism in which the hash table is implemented using an array of type nodes, where each bucket is of node type and can contain a long chain of linked lists to store the data records. So, even if a hash function generates the same value for any data record it can still be stored in a bucket by adding a new node.
However, this will give rise to the problem bucket skew that is, if the hash function keeps generating the same value again and again then the hashing will become inefficient as the remaining data buckets will stay unoccupied or store minimal data.

(d)2. Open Addressing/Closed Hashing:

This is also called closed hashing this aims to solve the problem of collision by looking out for the next empty slot available which can store data. It uses techniques like **linear probing**, **quadratic probing**, **double hashing,** etc.

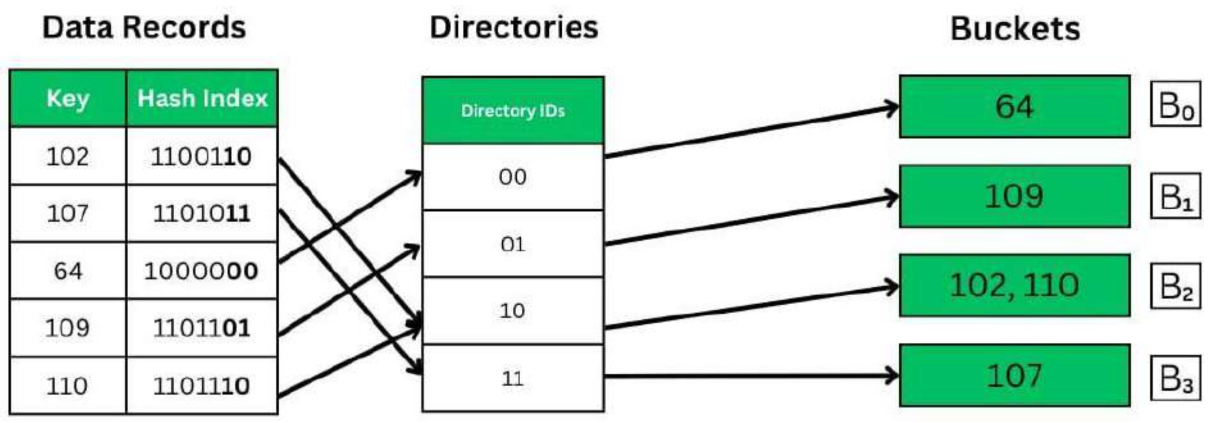
2. Dynamic Hashing:

Dynamic hashing is also known as extendible hashing, used to handle database that frequently changes data sets. This method offers us a way to add and remove data buckets on demand dynamically. This way as the number of data records varies, the buckets will also grow and shrink in size periodically whenever a change is made.
Properties of Dynamic Hashing

- The buckets will vary in size dynamically periodically as changes are made offering more flexibility in making any change.
- Dynamic Hashing aids in improving overall performance by minimizing or completely preventing collisions.
- **It has the following major components:** Data bucket, Flexible hash function, and directories
- A flexible hash function means that it will generate more dynamic values and will keep changing periodically asserting to the requirements of the database.
- Directories are containers that store the pointer to buckets. If bucket overflow or bucket skew-like problems happen to occur, then bucket splitting is done to maintain efficient retrieval time of data records. Each directory will have a directory id.
- **Global Depth:** It is defined as the number of bits in each directory id. The more the number of records, the more bits are there.

Working of Dynamic Hashing:-

Example: If global depth: k = 2, the keys will be mapped accordingly to the hash index. K bits starting from LSB will be taken to map a key to the buckets. That leaves us with the following 4 possibilities: 00, 11, 10, 01.



As we can see in the above image, the k bits from LSBs are taken in the hash index to map to their appropriate buckets through directory IDs. The hash indices point to the directories, and the k bits are taken from the directories' IDs and then mapped to the buckets. Each bucket holds the value corresponding to the IDs converted in binary.