DATABASE MANAGEMENT SYSTEM (303105203)

UNIT 1: INTRODUCTION

Q1: Difference Between DBMS and File Processing System

Feature	File Processing System	DBMS
Data Storage	Data is stored in separate files.	Data is stored in a centralized database.
Data Redundancy	High redundancy, same data may be duplicated in multiple files.	Redundancy is minimized due to data integration.
Data Inconsistency	Common due to redundant data.	Minimized due to centralized control.
Data Access	Manual and application-specific programs required.	Accessed using query languages like SQL.
Security	Limited control; handled by the application.	High-level security features like user roles and access rights.
Backup & Recovery	Not automated; requires manual procedures.	Built-in backup and recovery mechanisms.
Data Integrity	Difficult to enforce constraints.	Integrity constraints can be enforced at schema level.

In the realm of data management, two predominant systems have emerged: the traditional File Processing System (FPS) and the more sophisticated Database Management System (DBMS). While both systems serve the purpose of data storage and management, they exhibit significant differences in their architecture, functionality, and overall efficiency. Understanding these differences is essential for grasping the evolution of data management technologies and their implications for modern applications.

1. Data Storage

In a File Processing System, data is stored in separate, independent files, often tailored for specific applications. Each application that requires data access must define its own file structure and handling mechanism. This leads to a fragmented data environment where each application maintains its own data files, complicating data management and integration. For instance, if a company has multiple applications for sales, inventory, and customer

management, each may store customer data in its own file, leading to potential discrepancies and difficulties in data retrieval.

Conversely, a Database Management System provides a centralized data repository where data is organized in a structured format, typically in tables. This centralized approach allows for better organization and management of data, as all applications can interact with the same set of data in a uniform manner. The DBMS architecture supports relationships between different data entities, enabling complex queries and data manipulation across various applications.

2. Data Redundancy

Data redundancy refers to the unnecessary duplication of data across different files. In a file processing system, the same data may be stored in multiple files because each application maintains its own copy of the data it requires. This redundancy not only consumes excessive storage space but also increases the risk of data inconsistency. For example, if a customer's address is updated in one file but not in others, it can lead to conflicting information across the system.

In contrast, a DBMS is designed to minimize redundancy by centralizing data storage. Each piece of data is stored only once and can be accessed by multiple applications or users. This not only conserves storage space but also ensures consistency across the system. By eliminating redundancy, a DBMS enhances data integrity and simplifies data management.

3. Data Inconsistency

Data inconsistency is a common issue in file-based systems due to high redundancy. When the same data is stored in multiple locations, updates made in one location may not be reflected in others, leading to conflicting information. For instance, if a customer's phone number is updated in one application but not in another, it creates confusion and potential errors in communication.

A DBMS addresses this problem by maintaining a single source of truth. Since data is stored centrally and shared across applications, any update to the data is reflected system-wide, significantly reducing the chances of inconsistency. This centralized control ensures that all users access the most current and accurate data, enhancing the reliability of the information.

4. Data Access

In file processing systems, data access is often cumbersome and requires manual intervention or the development of application-specific programs. This can be particularly challenging as the system grows in size and complexity. Developers may need to write custom code to retrieve or manipulate data, which can be time-consuming and error-prone.

In contrast, Database Management Systems provide powerful query languages, such as SQL (Structured Query Language), that allow users to retrieve and manipulate data efficiently. SQL

is designed to be user-friendly and can handle complex queries with minimal code. This ease of access empowers users to interact with the database without needing extensive programming knowledge, thereby increasing productivity and reducing the likelihood of errors.

5. Security

File-based systems typically offer limited security mechanisms. Since each file is managed independently, controlling access to sensitive data becomes challenging. Security features must be integrated into each application, which can lead to inconsistencies and vulnerabilities.

A DBMS, on the other hand, provides robust security features that allow administrators to define user roles, permissions, and access controls. This centralized security model ensures that only authorized users can view or modify sensitive data, making DBMS a much safer option for organizations that handle confidential or critical information.

6. Backup and Recovery

In file processing systems, backup and recovery processes are often manual and require custom scripts or programs. This can lead to errors and inconsistencies during backup operations, increasing the risk of data loss.

A DBMS includes automated tools for backup and recovery, allowing for regular backups and maintaining logs of changes. In the event of a failure, the system can be restored to a consistent state, ensuring high availability and data durability. This automation not only simplifies the backup process but also enhances the reliability of data management.

7. Data Integrity

Maintaining data integrity—ensuring that data is accurate and consistent—is a significant challenge in file processing systems. Without centralized control, enforcing rules and constraints requires manual checks or application code, which can be error-prone.

A DBMS allows for the definition of integrity constraints at the schema level. Constraints such as primary keys, foreign keys, unique values, and not-null conditions are enforced automatically by the DBMS. This capability improves the quality and reliability of the data, ensuring that it adheres to predefined rules and standards.

While file processing systems laid the groundwork for early data storage techniques, they are fraught with limitations such as redundancy, inconsistency, poor security, and limited data integrity. Modern organizations increasingly rely on Database Management Systems due to their robust, efficient, and scalable approach to managing large volumes of data. With features like centralized control, data independence, secure access, automated backup and recovery, and powerful query capabilities, DBMS has become the standard for building reliable and efficient information systems.

Q2: ANSI/SPARC 3-Level Architecture

The ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee) 3-level architecture is a framework designed to standardize the way databases are structured and accessed. This architecture separates the user views from the physical data storage, providing a clear distinction between different levels of abstraction in database management. Understanding this architecture is crucial for database design and implementation, as it enhances data independence and simplifies data management.

1. External Level (View Level)

The external level, also known as the view level, is the highest level of abstraction in the ANSI/SPARC architecture. It describes user-specific views of the database, allowing different users to interact with the data in a way that is relevant to their needs. Each user or application can have its own external view, which may include a subset of the data or a specific representation of the data.

For example, in a university database, a student may only see their personal information, course enrollments, and grades, while an administrator may have access to a broader view that includes all student records and administrative data. This level of abstraction ensures that users can work with the data without needing to understand the underlying complexities of the database structure.

2. Conceptual Level (Logical Level)

The conceptual level, also referred to as the logical level, provides a unified view of the entire database. It describes what data is stored in the database and the relationships among the various data entities. This level abstracts away the details of how the data is physically stored, focusing instead on the logical structure of the data.

At the conceptual level, the database designer defines the overall schema, including entities, attributes, and relationships. For instance, in a university database, the conceptual schema might define entities such as Students, Courses, and Instructors, along with their attributes and the relationships between them (e.g., students enroll in courses). This level serves as a bridge between the external views and the internal storage, ensuring that all users have a consistent understanding of the data.

3. Internal Level (Physical Level)

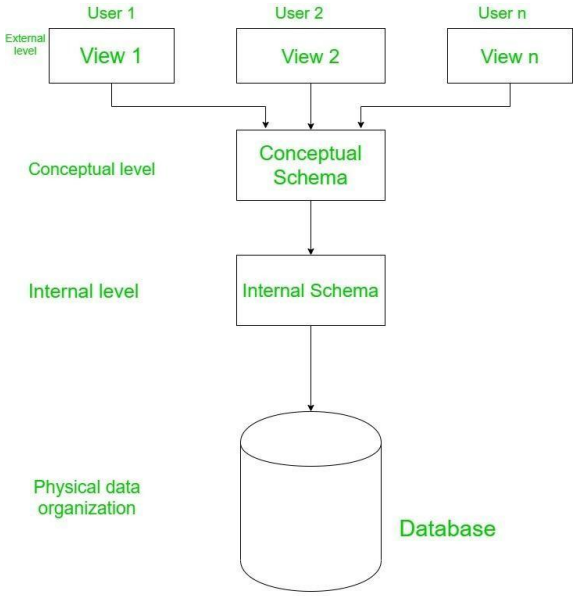
The internal level, also known as the physical level, describes how the data is actually stored on the storage medium. This level includes details about data structures, file organization, indexing, access paths, and data compression techniques. It focuses on the physical storage of data and the methods used to retrieve it efficiently.

For example, the internal level may specify that student records are stored in a particular file format on disk, with indexes created to speed up search operations. This level is crucial for

database performance, as it determines how quickly and efficiently data can be accessed and manipulated.

Diagram of ANSI/SPARC 3-Level Architecture

Here is a simple representation of the ANSI/SPARC 3-level architecture:



The ANSI-SPARC three-level architecture

Importance of ANSI/SPARC 3-Level Architecture

The ANSI/SPARC 3-level architecture is significant for several reasons:

1. **Data Independence**: The architecture promotes data independence, allowing changes to be made at one level without affecting other levels. For example, modifications to the internal storage structure do not impact the conceptual schema or external views.
2. **Simplified Database Design**: By separating the different levels of abstraction, database designers can focus on the logical structure of the data without being concerned about physical storage details. This simplifies the design process and enhances collaboration among stakeholders.
3. **Enhanced Security**: The external level allows for tailored user views, ensuring that users only access the data relevant to them. This enhances security by limiting exposure to sensitive information.
4. **Improved Data Management**: The clear separation of concerns among the three levels facilitates better data management practices. Changes in user requirements or data structures can be accommodated more easily, leading to a more adaptable database system.

The ANSI/SPARC 3-level architecture provides a robust framework for understanding and designing database systems. By separating the external, conceptual, and internal levels, this architecture enhances data independence, simplifies database design, and improves security and data management. As organizations continue to rely on complex databases, the principles of the ANSI/SPARC architecture remain relevant and essential for effective data management.

Q3: Data Independence (Logical & Physical)

Data independence is a fundamental concept in database management systems that refers to the ability to change the schema definition at one level without affecting the schema at the next higher level. This capability is crucial for maintaining the flexibility and adaptability of database systems, allowing organizations to evolve their data structures in response to changing requirements without incurring significant costs or disruptions.

1. Logical Data Independence

Logical data independence is the ability to change the conceptual schema without altering the external schema or application programs. This means that modifications to the logical structure of the database—such as adding new fields, changing data types, or altering relationships—can be made without requiring changes to the user interfaces or application code that interacts with the database.

For example, consider a university database where the conceptual schema includes a table for Students with attributes such as StudentID, Name, and DateOfBirth. If the university decides to add a new attribute, such as EmailAddress, logical data independence allows this change to be made without requiring updates to the applications that access student data. Users will still interact with the same external view, and the applications will continue to function as expected.

The importance of logical data independence lies in its ability to reduce maintenance overhead and enhance the adaptability of database systems. As business requirements evolve, organizations can make necessary changes to the database structure without incurring significant costs or disruptions to ongoing operations.

2. Physical Data Independence

Physical data independence refers to the ability to change the internal schema without affecting the conceptual schema. This means that modifications to the physical storage of data—such as changing file organization, storage devices, or indexing methods—can be made without impacting the logical structure of the database or the applications that rely on it.

For instance, if a database administrator decides to switch from a traditional hard disk drive (HDD) to a solid-state drive (SSD) for improved performance, this change can be implemented at the internal level without requiring any modifications to the conceptual schema or the

applications that access the data. The logical structure remains intact, and users continue to interact with the database as before.

Physical data independence is essential for optimizing database performance and ensuring that the system can adapt to technological advancements. As storage technologies evolve, organizations can implement improvements without disrupting their existing data structures or applications.

Importance of Data Independence

Data independence is vital for several reasons:

1. **Flexibility and Adaptability**: Organizations can respond to changing business needs by modifying their database structures without incurring significant costs or disruptions. This flexibility is crucial in today's fast-paced business environment.
2. **Reduced Maintenance Overhead**: By allowing changes to be made at one level without affecting others, data independence reduces the maintenance burden on database administrators and developers. This leads to more efficient resource allocation and improved productivity.
3. **Improved Data Integrity**: Data independence enhances data integrity by allowing changes to be made in a controlled manner. When modifications are isolated to specific levels, the risk of introducing errors or inconsistencies is minimized.
4. **Simplified Application Development**: Developers can create applications that interact with the database without needing to worry about the underlying physical storage details. This simplifies the development process and allows for faster application deployment.

Data independence—both logical and physical—is a fundamental principle of modern database management systems. By allowing changes to be made at one level without affecting others, data independence enhances flexibility, reduces maintenance overhead, and improves data integrity. As organizations continue to rely on complex databases, the importance of data independence will only grow, making it a critical consideration in database design and management.

Q4: Role of DBA and Different Types of Users

The Database Administrator (DBA) plays a crucial role in managing and maintaining database systems within an organization. The DBA is responsible for ensuring the availability, performance, security, and integrity of the database, making them a key player in the overall data management strategy. In addition to the DBA, there are various types of users who interact with the database, each with distinct roles and responsibilities.

Role of DBA (Database Administrator)

1. **Schema Definition**: The DBA is responsible for defining the logical schema and physical storage structure of the database. This includes designing tables, relationships, and constraints to ensure that the database meets the organization's data requirements.
2. **Authorization & Security**: The DBA manages user access and security by granting permissions and defining user roles. This ensures that only authorized users can access sensitive data, protecting the organization from potential security breaches.
3. **Performance Tuning**: The DBA is tasked with optimizing database performance through various techniques, such as indexing, query optimization, and resource allocation. By monitoring performance metrics and making necessary adjustments, the DBA ensures that the database operates efficiently.
4. **Backup & Recovery**: The DBA is responsible for implementing backup and recovery strategies to protect data from loss or corruption. This includes scheduling regular backups, maintaining logs of changes, and developing recovery plans to restore the database in case of failure.
5. **Data Integrity Enforcement**: The DBA enforces data integrity by applying constraints and validation rules at the schema level. This ensures that the data remains accurate and consistent, adhering to predefined standards.
6. **Database Maintenance**: The DBA performs routine maintenance tasks, such as updating software, applying patches, and monitoring system health. This proactive approach helps prevent issues before they impact database performance.

Types of Database Users

1. **End Users**: End users are individuals who interact with the database through applications. They may include employees, customers, or clients who use the database to perform specific tasks, such as querying data, generating reports, or updating records. End users typically do not have direct access to the database but interact with it through user-friendly interfaces.
2. **Application Programmers**: Application programmers are responsible for writing application programs that interact with the database. They use programming languages and database APIs to develop software that retrieves, manipulates, and displays data. Programmers must have a good understanding of the database schema and query languages to create effective applications.
3. **Database Designers**: Database designers are responsible for designing the database structure and schema. They work closely with stakeholders to understand data requirements and create a logical model that meets those needs. Database designers

focus on defining entities, attributes, relationships, and constraints to ensure that the database is well-structured and efficient.

4. **DBA (Database Administrator)**: As previously discussed, the DBA is responsible for the overall management of the database system. They ensure that the database is secure, performant, and reliable, serving as the primary point of contact for database-related issues.

The role of the Database Administrator is critical to the success of database management within an organization. The DBA is responsible for schema definition, security, performance tuning, backup and recovery, data integrity enforcement, and routine maintenance. Additionally, various types of users, including end users, application programmers, database designers, and DBAs, interact with the database, each contributing to the overall data management strategy. Understanding these roles and responsibilities is essential for effective database management and ensuring that data is utilized efficiently and securely.

Q5: Database System Architecture (Basic Block Diagram)

Database system architecture refers to the structure and organization of a database management system (DBMS). It encompasses the various components that work together to manage data effectively. A well-designed database architecture is crucial for ensuring data integrity, security, and performance. This section outlines the basic components of a database system architecture and provides a block diagram for visual representation.

Components of Database System Architecture

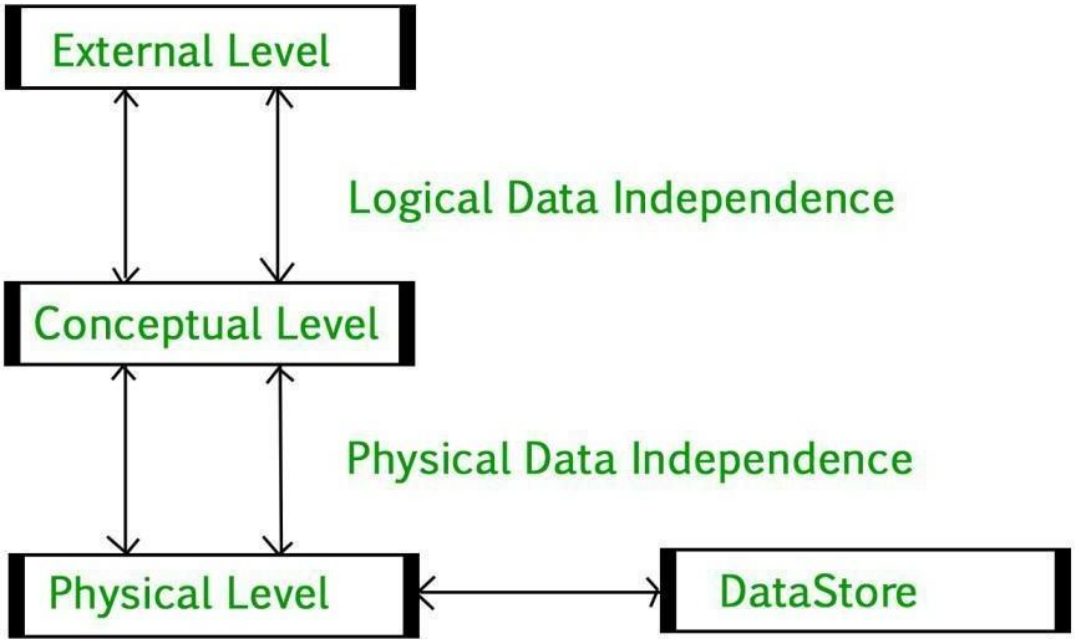
1. **Users/Applications**: At the top of the architecture are the users and applications that interact with the database. Users may include end users, application programmers, and database administrators. Applications are software programs that utilize the database to perform specific tasks, such as data entry, reporting, and analysis.
2. **Query Processor**: The query processor is responsible for interpreting and executing database queries. It translates user requests into a format that the DBMS can understand and process. The query processor includes several subcomponents:
    - **DDL Compiler**: The Data Definition Language (DDL) compiler processes schema definitions and creates the database structure.
    - **DML Compiler**: The Data Manipulation Language (DML) compiler processes queries that manipulate data, such as insertions, updates, and deletions.
    - **Query Optimizer**: The query optimizer analyzes queries and determines the most efficient execution plan, considering factors such as available indexes and data distribution.

- **Transaction Manager**: The transaction manager ensures that database transactions are executed reliably and adhere to ACID (Atomicity, Consistency, Isolation, Durability) properties.
- **Authorization Manager**: The authorization manager handles user access control and permissions, ensuring that only authorized users can perform specific actions on the database.

3. **DBMS Engine**: The DBMS engine is the core component of the database system, responsible for managing data storage, retrieval, and manipulation. It interacts with the query processor and storage manager to execute queries and manage data efficiently.
4. **Storage Manager**: The storage manager is responsible for managing the physical storage of data. It includes several subcomponents:
    - **File Manager**: The file manager handles the organization and storage of data files on disk, ensuring efficient access and retrieval.
    - **Buffer Manager**: The buffer manager manages memory allocation for data caching, optimizing data access and reducing disk I/O operations.
    - **Recovery Manager**: The recovery manager ensures data durability and consistency by implementing backup and recovery strategies, allowing the system to restore data in case of failures.
5. **Physical Database**: The physical database is the actual data stored on disk. It includes all the data files, indexes, and other structures necessary for the DBMS to function. The physical database is managed by the storage manager and is accessed through the DBMS engine.

Basic Block Diagram of Database System Architecture

Here is a simple representation of the basic block diagram of a database system architecture:



Importance of Database System Architecture

1. **Data Integrity**: A well-defined architecture ensures that data integrity is maintained throughout the database system. By enforcing constraints and validation rules at various levels, the architecture helps prevent data anomalies.
2. **Performance Optimization**: The architecture allows for performance tuning through components like the query optimizer and buffer manager. By optimizing query execution plans and managing memory effectively, the system can handle large volumes of data efficiently.
3. **Scalability**: A modular architecture enables the database system to scale as the organization grows. New components can be added or existing ones modified without disrupting the overall system.
4. **Security**: The architecture incorporates security measures at multiple levels, ensuring that data is protected from unauthorized access. The authorization manager plays a crucial role in defining user roles and permissions.
5. **Ease of Maintenance**: A clear architecture simplifies maintenance tasks, allowing database administrators to identify and resolve issues more efficiently. Routine tasks such as backups, updates, and performance monitoring can be managed effectively.

The database system architecture is a critical aspect of database management that encompasses various components working together to manage data effectively. By understanding the roles of users, the query processor, DBMS engine, storage manager, and physical database, organizations can design robust and efficient database systems. A well-structured architecture enhances data integrity, performance, scalability, security, and ease of maintenance, making it essential for successful data management in any organization.

UNIT 2 : SQL

Q1: SQL Command Types (DDL, DML, DCL, TCL)

SQL (Structured Query Language) is the standard language used for managing and manipulating relational databases. It consists of several command types, each serving a specific purpose in database management. The four primary categories of SQL commands are Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), and Transaction Control Language (TCL). Understanding these command types is essential for effective database management and manipulation.

1. Data Definition Language (DDL)

Data Definition Language (DDL) commands are used to define and manage the structure of database objects such as tables, indexes, and schemas. DDL commands allow users to create, alter, and drop database objects. The key DDL commands include:

- **CREATE**: This command is used to create new database objects. For example, to create a new table, the syntax is as follows:

```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  FirstName VARCHAR(50),
  LastName VARCHAR(50),
  HireDate DATE
);
```

- **ALTER**: This command modifies the structure of an existing database object. For example, to add a new column to the Employees table:

```
ALTER TABLE Employees
ADD Email VARCHAR(100);
```

- **DROP**: This command removes an existing database object. For example, to delete the Employees table:

```
DROP TABLE Employees;
```

DDL commands are crucial for establishing the database schema and ensuring that the database structure aligns with the organization's data requirements.

2. Data Manipulation Language (DML)

Data Manipulation Language (DML) commands are used to manipulate and manage the data stored in database objects. DML commands allow users to insert, update, delete, and retrieve data. The key DML commands include:

- **INSERT**: This command adds new records to a table. For example, to insert a new employee record:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate)

VALUES (1, 'John', 'Doe', '2023-01-15');
```

- **UPDATE**: This command modifies existing records in a table. For example, to update an employee's last name:

```
UPDATE Employees

SET LastName = 'Mohit'

WHERE EmployeeID = 1;
```

- **DELETE**: This command removes records from a table. For example, to delete an employee record:

```
DELETE FROM Employees

WHERE EmployeeID = 1;
```

DML commands are essential for managing the data within the database, allowing users to perform CRUD (Create, Read, Update, Delete) operations.

3. Data Control Language (DCL)

Data Control Language (DCL) commands are used to control access to data within the database. DCL commands allow administrators to grant or revoke permissions to users and roles. The key DCL commands include:

- **GRANT**: This command gives specific privileges to users or roles. For example, to grant SELECT permission on the Employees table to a user:

```
GRANT SELECT ON Employees TO UserName;
```

- **REVOKE**: This command removes specific privileges from users or roles. For example, to revoke SELECT permission from a user:

```
REVOKE SELECT ON Employees FROM UserName;
```

DCL commands are crucial for maintaining data security and ensuring that only authorized users can access or manipulate sensitive data.

4. Transaction Control Language (TCL)

Transaction Control Language (TCL) commands are used to manage transactions in a database. Transactions are sequences of operations performed as a single logical unit of work. The key TCL commands include:

- **COMMIT**: This command saves all changes made during the current transaction. For example:

COMMIT;

- **ROLLBACK**: This command undoes all changes made during the current transaction. For example:

ROLLBACK;

- **SAVEPOINT**: This command creates a point within a transaction to which you can later roll back. For example:

SAVEPOINT SavePointName;

TCL commands are essential for ensuring data integrity and consistency, especially in multi-user environments where concurrent transactions may occur.

SQL commands are categorized into four main types: DDL, DML, DCL, and TCL. Each command type serves a specific purpose in database management, from defining the structure of database objects to manipulating data and controlling access. Understanding these command types is fundamental for effectively managing relational databases and ensuring data integrity, security, and performance.

Q2: Syntax and Examples of SQL Commands

SQL provides a rich set of commands for managing and manipulating data in relational databases. This section covers the syntax and examples of key SQL commands, including CREATE, INSERT, UPDATE, DELETE, SELECT, GRANT, and REVOKE.

1. CREATE

The CREATE command is used to create new database objects, such as tables, views, and indexes. The syntax for creating a table is as follows:

```
CREATE TABLE TableName (
    Column1 DataType Constraints,
    Column2 DataType Constraints,
    ...
);
```

Example: Creating a table for storing employee information:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    HireDate DATE,
    Salary DECIMAL(10, 2)
);
```

2. INSERT

The INSERT command is used to add new records to a table. The syntax for inserting data is as follows:

```
INSERT INTO TableName (Column1, Column2, ...)
VALUES (Value1, Value2, ...);
```

Example: Inserting a new employee record:

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate, Salary)
VALUES (1, 'John', 'Doe', '2023-01-15', 60000.00);
```

3. UPDATE

The UPDATE command is used to modify existing records in a table. The syntax for updating data is as follows:

```
UPDATE TableName
SET Column1 = Value1, Column2 = Value2, ...
WHERE Condition;
```

Example: Updating the salary of an employee:

```
UPDATE Employees
SET Salary = 65000.00
WHERE EmployeeID = 1;
```

4. DELETE

The DELETE command is used to remove records from a table. The syntax for deleting data is as follows:

```
DELETE FROM TableName

WHERE Condition;
```

Example: Deleting an employee record:

```
DELETE FROM Employees

WHERE EmployeeID = 1;
```

5. SELECT

The SELECT command is used to retrieve data from one or more tables. The syntax for selecting data is as follows:

```
SELECT Column1, Column2, ...

FROM TableName

WHERE Condition

ORDER BY Column1 [ASC|DESC];
```

Example: Selecting all employees and sorting by last name:

```
SELECT FirstName, LastName, Salary

FROM Employees

ORDER BY LastName ASC;
```

6. GRANT

The GRANT command is used to give specific privileges to users or roles. The syntax for granting permissions is as follows:

```
GRANT Privilege ON Object TO UserName;
```

Example: Granting SELECT permission on the Employees table to a user:

```
GRANT SELECT ON Employees TO UserName;
```

7. REVOKE

The REVOKE command is used to remove specific privileges from users or roles. The syntax for revoking permissions is as follows:

```
REVOKE Privilege ON Object FROM UserName;
```

Example: Revoking SELECT permission from a user:

```
REVOKE SELECT ON Employees FROM UserName;
```

SQL provides a comprehensive set of commands for managing and manipulating data in relational databases. The commands discussed in this section—CREATE, INSERT, UPDATE, DELETE, SELECT, GRANT, and REVOKE—are fundamental for performing various database operations. Understanding the syntax and usage of these commands is essential for effective database management and data manipulation.

Q3: Use of WHERE, BETWEEN, LIKE, IN, NOT IN, Logical Operators

SQL provides various operators and clauses that allow users to filter and manipulate data effectively. This section discusses the use of the WHERE clause, as well as operators such as BETWEEN, LIKE, IN, NOT IN, and logical operators.

1. WHERE Clause

The WHERE clause is used to filter records based on specified conditions. It is commonly used in SELECT, UPDATE, and DELETE statements to restrict the number of rows affected by the query.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE Condition;

Example: Selecting employees with a salary greater than 50000:

SELECT FirstName, LastName, Salary

FROM Employees

WHERE Salary > 50000;

2. BETWEEN Operator

The BETWEEN operator is used to filter records within a specified range. It can be used with numeric, date, and string values.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE ColumnName BETWEEN Value1 AND Value2;

Example: Selecting employees hired between January 1, 2023, and December 31, 2023:

SELECT FirstName, LastName, HireDate

FROM Employees

WHERE HireDate BETWEEN '2023-01-01' AND '2023-12-31';

3. LIKE Operator

The LIKE operator is used to search for a specified pattern in a column. It is often used with wildcard characters:

- **%**: Represents zero or more characters.
- **_**: Represents a single character.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE ColumnName LIKE 'Pattern';

Example: Selecting employees whose last name starts with 'S':

SELECT FirstName, LastName

FROM Employees

WHERE LastName LIKE 'S%';

4. IN Operator

The IN operator is used to filter records based on a list of specified values. It allows users to specify multiple values in a WHERE clause.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE ColumnName IN (Value1, Value2, …);

Example: Selecting employees with specific EmployeeIDs:

SELECT FirstName, LastName

FROM Employees

WHERE EmployeeID IN (1, 2, 3);

5. NOT IN Operator

The NOT IN operator is used to filter records that do not match any of the specified values in a list.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE ColumnName NOT IN (Value1, Value2, …);

Example: Selecting employees whose EmployeeID is not in a specified list:

SELECT FirstName, LastName

FROM Employees

WHERE EmployeeID NOT IN (1, 2, 3);

6. Logical Operators

Logical operators are used to combine multiple conditions in a WHERE clause. The primary logical operators are:

- **AND**: Returns true if both conditions are true.
- **OR**: Returns true if at least one condition is true.
- **NOT**: Reverses the result of a condition.

Syntax:

SELECT Column1, Column2, ...

FROM TableName

WHERE Condition1 AND/OR Condition2;

Example: Selecting employees with a salary greater than 50000 and hired after January 1, 2023:

SELECT FirstName, LastName, Salary

FROM Employees

WHERE Salary > 50000 AND HireDate > '2023-01-01';

SQL provides various operators and clauses that enhance data filtering and manipulation capabilities. The WHERE clause, along with operators such as BETWEEN, LIKE, IN, NOT IN, and logical operators, allows users to construct complex queries to retrieve specific data from relational databases. Understanding how to use these operators effectively is essential for efficient data management and analysis.

Q4: Aggregate Functions (SUM, COUNT, etc.)

Aggregate functions in SQL are used to perform calculations on a set of values and return a single value. These functions are commonly used in conjunction with the GROUP BY clause to summarize data. This section discusses several key aggregate functions, including SUM, COUNT, AVG, MIN, and MAX, along with their syntax and examples.

1. SUM Function

The SUM function calculates the total sum of a numeric column.

Syntax:

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

Example: Calculating the total salary of all employees:

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

2. COUNT Function

The COUNT function returns the number of rows that match a specified condition. It can count all rows or only distinct values.

Syntax:

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

Example: Counting the total number of employees:

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

Example with DISTINCT: Counting the number of distinct job titles:

```
SELECT COUNT(DISTINCT JobTitle) AS UniqueJobTitles
FROM Employees;
```

3. AVG Function

The AVG function calculates the average value of a numeric column.

Syntax:

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

Example: Calculating the average salary of employees:

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

4. MIN Function

The MIN function returns the smallest value in a specified column.

Syntax:

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

Example: Finding the minimum salary among employees:

```
SELECT MIN(Salary) AS MinimumSalary
FROM Employees;
```

5. MAX Function

The MAX function returns the largest value in a specified column.

Syntax:

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

Example: Finding the maximum salary among employees:

```
SELECT MAX(Salary) AS MaximumSalary
FROM Employees;
```

Using Aggregate Functions with GROUP BY

Aggregate functions are often used with the GROUP BY clause to group rows that have the same values in specified columns. This allows for summarizing data based on specific categories.

Syntax:

```
SELECT Column1, AGGREGATE_FUNCTION(Column2)
FROM TableName
GROUP BY Column1;
```

Example: Calculating the total salary for each department:

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

Aggregate functions are powerful tools in SQL that allow users to perform calculations on sets of values and summarize data effectively. Functions such as SUM, COUNT, AVG, MIN, and MAX enable users to derive meaningful insights from their data. When combined with the GROUP BY clause, aggregate functions provide a robust mechanism for analyzing and reporting on data in relational databases.

Q5: SQL Queries with Conditions, Sorting, and Functions

SQL queries can be constructed to retrieve and manipulate data based on specific conditions, sorting criteria, and functions. This section discusses how to create SQL queries that incorporate conditions, sorting, and various functions to enhance data retrieval and analysis.

1. SQL Queries with Conditions

Conditions in SQL queries are specified using the WHERE clause, allowing users to filter records based on specific criteria. Conditions can include comparisons, logical operators, and various SQL functions.

Example: Selecting employees with a salary greater than 50000 and hired after January 1, 2023:

```
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary > 50000 AND HireDate > '2023-01-01';
```

2. Sorting Results

SQL provides the ORDER BY clause to sort the results of a query based on one or more columns. The sorting can be done in ascending (ASC) or descending (DESC) order.

Syntax:

```
SELECT Column1, Column2, ...
```

FROM TableName

ORDER BY Column1 [ASC|DESC], Column2 [ASC|DESC];

Example: Selecting all employees and sorting by last name in ascending order:

SELECT FirstName, LastName, Salary

FROM Employees

ORDER BY LastName ASC;

Example with Multiple Columns: Sorting by department and then by salary in descending order:

SELECT FirstName, LastName, Department, Salary

FROM Employees

ORDER BY Department ASC, Salary DESC;

3. Using Functions in Queries

SQL functions can be used in queries to perform calculations or manipulate data. Common functions include aggregate functions (e.g., SUM, COUNT) and string functions (e.g., UPPER, LOWER).

Example: Calculating the total salary of employees in each department and sorting by total salary:

SELECT Department, SUM(Salary) AS TotalSalary

FROM Employees

GROUP BY Department

ORDER BY TotalSalary DESC;

Example with String Functions: Selecting employees and converting their first names to uppercase:

SELECT UPPER(FirstName) AS UpperFirstName, LastName

FROM Employees;

4. Combining Conditions and Functions

SQL queries can combine conditions, sorting, and functions to create complex data retrieval scenarios. This allows users to extract meaningful insights from their data.

Example: Selecting departments with an average salary greater than 60000 and sorting by average salary:

```
SELECT Department, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY Department
HAVING AVG(Salary) > 60000
ORDER BY AverageSalary DESC;
```

SQL queries can be constructed to incorporate conditions, sorting, and functions to enhance data retrieval and analysis. The WHERE clause allows for filtering records based on specific criteria, while the ORDER BY clause enables sorting of results. Additionally, SQL functions can be used to perform calculations and manipulate data effectively. Understanding how to create and utilize these queries is essential for effective data management and analysis in relational databases.

UNIT 3: DATA MODELS & ER DIAGRAM

Q1. Types of Data Models (Hierarchical, Network, Relational, OOP)

Types of database models

There are so many database models, but most used database models are −

- Hierarchical database model
- Relational model
- Network model
- Object-oriented database model

The major differences between the hierarchical, network and the relational models are as follows −

Hierarchical Model	**Network Model**	**Relational Model**
One to many or one to one relationship.	Allowed the network mode to support many to many relationships.	One to one, one to many, many to one relationship.
Retrieve algorithms are complex and asymmetric.	Retrieve algorithms are complex and symmetric.	Retrieve algorithms are simple and symmetric.
Based on parent child relationship.	A record can have many parents as well as many children.	Based on relational data structures.
Doesn't provide an independent stand alone query interface.	Conference on data system language.	Relational databases are what bring many sources into a common query such as SQL.
Cannot insert the information of a child who does not have any parent.	Does not suffer from any insertion anomaly.	Doesn't suffer from any insertion anomaly.
Multiple occurrences of child records which lead to problems of inconsistency during the update operation.	Free from update anomalies.	Free from update anomalies.
Deletion of parent results in deletion of child record.	Free from delete anomalies.	Free from delete anomalies.

This model lacks data independence.	There is partial data independence.	It provides data independence.
Less flexible.	Flexible.	Flexible.
Difficult to access data.	Easier to access data.	Easier to access data.
Arrange data in a tree-like structure.	Organizes data in graph-like structure.	Arrange data in tables.

Q2 . Components of E-R Diagram: Entities, Attributes, Relationships

ER model stands for the Entity Relationship Model in the **database management system** (DBMS). It is the first step of designing to give the flow for a concept. It is the DFD (Data Flow Diagram) requirement of a company.

It is the basic building block for relational models. Not that much training is required to design the database project. It is very easy to convert the E-R model into a relational table or to a normalized table.

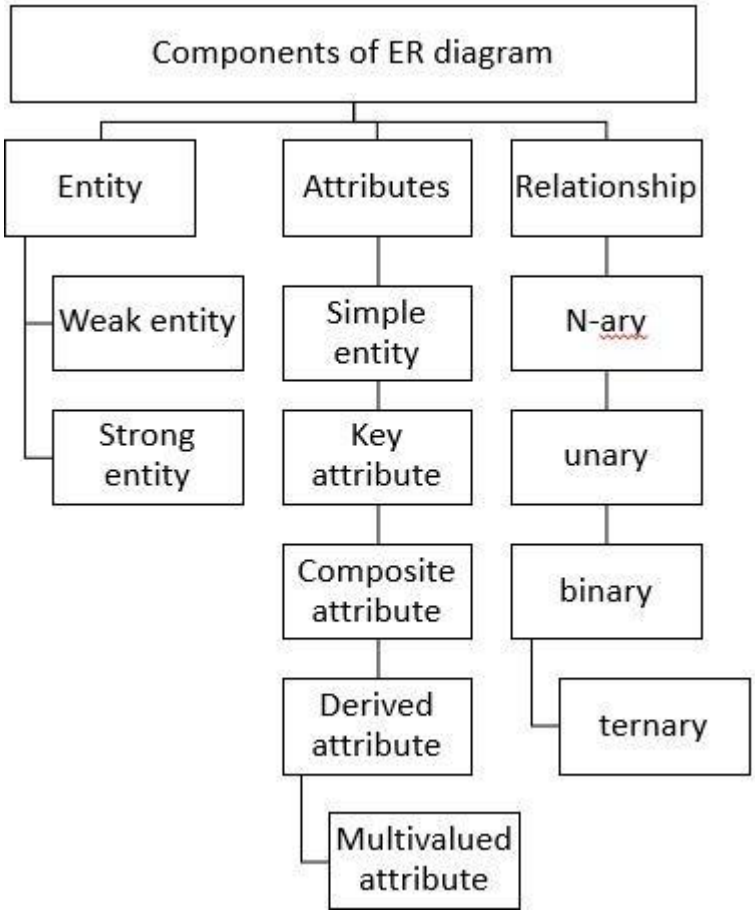
It is a high-level data model diagram that defines the conceptual view of the database. It acts as a blueprint to implement a database in future.

Components of ER diagram

The components of ER diagram are as follows ?

- Entity
- Attributes
- Relationship
- Weak entity
- Strong entity
- Simple attribute
- Key attribute
- Composite attribute
- Derived attribute
- Multivalued attribute

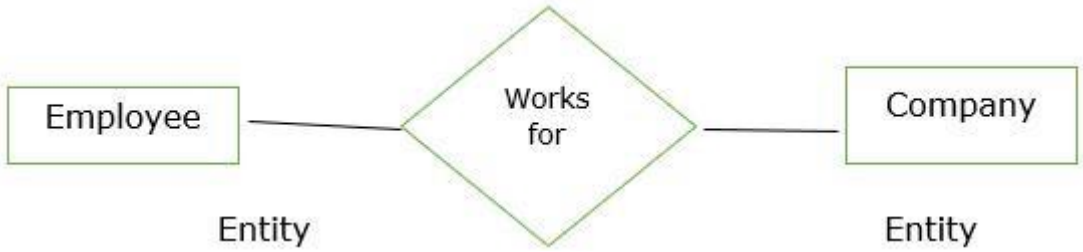
The components of the ER diagram are pictorially represented as follows ?



Entity

It may be an object, person, place or event that stores data in a database. In a relationship diagram an entity is represented in rectangle form. For example, students, employees, managers, etc.

The entity is pictorially depicted as follows ?



Entity set

It is a collection of entities of the same type which share similar properties. For example, a group of students in a college and students are an entity set.

Entity is characterised into two types as follows ?

- Strong entity set
- Weak entity set

Strong entity set

The entity types which consist of key attributes or if there are enough attributes for forming a primary key attribute are called a strong entity set. It is represented by a single rectangle.

For Example,

Roll no of student

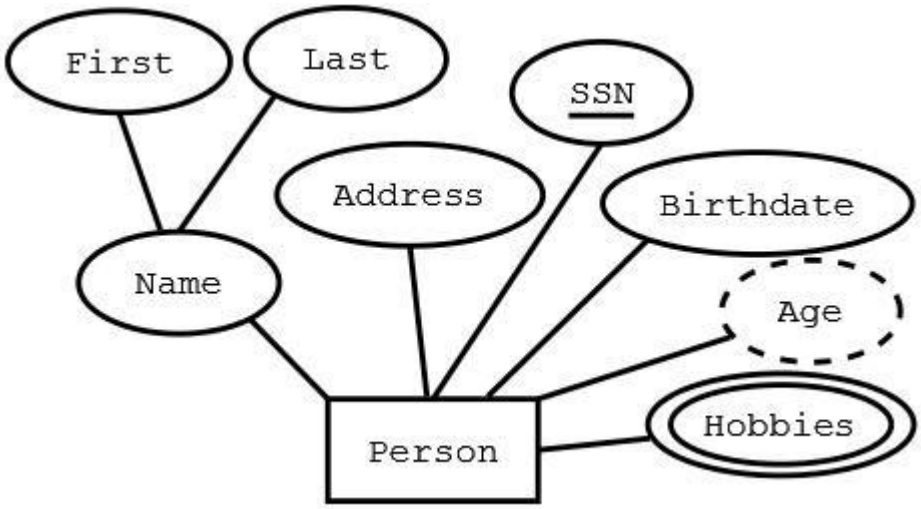
EmpID of employee

Weak entity set

An entity does not have a primary key attribute and depends on another strong entity via foreign key attribute. It is represented by a double rectangle.

Attributes

It is the name, thing etc. These are the data characteristics of entities or data elements and data fields.

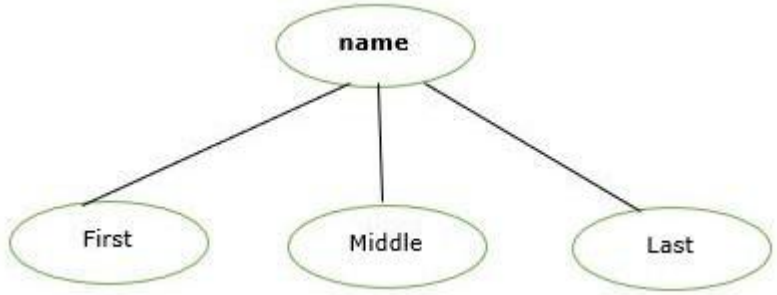


Types of attributes

The types of attributes in the Entity Relationship (ER) model are as follows ?

- **Single value attribute** ? These attributes contain a single value. For example, age, salary etc.
- **Multivalued attribute** ? They contain more than one value of a single entity. For example, phone numbers.
- **Composite attribute** ? The attributes which can be further divided. For example, Name-> First name, Middle name, last name

- **Derived attribute** ? The attribute that can be derived from others. For example, Date of Birth.

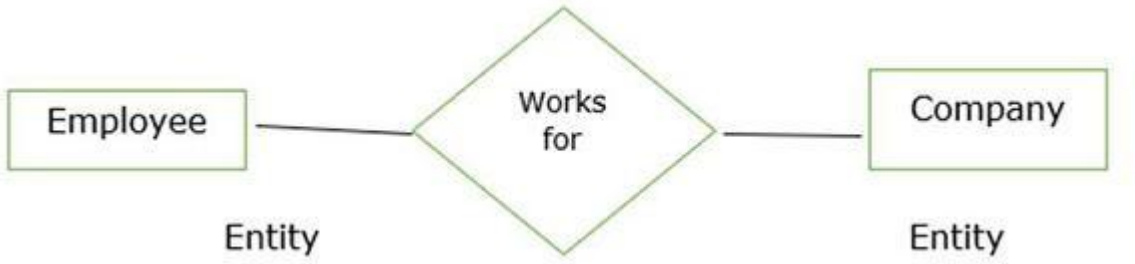


Relationship

It is used to describe the relation between two or more entities. It is represented by a diamond shape.

For Example, students study in college and employees work in a department.

The relationship is pictorially represented as follows ?



Here works for is a relation between two entities.

Degree of Relationship

A relationship where a number of different entities set participate is called a degree of a relationship.

It is categorised into the following ?

- Unary Relationship
- Binary Relationship
- Ternary Relationship
- n-ary Relationship

Q3 . Specialization, Generalization, Aggregation

1. Generalisation

2. Specialisation
3. Aggregation

We will learn more about them below –

What is Generalization

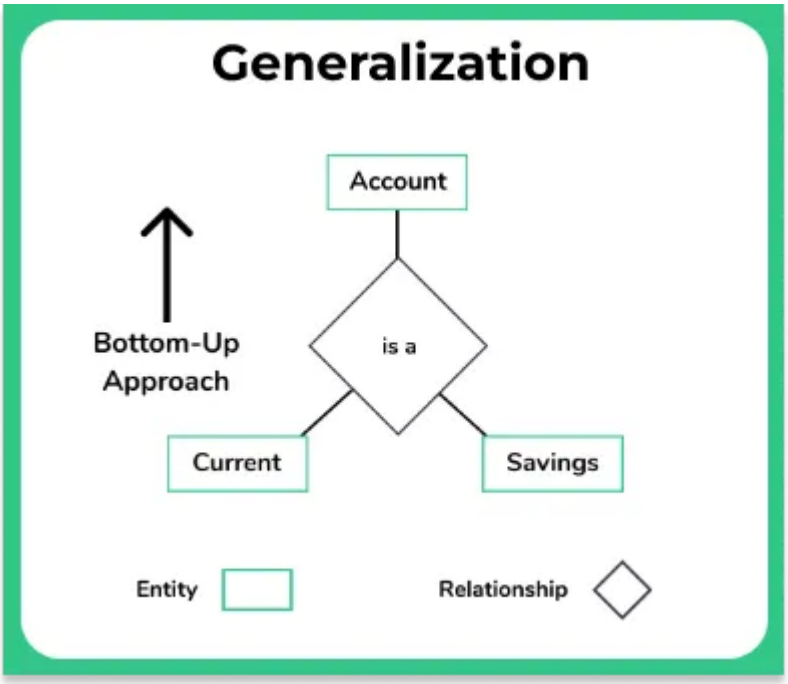
In generalization we combine lower level entities to form a higher level entity. Thus its clear that it follows a bottom up approach.

Example –

In a bank there are two different types of accounts – Current and Savings, combine to form a super entity Account.

It thus follows system like classes, like super-classes and sub-classes right ?

It may also be possible that the higher level entity may also combine with further entity to form a one more higher level entity.

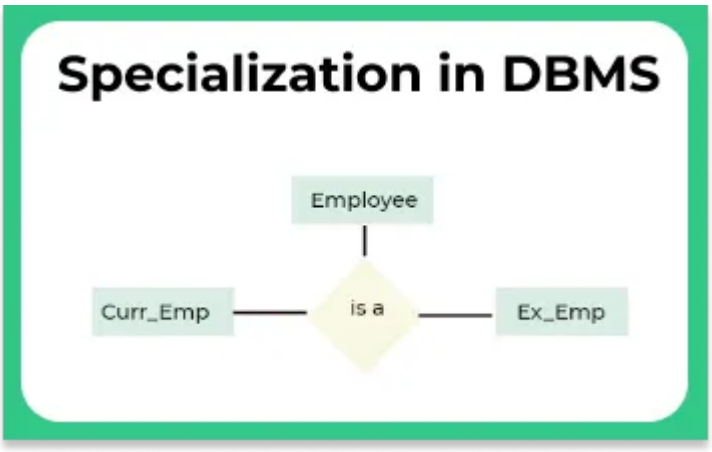


What is Specialization

While generalization may follow a bottom up approach. **Specialization** is opposite to that, it follows a top down approach rather.

Example –

Employee may be decomposed to further as current employee entity and ex employee entity.

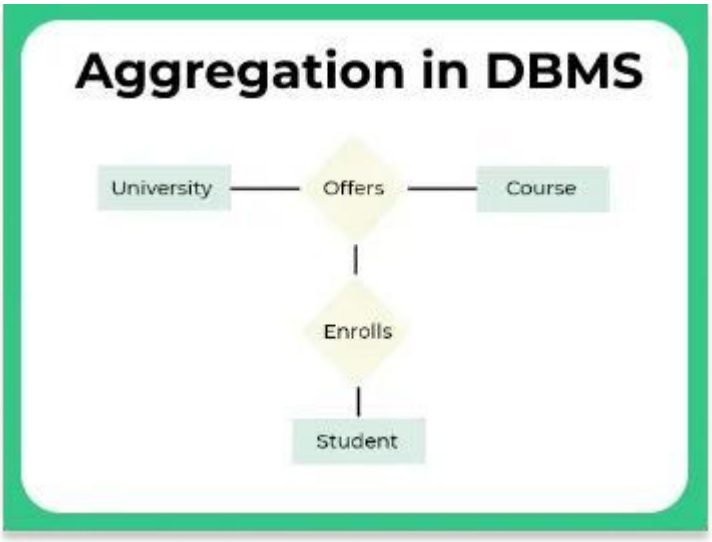


What is Aggregation

Aggregation is simply when we would consider two different entities as a single entity together.

Example –

University offering course can be considered a same entity, when viewed from a student entity perspective.



Q4. Weak Entity Sets

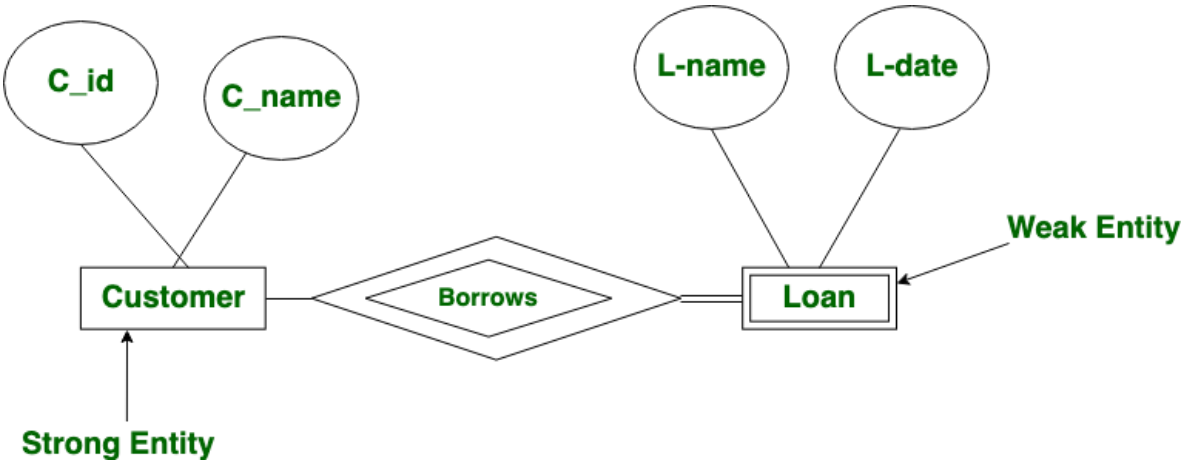
An entity type should have a key attribute which uniquely identifies each entity in the entity set, but there exists some entity type for which key attribute can't be defined. These are called Weak Entity type.

The entity sets which do not have sufficient attributes to form a primary key are known as **weak entity sets** and the entity sets which have a primary key are known as strong entity sets.

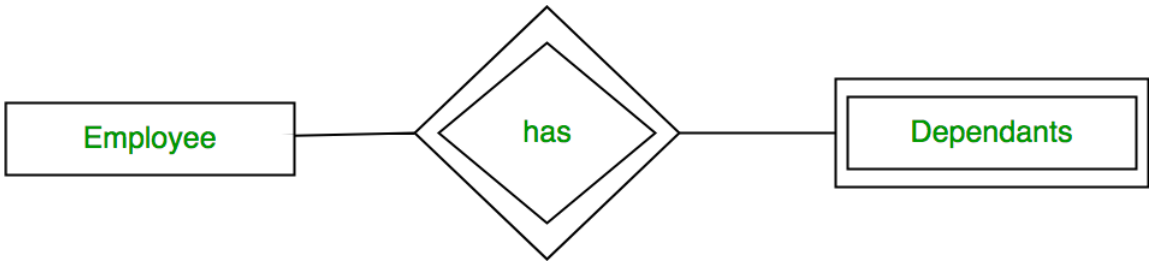
As the weak entities do not have any primary key, they cannot be identified on their own, so they depend on some other entity (known as owner entity). The weak entities have total participation constraint (existence dependency) in its identifying relationship with owner identity. Weak entity types have partial keys. Partial Keys are set of attributes with the help of which the tuples of the weak entities can be distinguished and identified.

Note - Weak entity always has total participation but Strong entity may not have total participation.

Weak entity is **depend on strong entity** to ensure the existence of weak entity. Like strong entity, weak entity does not have any primary key, It has partial discriminator key. Weak entity is represented by double rectangle. The relation between one strong and one weak entity is represented by double diamond.

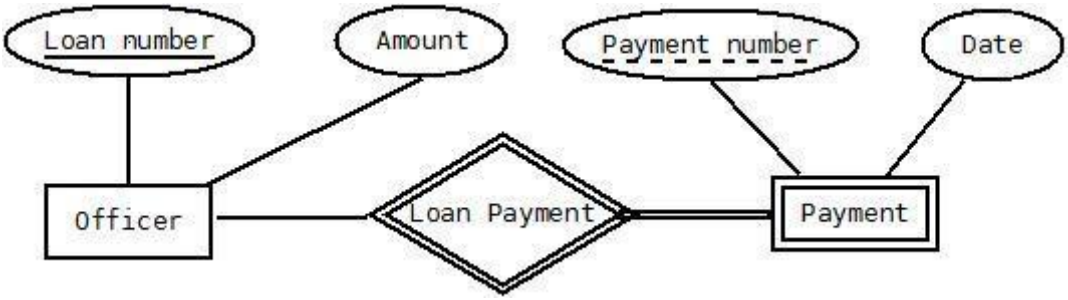


Weak entities are represented with **double rectangular** box in the ER Diagram and the identifying relationships are represented with double diamond. Partial Key attributes are represented with dotted lines.



Example-1:

In the below ER Diagram, 'Payment' is the weak entity. 'Loan Payment' is the identifying relationship and 'Payment Number' is the partial key. Primary Key of the Loan along with the partial key would be used to identify the records.



Example-2:

The existence of rooms is entirely dependent on the existence of a hotel. So room can be seen as the weak entity of the hotel.

Example-3:

The bank account of a particular bank has no existence if the bank doesn't exist anymore.

Example-4:

A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents don't have existence without the employee. So Dependent will be weak entity type and Employee will be Identifying Entity type for Dependent.

Other examples:

Strong entity | Weak entity
Order | Order Item
Employee | Dependent
Class | Section
Host | Logins

Q5 . Drawing ER diagram for a scenario

Problem

Drawing of **ER model** of university database application considering the constraints −

- A university has many departments.
- Each department has multiple instructors (one person is HOD). Here the HOD refers to the head of department.
- An instructor belongs to only one department.
- Each department offers multiple courses, each subject is taught by a single instructor.
- A student may enroll for many courses offered by different departments.

Solution

Follow the steps given below to draw an **Entity Relationship (ER)** diagram for a University database application −

Step 1 − Identifying the entity sets.

The entity set has multiple instances in a given business scenario.

As per the given constraints the entity sets are as follows −

- Department
- Course
- Student
- Instructor

Head of the Department (HOD) is not an entity set. It is a relationship between the instructor and department entities.

Step 2 − Identifying the attributes for the given entities

- Department − the relevant attributes are department Name and location.
- Course − The relevant attributes are courseNo, course Name, Duration, and prerequisite.
- Instructor − The relevant attributes are Instructor Name, Room No, and telephone number.
- Student − The relevant attributes are Student No, Student Name, and date of birth.

Step 3 − Identifying the Key attributes

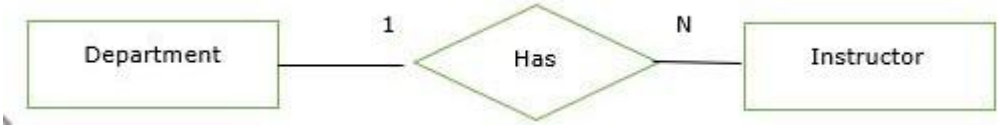
- Department Name is the key attribute for Department.
- CourseNo is the key attribute for Course entity.
- Instructor Name is the key attribute for the Instructor entity.
- StudentNo is the key attribute for Student entities.

Step 4 – Identifying the relationship between entity sets

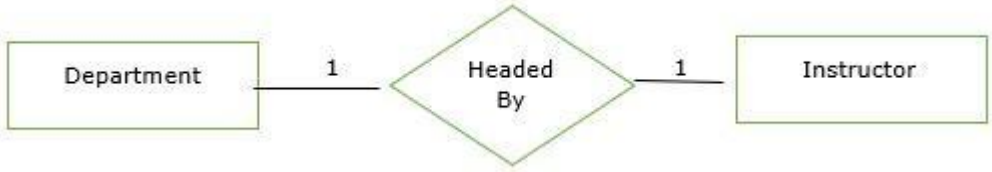
- The department offers multiple courses and each course belongs to only one department, hence cardinality between department and course if one to many.

- One course is enrolled by multiple students and one student for multiple courses. Hence, relationships are many to many.

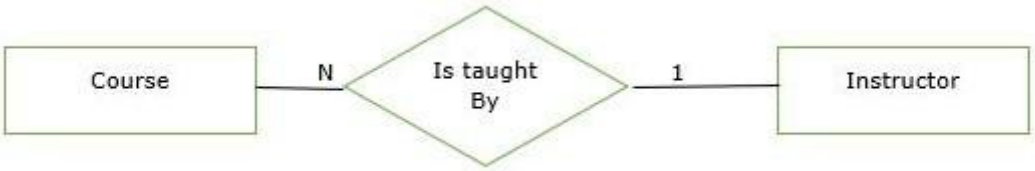
- One department has multiple instructors and one instructor belongs to one and only one department, hence the relationship is one to many.



- Each department has one “HOD” and one instructor is “HOD” for only one department, hence the relationship is one to one. Here, HOD refers to the head of the department.



- One course is taught by only one instructor but one instructor teaches many courses hence the relationship between course and instructor is many to one.

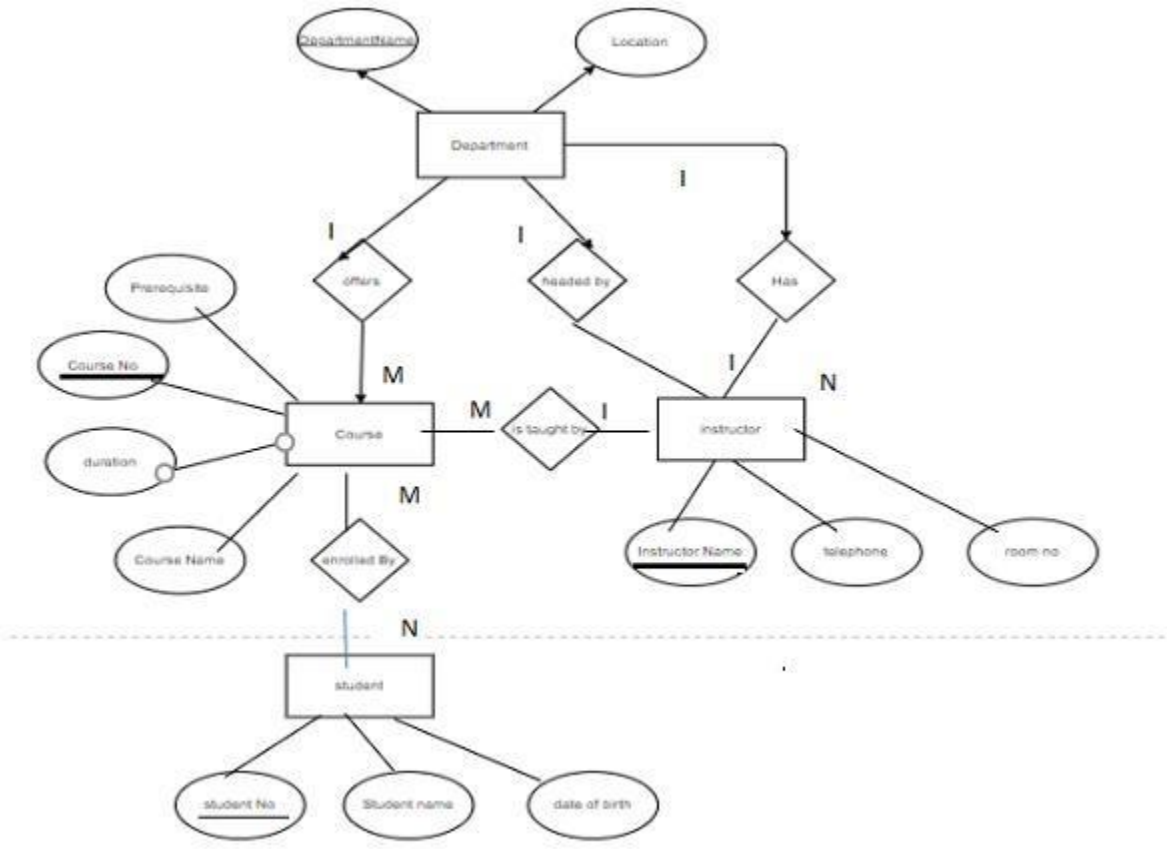


The relationship between instructor and student is not defined because of the following reasons −

- There is no significance in the relationship.
- We can always derive this relationship indirectly through course and instructors, and course and student.

Step 5 − Complete ER model

The complete ER Model is as follows −



CH-4 RELATIONAL DATA MODEL

Q1 Relational Model terms: Tuple, Attribute, Degree, Cardinality

Relation:

- A relation is a two-dimensional table i.e the information is arranged in rows and columns.
- It is called a relation because the data values in the table are not homogenous i.e not if the same type or we can say that, the values in a row are not homogenous.

Domain:

- The term domain refers to the current set of values found under an attribute name.
- It is the values under a column.

Tuple: It is a row. One row in a table is known as a tuple.

Attribute: It is a column of a table.

Degree: Number of columns in a table.

Cardinality: Number of rows in a table.

Q2 Key types: Primary, Foreign, Candidate, Super Key

Why do we require Keys in a DBMS?

Keys are important in a Database Management System (DBMS) for several reasons:

- Uniqueness: Keys ensure that each record in a table is unique and can be identified distinctly.
- Data Integrity: Keys prevent data duplication and maintain the consistency of the data.
- Efficient Data Retrieval: By defining relationships between tables, keys enable faster querying and better data organization. Without keys, it would be extremely difficult to manage large datasets and queries would become inefficient and prone to errors.

Types of Database Keys

1. Super Key

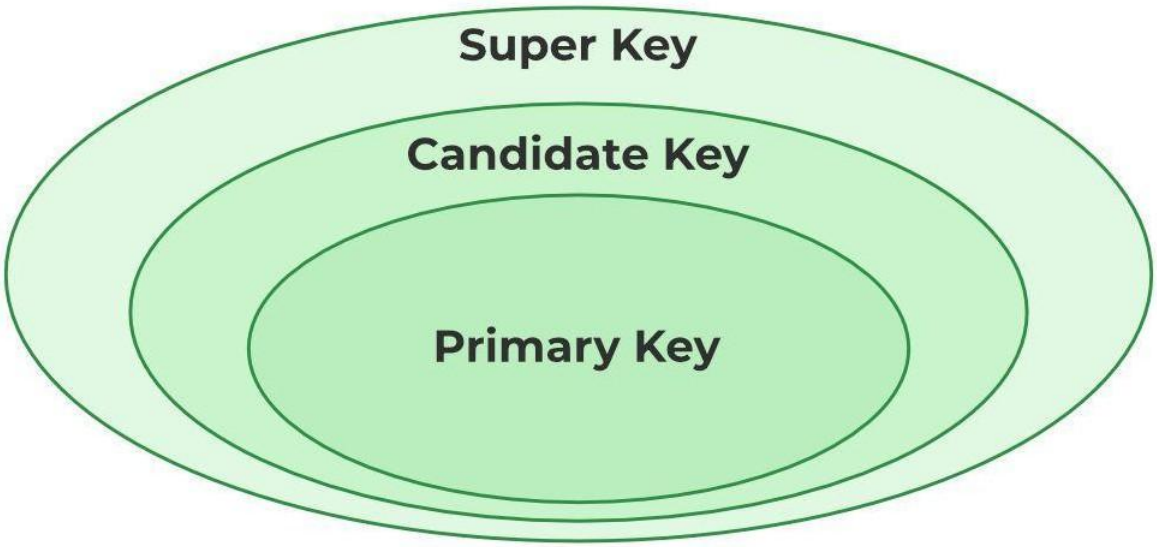
The set of one or more attributes (columns) that can uniquely identify a tuple (record) is known as Super Key. It may include extra attributes that aren't important for uniqueness but still uniquely identify the row. For Example, STUD_NO, (STUD_NO, STUD_NAME), etc.

- A super key is a group of single or multiple keys that uniquely identifies rows in a table. It supports NULL values in rows.
- A super key can contain extra attributes that aren’t necessary for uniqueness.
- For example, if the "STUD_NO" column can uniquely identify a student, adding "SNAME" to it will still form a valid super key, though it's unnecessary.

Example: Consider the STUDENT table

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

A super key could be a combination of STUD_NO and PHONE, as this combination uniquely identifies a student.



Relation between Primary Key, Candidate Key, and Super Key

2. Candidate Key

The minimal set of attributes that can uniquely identify a tuple is known as a candidate key. For Example, STUD_NO in STUDENT relation.

- A candidate key is a minimal super key, meaning it can uniquely identify a record but contains no extra attributes.
- It is a super key with no repeated data is called a candidate key.
- The minimal set of attributes that can uniquely identify a record.

- A candidate key must contain unique values, ensuring that no two rows have the same value in the candidate key's columns.
- Every table must have at least a single candidate key.
- A table can have multiple candidate keys but only one primary key.

Example: For the STUDENT table below, STUD_NO can be a candidate key, as it uniquely identifies each record.

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

Table: STUDENT_COURSE

STUD_NO	TEACHER_NO	COURSE_NO
1	001	C001
2	056	C005

A composite candidate key example: {STUD_NO, COURSE_NO} can be a candidate key for a STUDENT_COURSE table.

3. Primary Key

There can be more than one candidate key in relation out of which one can be chosen as the primary key. For Example, STUD_NO, as well as STUD_PHONE, are candidate keys for relation STUDENT but STUD_NO can be chosen as the primary key (only one out of many candidate keys).

- A primary key is a unique key, meaning it can uniquely identify each record (tuple) in a table.
- It must have unique values and cannot contain any duplicate values.
- A primary key cannot be NULL, as it needs to provide a valid, unique identifier for every record.
- A primary key does not have to consist of a single column. In some cases, a composite primary key (made of multiple columns) can be used to uniquely identify records in a table.
- Databases typically store rows ordered in memory according to primary key for fast access of records using primary key.

Example:

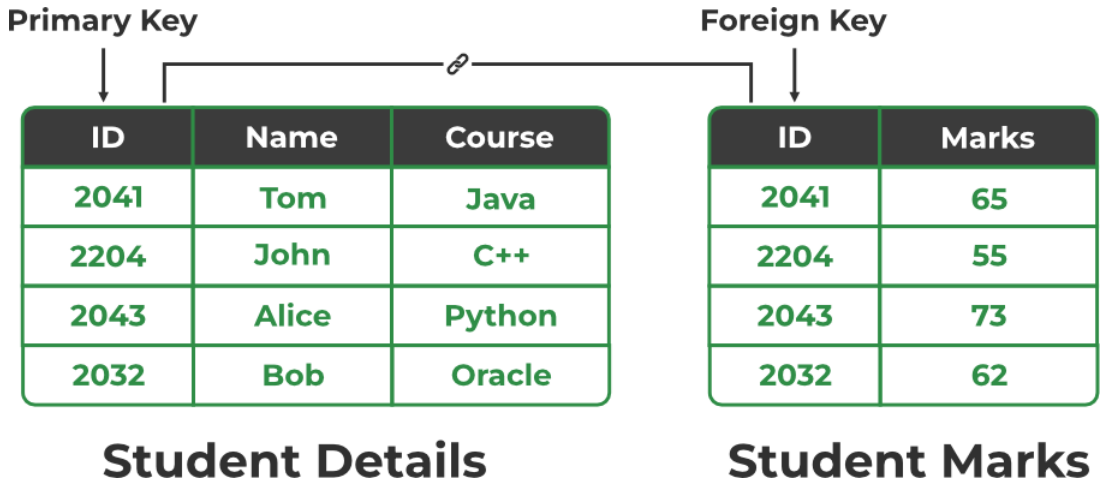
STUDENT table -> Student(STUD_NO, SNAME, ADDRESS, PHONE) , STUD_NO is a primary key

Table: STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

5. Foreign Key

A foreign key is an attribute in one table that refers to the primary key in another table. The table that contains the foreign key is called the referencing table and the table that is referenced is called the referenced table.



Relation between Primary Key and Foreign Key

- A foreign key in one table points to the primary key in another table, establishing a relationship between them.
- It helps connect two or more tables, enabling you to create relationships between them. This is important for maintaining data integrity and preventing data redundancy.
- They act as a cross-reference between the tables.

Example: Consider the STUDENT_COURSE table

STUD_NO	TEACHER_NO	COURSE_NO
1	005	C001
2	056	C005

Explanation:

- Here, STUD_NO in the STUDENT_COURSE table is a foreign key that references the STUD_NO primary key in the STUDENT table.

- Unlike the Primary Key of any given relation, Foreign Key can be NULL as well as may contain duplicate tuples i.e. it need not follow uniqueness constraint. For Example, STUD_NO in the STUDENT_COURSE relation is not unique.
- It has been repeated for the first and third tuples. However, the STUD_NO in STUDENT relation is a primary key and it needs to be always unique and it cannot be null.

Q3 Constraints: NOT NULL, UNIQUE, CHECK

SQL Constraints

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into the following two types,

1. **Column level constraints:** Limits only column data.
2. **Table level constraints:** Limits whole table data.

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, use the NOT NULL constraint.

- It restricts a column from having a NULL value.
- We use ALTER statement and MODIFY statement to specify this constraint.

One important point to note about this constraint is that it cannot be defined at table level.

Example using NOT NULL constraint:

```
CREATE TABLE Student
(       s_id int NOT NULL,
```

```
    name varchar(60),
    age  int
);
```

The above query will declare that the **s_id** field of **Student** table will not take NULL value.

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

```
ALTER TABLE Student
MODIFY s_id int NOT NULL;
```

UNIQUE Constraint

It ensures that a column will only have unique values. A UNIQUE constraint field cannot have any duplicate data.

- It prevents two records from having identical values in a column
- We use ALTER statement and MODIFY statement to specify this constraint.

Example of UNIQUE Constraint:

Here we have a simple CREATE query to create a table, which will have a column **s_id** with unique values.

```
CREATE TABLE Student
(    s_id int  NOT NULL,
     name varchar(60),
     age int  NOT NULL UNIQUE
);
```

The above query will declare that the **s_id** field of **Student** table will only have unique values and wont take NULL value.

If you wish to alter the table after it has been created, then we can use the ALTER command for it:

```
ALTER TABLE Student
MODIFY age INT NOT NULL UNIQUE;
```

The above query specifies that **s_id** field of **Student** table will only have unique value.

CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

Using CHECK constraint at Table Level

```
CREATE table Student(
   s_id int NOT NULL CHECK(s_id > 0),
   Name varchar(60) NOT NULL,
   Age int
);
```

The above query will restrict the **s_id** value to be greater than zero.

Using CHECK constraint at Column Level

```
ALTER table Student ADD CHECK(s_id > 0);
```

Q4 Relational Algebra operations: o Selection, Projection, Joins, Union, Intersection, Set Difference

Introduction of Relational Algebra in DBMS

Relational Algebra is a formal language used to query and manipulate relational databases, consisting of a set of operations like **selection**, **projection**, **union**, and **join**. It provides a mathematical framework for querying databases, ensuring efficient data retrieval and manipulation. Relational algebra serves as the mathematical foundation for query SQL

Relational algebra simplifies the process of querying databases and makes it easier to understand and optimize query execution for better performance. It is essential for learning SQL because SQL queries are based on relational algebra operations, enabling users to retrieve data effectively.

Key Concepts in Relational Algebra

Before explaining relational algebra operations, let's define some fundamental concepts:

1. Relations: In relational algebra, a relation is a table that consists of rows and columns, representing data in a structured format. Each relation has a unique name and is made up of tuples.

2. Tuples: A tuple is a single row in a relation, which contains a set of values for each attribute. It represents a single data entry or record in a relational table.

3. Attributes: Attributes are the columns in a relation, each representing a specific characteristic or property of the data. For example, in a "Students" relation, attributes could be "Name", "Age", and "Grade".

4. Domains: A domain is the set of possible values that an attribute can have. It defines the type of data that can be stored in each column of a relation, such as integers, strings, or dates.

Basic Operators in Relational Algebra

Relational algebra consists of various **basic operators** that help us to fetch and manipulate data from **relational tables** in the database to perform certain operations on relational data. **Basic operators** are fundamental operations that include selection (σ), projection (π), union (U), set difference (−), Cartesian product (×), and rename (ρ).

Operators in Relational Algebra

1. Selection(σ)

The Selection Operation is basically used to filter out rows from a given table based on certain given condition. It basically allows us to retrieve only those rows that match the condition as per condition passed during SQL Query.

Example: If we have a relation **R** with attributes **A**, **B**, and **C**, and we want to select tuples where **C > 3**, we write:

A	B	C
1	2	4
2	2	3
3	2	3
4	3	4

σ(c>3)(R) *will select the tuples which have c more than 3.*

Output:

A	B	C
1	2	4
4	3	4

Explanation: The selection operation only filters rows but does not display or change their order. The **projection** operator is used for displaying specific columns.

2. Projection(π)

While Selection operation works on **rows**, similarly projection operation of relational algebra works on columns. It basically allows us to pick specific columns from a given relational table based on the given condition and ignoring all the other remaining columns.

Example: Suppose we want columns B and C from Relation R.

π(B,C)(R) will show following columns.

Output:

B	C
2	4
2	3
3	4

Explanation: By Default, projection operation removes duplicate values.

3. Union(U)

The **Union** Operator is basically used to combine the results of two queries into a single result. The only condition is that both queries must return same number of columns with same data types. Union operation in relational algebra is the same as union operation in set theory.

Example: Consider the following table of Students having different optional subjects in their course.

FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN

Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21
Rohan	25

If **FRENCH** and **GERMAN** relations represent student names in two subjects, we can combine their student names as follows:

π(Student_Name)(FRENCH) U π(Student_Name)(GERMAN)

Output:

Student_Name
Ram
Mohan

Vivek
Geeta
Shyam
Rohan

Explanation: The only constraint in the union of two relations is that both relations must have the same set of Attributes.

4. Set Difference(-)

Set difference basically provides the rows that are present in one table, but not in another tables. Set Difference in relational algebra is the same set difference operation as in set theory.

Example: To find students enrolled only in FRENCH but not in GERMAN, we write:

π(Student_Name)(FRENCH) - π(Student_Name)(GERMAN)

Student_Name
Ram
Mohan

Explanation: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

. Join Operators

Join operations in relational algebra combine data from two or more relations based on a related attribute, allowing for more complex queries and data retrieval. Different types of joins include:

Inner Join

An inner join combines rows from two relations based on a matching condition and only returns rows where there is a match in both relations. If a record in one relation doesn't have a corresponding match in the other, it is excluded from the result. This is the most common type of join.

- **Conditional Join**: A conditional join is an inner join where the matching condition can involve any comparison operator like equals (=), greater than (>), etc. **Example:** Joining Employees and Departments on DepartmentID where Salary > 50000 will return employees in departments with a salary greater than 50,000
- **Equi Join**: An equi join is a type of conditional join where the condition is specifically equality (=) between columns from both relations. **Example:** Joining Customers and Orders on CustomerID where both relations have this column, returning only matching records.
- **Natural Join**: A natural join automatically combines relations based on columns with the same name and type, removing duplicate columns in the result. It’s a more efficient

way of joining. **Example:** Joining Students and Enrollments where StudentID is common in both, and the result contains only unique columns.

Outer Join

An outer join returns all rows from one relation, and the matching rows from the other relation. If there is no match, the result will still include all rows from the outer relation with NULL values in the columns from the unmatched relation.

- **Left Outer Join**: A left outer join returns all rows from the left relation and the matching rows from the right relation. If there is no match, the result will include NULL values for the right relation attributes. **Example:** Joining Employees with Departments using a left outer join ensures all employees are listed, even those who aren't assigned to any department, with NULL values for the department columns.
- **Right Outer Join**: A right outer join returns all rows from the right relation and the matching rows from the left relation. If no match exists, the left relation's columns will contain NULL values. **Example:** Joining Departments with Employees using a right outer join includes all departments, even those with no employees assigned, filling unmatched employee columns with NULL.
- **Full Outer Join**: A full outer join returns all rows when there is a match in either the left or right relation. If a row from one relation does not have a match in the other, NULL values are included for the missing side. **Example:** Joining Customers and Orders using a full outer join will return all customers and orders, even if there's no corresponding order for a customer or no customer for an order.

2. Set Intersection(∩)

Set Intersection basically allows to fetches only those rows of data that are common between two sets of relational tables. Set Intersection in relational algebra is the same set intersection operation in set theory.

Example: Consider the following table of Students having different optional subjects in their course.

Relation FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

Relation GERMAN

Student_Name	Roll_Number
Vivek	13

Geeta	17
Shyam	21
Rohan	25

From the above table of FRENCH and GERMAN, the Set Intersection is used as follows:

π(Student_Name)(FRENCH ∩ π(Student_Name)(GERMAN)

Output:

Student_Name
Vivek
Geeta

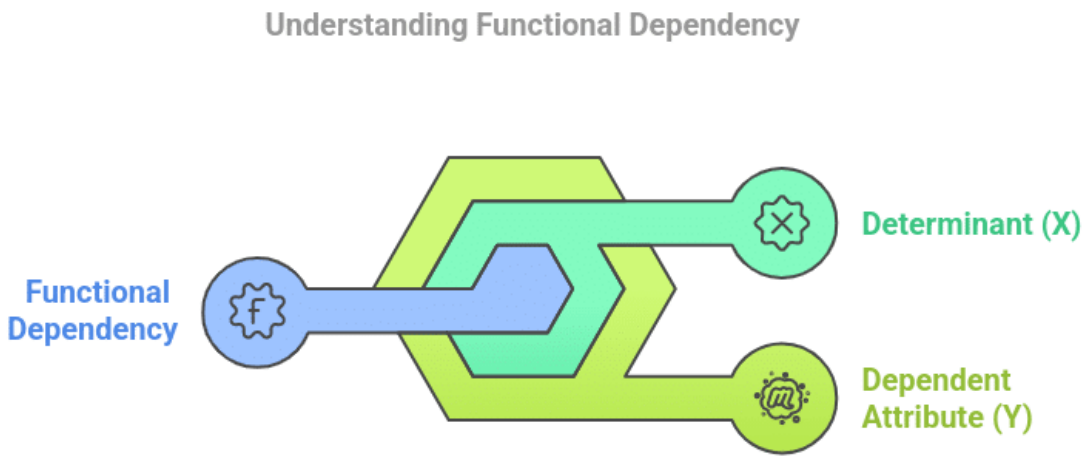
Explanation: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

UNIT 5: RELATIONAL DATABASE DESIGN

Q1 Functional Dependency: Definition, types, examples

Functional Dependency in DBMS

Functional Dependency is essential for database normalization, reducing redundancy, and maintaining data consistency. This guide explains its basics and practical applications.



Database Management System (DBMS) is a system for storing, managing and organizing data efficiently. Reducing redundancy and preserving data consistency is essential for relational databases to function at their best. A key idea that helps achieve these objectives is functional dependency (FD). This idea is the foundation of database normalization and practical design.

In a relational database, a functional dependency is a link between attributes that establishes how one property uniquely identifies another. An employee's ID, for instance, can be used to uniquely determine their name and other information in an employee database.

Building reliable and effective databases requires a thorough understanding of and adherence to functional dependencies.

What is functional dependency?

A constraint between two sets of attributes in a relation is known as functional dependency. It is denoted as X→Y, where:

- X is the determinant, and
- Y is the dependent attribute.

According to this notation, two rows in a table must have the same value for X if they also have the same value for Y. In simpler words, the value of X determines the value of Y.

Example: Consider a table with table name Students:

StudentID	Name	Department

101	Alice	Computer
102	Bob	Electrical
103	Alice	Computer

Here, StudentID uniquely determines the Name and Department. This can be represented as:

- StudentID → Name
- StudentID → Department

Functional dependency is commonly used in database architecture and normalisation and is vital for ensuring data consistency.

Also Read: ACID Properties in DBMS

Types of Functional Dependency

1. Full Functional Dependency:

When every attribute in the determinant is required to determine the dependent attribute, the functional dependency is said to be complete. The reliance would be broken if the determinant's attributes were removed.

Example: Consider a table with the table name Course:

CourseID	Instructor	Duration
C101	John	3 months
C102	Sarah	6 months

Here, CourseID → Instructor and CourseID → Duration are full functional dependencies, as CourseID alone determines both Instructor and Duration.

2. Partial Functional Dependency:

If an attribute is determined by only a part of a composite key, the dependency is said to be partial.

Example: Consider a table Subject:

CourseID	SubjectID	SubjectName
C101	S1	Mathematics
C102	S2	Physics

{CourseID, SubjectID} → SubjectName is a composite key dependency, but SubjectID → SubjectName is a partial dependency as SubjectID alone determines SubjectName.

Normalisation is necessary to eliminate data anomalies and redundancies that are frequently caused by partial dependencies.

3. Transitive Dependency:

When an attribute indirectly depends on the determinant through another attribute, this is known as a transitive dependency.

Example:

EmployeeID	DepartmentID	DepartmentName
E01	D1	HR
E02	D2	IT

Here, EmployeeID → DepartmentID and DepartmentID → DepartmentName, implying EmployeeID → DepartmentName is a transitive dependency.

Higher stages of normalisation deal with transitive dependencies, which frequently make data linkages more difficult.

4. Multivalued Dependency

When a particular attribute, independent of other attributes, determines a set of values for another, this is known as a multivalued dependency.

Example: Consider a table Projects:

EmployeeID	Project
E01	Project A
E01	Project B

EmployeeID → Project is a multivalued dependency.

In situations where there are many-to-many links between attributes, multivalued dependencies usually arise. The Fourth Normal Form (4NF) addresses these dependencies.

5. Trivial and Non-Trivial Dependency

- **Trivial Dependency:** Takes place when the dependent attribute's subset of the determinant (e.g., A → A).
- **Non-Trivial Dependency:** When the dependent attribute is not a subset of the determinant, this is known as non-trivial dependency (e.g., A → B).

Designing databases that are devoid of redundancies and inconsistencies requires an

understanding of these differences.

Q2 Armstrong's Axioms

What are Armstrong's Axioms in Functional Dependency in DBMS?

Armstrong axioms are a complete set of inference rules or axioms, introduced and developed by William W. Armstrong in 1974. The inference rules are sound which is used to test logical inferences of functional dependencies. The axiom which also refers to as sound is used to infer all the functional dependencies on a relational database. The Axioms are a set of rules, that when applied to a specific set, generates a closure of functional dependencies.

Rules of Armstrong's Axioms in Functional Dependency

Armstrong's Axioms has two different set of rules,

1. Axioms or primary rules
    a. Axiom of Reflexivity
    b. Axiom of Augmentation
    c. Axiom of Transitivity
2. Additional rules or Secondary rules
    a. Union
    b. Composition
    c. Decomposition
    d. Pseudo Transitivity

1. Axioms or Primary Rules

Let suppose **T (k)** with the set of attributes **k** be a relation scheme. Subsequently, we will represent subsets of **k** as **A**, **B**, **C**. The standard notation in database theory for the set of attributes is **AB** rather than **AUB**.

a. **Axiom of Reflexivity**:
If a set of attributes is P and its subset is **Q**, then **P** holds **Q**. If **Q ⊆ P**, then **P → Q**. This property is called as Trivial functional dependency. Where **P** holds **Q** (**P → Q**) denote **P** functionally decides **Q**.

b. **Axiom of Augmentation**:
If **P** holds **Q** (**P → Q**) and **R** is a set of attributes, then **PR** holds **QR** (**PR → QR**). It means that a change in attributes in dependencies does not create a change in basic dependencies. If **P → Q**, then **PR → QR** for any **R**.

c. **Axiom of Transitivity**:
If **P** holds **Q** (**P → Q**) and **Q** holds **R** (**Q → R**), then **P** hold **R** (**P → R**). Where **P** holds **R** (**P → R**) denote **P** functionally decides **R**, same with **P** holds **Q** and **Q** holds **R**.

2. Additional Rules or Secondary Rules

These rules can be derived from the above axioms.

a. **Union**:
If **P** holds **Q** (**P → Q**) and **P** holds **R** (**P → R**), then **P → QR**. If **X → Y** and **X → Z**, then **X → YZ**.

b. **Composition**:
If **P** holds **Q** (**P → Q**) and **A** holds **B** (**A → B**), then **PA → QB**.
proof,

1. P → Q (Given)
2. A → B (Given)
3. PA → QA (Augmentation of 1 and A)
4. PA → Q (Decomposition of 3)
5. PA → PB (Augmentation of 2 and P)
6. PA → B (Decomposition of 5)
7. PA → QB (Union 4 and 6)

c. **Decomposition**:
This rule is contrary of union rule. If **P → QR**, then **P** holds **Q** (**P → Q**) and **P** holds **R** (**P → R**).
If **X → YZ**, then **X → Y** and **X → Z**.
proof,

1. P → QR (Given)
2. QR → Q (Reflexivity)
3. P → Q (Transitivity of 1 and 2)

d. **Pseudo Transitivity**:
If **P → RQ** and **Q → S**, then **P → RS**.
proof,

1. P → RQ (Given)
2. Q → S (Given)
3. RQ → RS (Augmentation of 2 and R)
4. P → RS (Transitivity of 1 and 3)

Trivial Functional Dependency

Trivial	If **P** holds **Q** (**P → Q**), where **P** is a subset of **Q**, then it is called a Trivial Functional Dependency. Trivial always holds Functional Dependency.
Non-Trivial	If **P** holds **Q** (**P → Q**), where **Q** is not a subset of **P**, then it is called as a Non-Trivial Functional Dependency.

Completely Non-Trivial	If **P** holds **Q** (**P → Q**), where **P** intersect **Y = Φ**, it is called as a Completely Non-Trivial Functional Dependency.

Q3 Attribute and FD closure

Functional Dependencies and Attribute Closure

Introduction

Functional dependency can be defined as the method that describes the relationship between the attributes in a given relation. It means we represent how the different attributes in a dataset table are related to each other with the help of functional dependencies. Suppose there is functional dependency A→B; it means 'A determines B' or 'B is determined by A .'Here 'A' is a determinant attribute, and 'B' is a determined attribute. We can also say that 'B' is dependent on 'A.'

Now let's understand with the help of an example, suppose there is a functional dependency given STUDENT_ID→ STUDENT_NAME, as mentioned in the table below:

STUDENT_ID	STUDENT_NAME
1	David
2	David

Here in the above table, we see two students with the same name. How will we differentiate whether they are the same students or different ones? To uniquely identify this, we will refer to STUDENT_ID. Now we can clearly see that both have different IDs, which means both are different students, and each tuple dataset (all the other attributes) will be different. Here STUDENT_NAME is a determined attribute, and STUDENT_ID is a determinant attribute that solves confusions related to determinant attribute by uniquely identifying the tuple.

Validating a Functional Dependency

Now, let's see how functional dependencies can help to validate a data set with a few examples. It will help us to understand the concept more clearly.

Table 1

STUDENT_ID	STUDENT_NAME
1	Bob
2	David

In the above table, the functional dependency from STUDENT_ID → STUDENT_NAME is valid since both students have different names and ids.

Table 2

STUDENT_ID	STUDENT_NAME
1	Bob

1	Bob

In the above table, functional dependency STUDENT_ID → STUDENT_NAME is valid since both students are the same, and there is redundant data in the table.

Table 3

STUDENT_ID	STUDENT_NAME
1	Bob
2	Bob

In the above table, functional dependency STUDENT_ID → STUDENT_NAME is valid since both students are different, and STUDENT_ID helps identify it uniquely.

Table 4

STUDENT_ID	STUDENT_NAME
1	Bob
1	David

In the above table, functional dependency STUDENT_ID,→ STUDENT_NAME is not valid since both students are different and cannot have the same STUDENT_ID.

Types of Functional Dependencies

They are four types of functional dependencies -

1. Trivial Functional Dependency
2. Non-trivial Functional Dependency
3. Multivalued Functional Dependency
4. Transitive Functional Dependency

Now, let's understand what these are:

Trivial Functional Dependency

Let's assume there is functional dependency A→B. If it is a trivial functional dependency, it means that 'B is a subset of A.' It confirms that trivial dependencies are always valid because the attribute to be determined is the subset of the left-hand side attribute. Here reflexive relation holds.

For example, STUDENT_ID→STUDENT_ID is a trivial functional dependency; it will always be valid, validity and invalidity of a functional dependency have already been discussed above.

There is another method to determine trivial functional dependency: if we take the intersection of left and right attributes, it will never be empty('ϕ').

For example (STUDENT_ID, STUDENT_NAME)—>STUDENT_NAME, LHS and RHS intersection is not null.

Non-trivial Functional Dependency

A non-trivial functional dependency A→B means that 'B is not a subset of A' and A intersection B will be 'null' or 'ϕ.'

For example, a functional dependency STUDENT_ID→STUDENT_NAME is a non-trivial dependency since STUDENT_NAME is not a subset of STUDENT_ID and the intersection of both of these will be 'ϕ';

In non-trivial dependency cases of validity and invalidity arise, trivial ones are always valid.

Multivalued Functional Dependency

When two attributes in a table are independent of each other but both rely on a third attribute, this is referred to as multivalued dependency.

A multivalued dependency consists of at least two attributes that are dependent on a third attribute, which is why at least three attributes are always required.

Let's see an example of the 'Student' table:

STU_ID	COURSE	PASSING_YEAR
1	Science	2020
2	Science	2021

Here, STU_ID can determine both COURSE and PASSING_YEAR. STU_ID→{ COURSE, PASSING_YEAR }, but there is no functional dependency between COURSE and PASSING_YEAR. Hence we can say that COURSE and PASSING_YEAR both are independent of each other which makes them a multivalued dependent on STU_ID.

Transitive Functional Dependency

A functional dependency that is indirectly formed by two functional dependencies is called transitive functional dependency.

For example, if A→B, B→C holds true, then according to the axiom of transitivity, A→C will also hold true.

Let's see an example of a 'Student' table:

STU_ID	CLASS	LECTURE_HALL
1	7	L202
2	6	B101

Here, with the STU_ID we can determine CLASS, and with CLASS, we can determine the LECTURE_HALL number for that particular class. It means with the help of STU_ID, we can determine LECTURE HALL. Therefore STU_ID→LECTURE_HALL holds true.

Advantages of Functional Dependencies

Functional dependencies in a relational database schema offer several benefits:

- **Data Integrity:** Functional dependencies help enforce data integrity by defining constraints that ensure each attribute's values are determined by another attribute or combination of attributes. This prevents inconsistent or invalid data from being stored in the database.

- **Normalization:** Functional dependencies aid in the normalization process, which involves organizing data into tables to reduce redundancy and dependency. By identifying and eliminating redundant data through normalization, databases become more efficient, easier to maintain, and less prone to anomalies.
- **Query Optimization:** Understanding functional dependencies allows database systems to optimize query execution by recognizing redundant attributes that can be eliminated from queries. This optimization reduces the amount of data retrieval and processing required, leading to faster query execution times.
- **Data Consistency:** By ensuring that attributes depend on specific combinations of other attributes, functional dependencies promote data consistency. This consistency helps maintain the accuracy and reliability of the database, ensuring that changes to one attribute do not inadvertently affect others.

Disadvantages of Functional Dependencies

- **Complexity:** Managing and enforcing functional dependencies in large databases can be complex and time-consuming. As the number of attributes and dependencies increases, it becomes challenging to ensure that all dependencies are properly maintained and enforced.
- **Performance Overhead:** Enforcing functional dependencies may introduce performance overhead, particularly during data modification operations such as insertion, update, and deletion. Maintaining referential integrity constraints and enforcing dependency rules can impact the performance of database transactions.
- **Dependency Maintenance:** Functional dependencies need to be updated and maintained as the database schema evolves over time. This requires careful analysis and potentially modifying existing dependencies to accommodate changes in business requirements or data structures.
- **Potential for Over-Normalization:** Over-reliance on functional dependencies for normalization can lead to over-normalization, where the database schema becomes excessively decomposed. This can result in increased join operations and decreased query performance, especially for complex queries involving multiple tables.

Functional Dependency Set

The functional Dependency set(or FD set) of a relation is the set of all functional dependencies present in the given relation.

For example, in the table given below FD set will be

STU_ID	STU_NAME	STU_AGE	STU_STATE	STU_CONTRY	STU_NO

A valid FD set for the above relation can be:

{

1. **STU_ID→STU_NAME,**
2. **STU_ID→STU_AGE,**
3. **STU_ID→STU_NO,**
4. **STU_STATE→STU_COUNTRY**

}

Here, STU_ID is a 'determinant attribute' that can determine and validate STU_NAME, STU_AGE, STU_NO. Similarly, STU_STATE→STU_COUNTRY is a functional dependency where STU_STATE can determine and validate STU_COUNTRY. So set of all these dependencies will form an FD set.

Properties of Functional Dependencies

Some essential properties of functional dependencies are:

1. **Reflexive**: if B is a subset of A, then A→B.

If $X \supseteq Y$ then $X \rightarrow Y$ // Reflexive property

For example {STU_ID, NAME} →NAME is valid reflexive relation.

2**. Augmentation**: if A→B then AC→BC for any C.

For example {STU_ID, NAME} →{ DEPT_BUILDING} is valid then {STU_ID, NAME,DEPT_NAME} →{ DEPT_BUILDING,DEPT_NAME} is also valid.

3**. Transitive**: if A→B and B→C, then A→C.

For example, if STU_ID→CLASS, CLASS→LECTURE_HALL holds true then according to the axiom of transitivity, STU_ID→LECTURE_HALL will also hold true.

4. **Union**: if A→B and A→C, then A→BC.

For example, STU_ID → STU_NAME, STU_ID→COURSE then STU_ID→ {STU_NAME, COURSE} holds true.

5. **Decomposition**: if A→BC, then A→B, and A→C .

For example STU_ID→ {STU_NAME, COURSE} then STU_ID → STU_NAME, STU_ID→COURSE holds true.

After understanding the functional dependencies and their types, let's know what an attribute closure is:

Attribute Closure

Attribute Closure of an attribute set is defined as a set of all attributes that can be functionally determined from it.

The closure of an attribute is represented as $^{+}$

Finding Closure of an attribute set

You can follow the steps to find the Closure of an attribute set:

1. Determine A^+, the Closure of A under functional dependency set F.
2. A^+: = will contain A itself; **For example**, if we need to find the closure of an attribute X, the closure will incorporate the X itself and the other attributes that the X attribute can determine.
3. Repeat the process as
4. old A^+: = A Closure;
5. for each FB X → Y in the FD set, do
6. if X Closure is a subset of X, then A Closure:= A Closure U Y;
7. Repeat until (A^+= old A^+);

For example,

Given a relation R(A,B,C,D) and FD { A→B, B→C, C→D}, then determine the A^+:

1. A^+=A, since A can determine A itself.
2. A^+=AB, A can also determine B, it is because in the FD set A→B dependency is given.
3. A^+=ABC, A can also determine C with the help of B, since
   A → B, B → C, thus, A → C // Transitive property
4. A^+=ABCD, A can also determine D with the help of the C attribute, since C is already determined now in the FD set functional dependency C→D holds, D can be determined with C.
   Therefore,
   A^+(A closure) =ABCD. A can determine all attributes (ABCD).

Now let's see some questions for a clear understanding of the concept.

Question: In a schema with attributes A, B, C, D, E following set of functional dependencies are given

{ A→B, A→C, CD→E, B→D, E→A},

Which of the following dependencies is **not** implied by the above set? (**GATE CS 2005**).

1. CD→AC
2. BD→CD
3. BC→CD
4. AC→BC

Solution:

Checking option A, CD→AC,

1. CD can determine C and D(can determine themselves) and E(as CD→E, given in FD set) . Therefore, $(CD)^+$=CDE
2. Now E can determine A (as E→A), $(CD)^+$=CDEA
3. With the help of A, we can determine B, $(CD)^+$=CDEAB

So,$(CD)^+$=ABCDE

We can clearly see that with CD, AC can be determined. Therefore, CD→AC holds true.

Checking for Option B, BD→CD,

On taking BD closure $(BD)^+$=BD as they can determine themselves only, no other dependencies in the FD set.

We can see that BD→CD does not hold; **hence B is the correct option.**

Other options can be checked similarly by taking closure.

Another example:

Question: The following functional dependencies are given:

{AB→CD, AF→D, DE→F, C→G, F→E, G→A}

Which one of the following options is false? (**GATE 2006)**

1. CF+ = {ACDEFG}
2. BG+ = {ABCDG}
3. AF+ = {ACDEFG}
4. AB+ = {ABCDFG}

Solution:

If we take the attribute closure of option A, we will get, $(CF)^+$= {ACDEFG}

If we take the attribute closure of option B, we will get,$(BD)^+$={ABCDG}

This can be done with the steps discussed above in the article.

But option C and D have attribute closure: $(AF)^+$=(AFDE) and $(AB)^+$=(ABCDG).

Therefore, options C and D are false.

Finding Candidate Keys and Super Keys using the Attribute Closure

A candidate key is the minimal set of attributes that can uniquely identify a tuple. For example, STU_ID in the 'Student' relation can be a candidate key. The candidate key should be unique and not null.

A superkey is also a set of attributes that can uniquely identify a tuple in a given relation. The concept of the candidate key and the super key is closely related. Super key reduced to the minimum number of attributes makes the candidate key, which means the super key is the superset of a candidate key. In a given relation, 'Student' STU_ID and STU_NAME can be super key.

The attribute closure method can also be used to find all the **candidate keys and super keys** in the given relation.

If the attribute closure of an attribute set contains all the attributes of the given relation, then the attribute set will be the **superkey** of the relation.

Similarly, if no subset of this attribute set can functionally determine all relation attributes, the present set will be a candidate key.

For example given a relation R(A,B,C,D) and FD { A→B, B→C, C→D,D→A}, then

A^+=BCDA,

B^+=ACDB

C^+=ACDB

D^+=ACDB

Since all A, B, C, D can functionally determine all the attributes, so candidate key set will be {A, B, C, D}. With the help of this, we can also determine prime and non-prime attributes.

Prime attributes are the set of all attributes present in the candidate key, and non-prime attributes are the remaining attributes of the relation.

So here, Prime attributes are {A, B, C, D} and there is no non-prime attribute(non-prime attribute set will be empty {ϕ}).

Superkeys can be formed by combining other attributes with the candidate key. Suppose AB can uniquely identify all the other attributes in the given, and A alone can do this, so AB be super key and A will be candidate key

GATE Question: Consider a relation scheme **R = (A, B, C, D, E, H)** on which the following functional dependencies hold: **{A–>B, BC–>D, E–>C, D–>A}**. What are the candidate keys of R?

1. AE, BE
2. AE, BE, DE
3. AEH, BEH, BCH
4. AEH, BEH, DEH

Solution: A set of attributes S is a candidate key of relation R if the closure of S is all attributes of R and there is no subset of S whose closure is all attributes of R.

If we look closely, attributes E and H are not determined by any of the dependencies given in the FD set. Therefore they need to be present on the left-hand side. Only option D satisfies the given condition. Therefore, we can check for attribute closure.

$(AEH)^+$={ABCDEH}

$(BEH)^+$={ABCDEH}

$(DEH)^+$={ABCDEH}

Option D is correct.

Advantages of Attribute Closure

- **Determining Functional Dependencies:** Attribute closure is used to determine the functional dependencies present in a relation. By computing the closure of a set of attributes, we can identify all attributes that are functionally dependent on the given set, aiding in database normalization and integrity.
- **Database Design:** Attribute closure helps in designing well-structured and normalized database schemas. By understanding the dependencies between attributes, database designers can optimize the schema design to minimize redundancy and ensure data integrity.
- **Query Optimization:** Attribute closure provides valuable information for query optimization. By knowing which attributes are functionally dependent on others, query optimizers can generate efficient execution plans, leading to faster query processing times.
- **Data Integrity:** Attribute closure helps maintain data integrity by ensuring that data

dependencies are properly enforced. By identifying and enforcing functional dependencies, we can prevent data anomalies such as insertion, update, and deletion anomalies.

Disadvantages of Attribute Closure

- **Complexity:** Computing attribute closure for large sets of attributes or complex dependencies can be computationally expensive and time-consuming. As the number of attributes and dependencies increases, the complexity of computing closures grows, potentially impacting performance.
- **Dependency Maintenance:** Attribute closure needs to be updated and maintained as the database schema evolves over time. This requires careful analysis and potentially modifying existing closures to accommodate changes in business requirements or data structures.
- **Dependency Inference:** In some cases, inferring all functional dependencies from a given set of attributes may result in an excessive number of dependencies, including trivial or redundant ones. Managing and enforcing such dependencies can lead to unnecessary complexity and overhead.
- **Over-Normalization:** Over-reliance on attribute closure for normalization can lead to over-normalization, where the database schema becomes excessively decomposed. This can result in increased join operations and decreased query performance, especially for complex queries involving multiple tables.

Prime and Non-Prime Attributes

- **Prime Attributes:** Prime attributes are attributes that are part of any candidate key for a relation. They play a crucial role in determining the minimal set of attributes required to uniquely identify each tuple in a relation. Prime attributes are essential for database normalization and maintaining data integrity.
- **Non-Prime Attributes:** Non-prime attributes are attributes that are not part of any candidate key for a relation. They are functionally dependent on prime attributes and can be derived from them through attribute closure. Non-prime attributes contribute to the overall information content of the relation but are not involved in identifying tuples uniquely.

Q4 Candidate key identification

Candidate Key in DBMS

- Definition of candidate key: Super key with no redundant attributes known as candidate key i.e should not contain any column that contains duplicate data.
- Hence candidate key also called as minimal super key.

A candidate satisfies the following properties

- A candidate key column must be unique i.e all the columns that are involved in the

candidate key also must be unique

- A candidate key may have more than one attribute
- A candidate key column will not contain any null value
- Candidate key always selects a minimum column combination that helps to uniquely identify the record.

Example for Candidate Key

Consider the following table student

ID	NAME	PHONE
66	Elena	9789578575
67	Haley	8758273875
68	Alex	7847326558
69	Elena	7923857563

First, identify all the super keys present in the table, then eliminate the super keys that contain a column with duplicate data. Then the remaining superkeys that are left or nothing but candidate keys

1. {Id}: ID column will contain all unique values hence ID column is a candidate key
2. {phone}: As no two students have the same phone number, it is not a redundant data column and hence phone column is a candidate key
3. {Id, phone}: As both ID and phone are unique for all students this combination is a valid candidate key
4. {Id, Name}: This combination is not a candidate key because the name column may have duplicate values
5. {Id, phone, Name}: This combination is not a candidate key because the name column may have duplicate values
6. {Name, Phone}: This combination is not a candidate key because the name column may have duplicate values

Candidate keys available in the table student

- {Id}
- {phone}
- {Id, phone}

Candidate key versus super key

- First of all, you need to understand that all candidate keys are super keys this is because candidate keys are selected from super keys
- Look for those keys from which we can remove any columns that contain duplicate data. In

the above example, we have not chosen {Id, name} as candidate key because {Id} alone can identify a unique row in the table, and {Name} column is redundant.

Q5 Normalization (1NF to BCNF mandatory, 4NF & 5NF optional)

Types of DBMS Normal forms

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

Let's cover all the Database Normal forms one by one with some basic examples to help you understand the DBMS normal forms.
1. First Normal Form (1NF)

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single(**atomic**) valued attributes/columns.
2. Values stored in a column should be of the same domain.
3. All the columns in a table should have unique names.
4. And the order in which data is stored should not matter.

2. Second Normal Form (2NF)

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have **Partial Dependency**.

Let's take an example to understand Partial dependency and the Second Normal Form.

What is Partial Dependency?

When a table has a primary key that is made up of two or more columns, then all the columns(not included in the primary key) in that table should depend on the entire primary key and not on a part of it. If any column(which is not in the primary key) depends on a part of the primary key then we say we have Partial dependency in the table.

Confused? Let's take an example.

If we have two tables Students and Subjects, to store student information and information related

to subjects.

Student table:

student_id	**student_name**	**branch**
1	Akon	CSE
2	Bkon	Mechanical

Subject Table:

subject_id	**subject_name**
1	C Language
2	DSA
3	Operating System

And we have another table **Score** to store the marks scored by students in any subject like this,

student_id	**subject_id**	**marks**	**teacher_name**
1	1	70	Miss. C
1	2	82	Mr. D
2	1	65	Mr. Op

Now in the above table, the primary key is **student_id + subject_id**, because both these information are required to select any row of data.

But in the **Score** table, we have a column **teacher_name**, which depends on the subject information or just the **subject_id**, so we *should not keep* that information in the **Score** table.

The column **teacher_name** should be in the **Subjects** table. And then the entire system will be Normalized as per the Second Normal Form.

Updated **Subject** table:

subject_id	**subject_name**	**teacher_name**
1	C Language	Miss. C
2	DSA	Mr. D
3	Operating System	Mr. Op

Updated **Score** table:

student_id	subject_id	marks
1	1	70
1	2	82
2	1	65

3. Third Normal Form (3NF)

A table is said to be in the Third Normal Form when,

1. It satisfies the First Normal Form and the Second Normal form.
2. And, it doesn't have Transitive Dependency.

4. Boyce-Codd Normal Form (BCNF)

- **Boyce and Codd Normal Form** is a higher version of the Third Normal Form.
- This form deals with a certain type of anomaly that is not handled by 3NF.
- A 3NF table that does not have **multiple overlapping candidate keys** is said to be in BCNF.
- For a table to be in BCNF, the following conditions must be satisfied:
    - R must be in the 3rd Normal Form
    - and, for each functional dependency (X → Y), X should be a Super Key.

5. Fourth Normal Form (4NF)

A table is said to be in the Fourth Normal Form when,

1. It is in the Boyce-Codd Normal Form.
2. And, it doesn't have Multi-Valued Dependency.

6. Fifth Normal Form (5NF)

- The fifth normal form is also called the **PJNF** - **Project-Join Normal Form**
- It is the most advanced level of Database Normalization.
- Using Fifth Normal Form you can fix **Join dependency** and reduce data redundancy.
- It also helps in fixing **Update anomalies** in DBMS design.

Q6 Lossless 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.

Q7 Database anomalies

Anomalies in the relational model refer to inconsistencies or errors that can arise when working with relational databases, specifically in the context of data insertion, deletion and modification. Anomalies can compromise data integrity and make database management inefficient.

How Are Anomalies Caused in DBMS?

Anomalies in DBMS are caused by poor management of storing everything in the flat database, lack of normalization, data redundancy and improper use of primary or foreign keys. These issues result in inconsistencies during insert, update or delete operations, leading to data integrity problems. The three primary types of anomalies are:

- **Insertion Anomalies:** These anomalies occur when it is not possible to insert data into a database because the required fields are missing or because the data is incomplete. For example, if a database requires that every record has a primary key, but no value is provided for a particular record, it cannot be inserted into the database.
- **Deletion anomalies:** These anomalies occur when deleting a record from a database and can result in the unintentional loss of data. For example, if a database contains information about customers and orders, deleting a customer record may also delete all the orders associated with that customer.
- **Update anomalies:** These anomalies occur when modifying data in a database and can result in inconsistencies or errors. For example, if a database contains information about employees and their salaries, updating an employee's salary in one record but not in all related records could lead to incorrect calculations and reporting.

STUDENT Table:

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD-COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajasthan	India	18
4	SURESH		Punjab	India	21

STUDENT_COURSE:

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

1. Insertion Anomaly

If a tuple is inserted in referencing relation and referencing attribute value is not present in referenced attribute, it will not allow insertion in referencing relation. In simpler words, an insertion anomaly occurs when adding a new row to a table leads to inconsistencies.
Example: *If we try to insert a record into the STUDENT_COURSE table with STUD_NO = 7, it will not be allowed because there is no corresponding STUD_NO = 7 in the STUDENT table.*

2. Deletion and Updation Anomaly:

If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will not allow deleting the tuple from referenced relation.

Example: If we want to update a record from STUDENT_COURSE with STUD_NO =1, We have to update it in both rows of the table. If we try to delete a record from the STUDENT table with STUD_NO = 1, it will not be allowed because there are corresponding records in the STUDENT_COURSE table referencing STUD_NO = 1. To avoid this, the following can be used in query:

- **ON DELETE/UPDATE SET NULL:** If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and set the value of referencing attribute to NULL.
- **ON DELETE/UPDATE CASCADE:** If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and referencing relation as well.

Removal of Anomalies

Anomalies in DBMS can be removed by applying normalization. Normalization involves organizing data into tables and applying rules to ensure data is stored in a consistent and efficient manner. By reducing data redundancy and ensuring data integrity, normalization helps to eliminate anomalies and improve the overall quality of the database. According to **E. F. Codd**, who is the inventor of the Relational Database, the goals of Normalization include:

- It helps in vacating all the repeated data from the database.
- It helps in removing undesirable deletion, insertion and update anomalies.
- It helps in making a proper and useful relationship between tables.

Key steps include

1. **First Normal Form (1NF):** Ensures each column contains atomic values and removes repeating groups.
2. **Second Normal Form (2NF):** Eliminates partial dependencies by ensuring all non-key attributes are fully dependent on the primary key.
3. **Third Normal Form (3NF):** Removes transitive dependencies by ensuring non-key attributes depend only on the primary key.