Computer Engineering

22319 Database Management System

Unit - I Database System Concept

1.1 Concept of Data, database, DBMS, advantages of DBMS over file processing system, Application of database.

1. Concept of Data:

Data refers to raw facts and figures, or unprocessed information. It can be any sequence of characters, including text and numbers, that is suitable for processing in a computer. Data becomes **information** when it is processed and organized into a format that gives it context, relevance, and purpose.

---

2. Concept of Database:

A **database** is a systematic and organized collection of related data that supports storage, modification, and retrieval. The data can be of various types including text, images, sound, and video. Databases are designed so that they can be easily accessed, managed, and updated.

---

3. Concept of DBMS (Database Management System):

A **Database Management System (DBMS)** is software designed to assist in maintaining and utilizing a database. It provides users with a systematic way to create, retrieve, update, and manage data. The DBMS essentially serves as an interface between the database and the end-users or application programs, ensuring data consistency and security.

---

4. Advantages of DBMS Over File Processing System:

- **Reduced Data Redundancy**: Unlike file systems, in which each application has its own private files, all the data is centralized in a DBMS. This eliminates data redundancy.
- **Data Integrity**: Data integrity is ensured in a DBMS through a set of integrity constraints.
- **Easy Data Access**: Sophisticated DBMS allows users and programmers to access data in a flexible manner without needing to know the physical structure of the database.
- **Data Security**: Centralizing data increases the security of data as compared to file-based systems.
- **Support for Multiple User Views**: DBMS systems offer an environment in which end-users have better options to frequently achieve a personalized view of data.
- **Concurrent Access**: DBMS supports multiple users simultaneously with mechanisms that ensure data consistency.
- **Data Recovery**: Modern DBMS systems come with sophisticated backup and recovery sub-systems that allow for data recovery in case of system failures.
- **Data Relationships**: Data in databases can be inter-related with primary and foreign keys, allowing for intricate data relations.

5. Application of Database:
Databases find applications in:

- **Banking Systems**: For customer information, account activities, loans, and other banking transactions.
- **Airlines**: For reservations and schedule information.
- **Universities**: To maintain student records, course registrations, and grades.
- **E-Commerce**: To store product inventory, customer purchase history, and credit card purchase data.
- **Healthcare**: For patient records, billing, and tracking of medical history.
- **Telecommunication**: To keep call records, monthly bills, balance, and customer related data.

1.2 Three level Architecture for Database management.

The Three-Level Database Architecture, often called the **Three-Schema Architecture**, is a framework that divides the database architecture into three interconnected levels to separate the user applications and the physical database. This approach provides a separation of concerns, making it easier to manage, access, and understand data. The three levels in this architecture are:

1. External Level (User View):

- This is the **highest level** of the database architecture and pertains to the way individual end-users perceive data (thus sometimes called individual or user views).
- At this level, only a portion of the database's entirety might be visible to users. For instance, a clerk in a company might only view the data related to invoicing.
- It provides a user-friendly interface by allowing multiple external views to be defined based on the different requirements of various user groups.

2. Conceptual Level (Logical Level):

- This is the **middle level** that represents the entire database's logical view without concerning the physical complexities or details.
- It includes all the logical structures, entities, relationships, integrity constraints, and more.
- The conceptual schema presents the complete database as a whole rather than the viewpoint of just one user group. It hides the underlying physical storage structures and deals with the data's logical aspect.
- An important feature at this level is the Data Dictionary, which contains metadata (data about data) that defines the conceptual schema.

3. Internal Level (Physical Level):

- This is the **lowest level** and deals with the physical storage of the data.
- It involves data storage on physical media like magnetic disks, the data's actual location, access paths, and more.

- The internal schema defines how the data will be physically stored, including details like data structures, file organizations, indexing, hashing, and more.
- This level deals with complexities such as data compression and encryption.

Mapping:

- Mappings are crucial in three-level architecture.
- **External/Conceptual Mapping**: Between the external and conceptual levels. It determines how the external view(s) relate to the conceptual view.
- **Conceptual/Internal Mapping**: Between the conceptual and internal levels. It determines how the conceptual view relates to the physical data storage.

Advantages of the Three-Level Architecture:

1. **Flexibility**: Since there's a separation between physical and logical views, changes made in one level won't affect the other, thus providing flexibility.
2. **Security**: Different views can be provided to different user groups, so each user group can only access data they're authorized to see.
3. **Abstraction**: Each user interacts with the database at their respective level, and they don't need to be aware of the underlying complexities.
4. **Multiple User Views**: Different users or user groups can have different views of the database, depending on their requirements.

1.3 Data abstraction: Different levels of Data abstraction, Instance and schema, Data independence - Logical and Physical Independence.

1. Data Abstraction:

Data abstraction is a fundamental concept in database management and computer science that refers to the process of simplifying complex reality by modeling classes of objects with their relevant attributes and operations. There are different levels of data abstraction:

- **Physical Level Abstraction**: This is the lowest level of abstraction and deals with how data is physically stored, including aspects like storage media, file organization, and access methods. It's concerned with the hardware and physical characteristics.
- **Logical Level Abstraction**: This level of abstraction is a bit higher and focuses on the structure of data and relationships between data items, independent of the physical details. It deals with defining tables, keys, and relationships.
- **View Level Abstraction (User Level)**: This is the highest level of abstraction and involves how users or applications perceive and interact with the data. Users see a simplified and customized view of the data that suits their needs.

2. Instance and Schema:

- **Instance**: An instance of a database refers to a snapshot or a specific state of the database at a particular moment in time. It's the actual data stored in the database, including all the rows and columns.

- **Schema**: A database schema defines the structure, organization, and constraints of the data stored in the database. It represents the logical view of the entire database. The schema includes definitions of tables, columns, keys, relationships, and integrity constraints.

3. Data Independence:
Data independence refers to the separation between the logical and physical aspects of data storage and retrieval. There are two types of data independence:

- **Logical Data Independence**: It means that changes in the logical structure of the database, such as adding new tables or modifying existing ones, should not affect the application programs that use the database. Logical data independence is typically achieved by using views and schemas.
- **Physical Data Independence**: It means that changes in the physical structure of the database, like moving to a different storage system or changing the indexing method, should not affect the application programs either. This level of independence is usually guaranteed through a DBMS, which abstracts the physical storage details from users and applications.

Advantages of Data Independence:

1. **Flexibility**: It allows changes to be made in the database structure without impacting the application programs or users.
2. **Simplified Development**: Developers can focus on designing and modifying the logical structure without worrying about the physical storage details.
3. **Security**: Logical data independence ensures that data access controls remain intact even when the physical storage is altered.
4. **Performance Optimization**: Physical data independence allows for the optimization of data storage and retrieval methods without affecting the logical structure.

1.4 Overall Structure of DBMS.

A Database Management System (DBMS) is a sophisticated software system that enables users to create, read, update, and manage data within a structured environment. The overall structure of a DBMS is designed to ensure data integrity, security, and accessibility. The key components and their functionalities are outlined below:

1. Database Engine: This is the core service for storing, processing, and securing data. It provides controlled access and rapid transaction processing to meet the requirements of the most demanding applications within the DBMS.

2. Database Query Processor: It interprets and executes the database queries, often written in SQL (Structured Query Language). This component is responsible for query optimization, which selects the most efficient strategy for query processing.

3. Database Manager (or Database Management System): The manager serves as an interface between the physical database, the application programs, and the users. It ensures data integrity, manages access rights, and maintains database security.

4. Storage Manager: Responsible for managing the space on storage mediums, like hard disks. It encompasses the actual database storage files, the data dictionary, indices, and data structures like B-trees and hash tables that optimize search.

5. Transaction Manager: Ensures the DBMS's ACID properties (Atomicity, Consistency, Isolation, Durability) are upheld. It manages all transactions ensuring they are completed successfully and any failures are adequately handled.

6. Buffer Manager: It handles the way data is stored in memory (buffers) for faster retrieval. The buffer manager will decide when to load data into memory and when to write it back to disk.

7. DDL Compiler: Processes schema definitions, specified in the Data Definition Language (DDL), and stores metadata in the data dictionary.

8. Metadata Repository (or Data Dictionary): A repository storing metadata—data about data. It contains definitions of databases, tables, schemas, data types, and other elements.

9. Application Programming Interface (API): Provides a set of routines and protocols which allow the creation of applications that access the features or data of an operating system, application, or other service of the DBMS.

10. Concurrency Control Manager: Ensures multiple database transactions can occur simultaneously without compromising the integrity of the database.

11. Backup and Recovery Management: It oversees the backup processes and ensures that data can be recovered after any failure, be it system crashes or errors.

12. User Interface & SQL Compilers: This component allows users to interact with the database. SQL compilers convert query statements into a series of low-level instructions for the query processor.

13. Report Generator: This tool allows users to design and generate reports from data within the DBMS, based on user-specific requirements.

1.5 Data Modelling: Record based logical model- Relational, Network, Hierarchical

Data modelling is a structured approach used to define and analyze data requirements needed to support business processes. In the context of databases, it's the process of creating a visual representation (often called a schema or diagram) of the database's structure. This abstract representation organizes the data into specific structures, facilitating efficient data management. Record-based logical models are some of the most prevalent ways to model this data.

1. Relational Model:

- **Concept**: Proposed by E.F. Codd in 1970, the relational model is based on the concept of relation which, in essence, is akin to a table.
- **Components**: Relations (tables), attributes (columns), and tuples (rows).
- **Characteristics**:
  - Data is presented as a set of tables.
  - Relationships between data items are expressed by means of tables.
  - Tables have attributes, which are the properties or characteristics of the entity.
- **Advantages**:
  - Flexibility in data retrieval (using SQL).
  - Easier to understand and implement as it's based on the logic of sets and predicate logic.
  - Data independence from the applications.

2. Network Model:

- **Concept**: The network model allows each record to have multiple parent and child records, forming a generalized graph structure.
- **Components**: Data sets (similar to tables) and data relationships.
- **Characteristics**:
  - Represents data as records connected via links (pointers).
  - A single data structure can have multiple parent records.
- **Advantages**:
  - Complex data relationships can be expressed.
  - Can represent redundancy in data more naturally.
- **Drawbacks**:
  - As the structure is complex, the operations tend to be less efficient.
  - Lack of standards across its implementations.

3. Hierarchical Model:

- **Concept**: In the hierarchical model, data is organized in a tree-like structure, where each record has only one parent but can have multiple children.
- **Components**: Records and fields. The relationship is established using parent-child links.
- **Characteristics**:
  - Represents data in a tree structure.

    - Each record has only one owner or parent.
- **Advantages**:
    - Efficient data retrieval due to the prescriptive tree-like structure.
    - Represents parent-child (one-to-many) relations naturally.
- **Drawbacks**:
    - Doesn't represent many-to-many relationships well.
    - Rigid structure; adding a new field or record can be complex.

1.6 Data Modelling Using the E-R Model: Entity Relationship Model, Strong Entity set, Weak Entity set, Types of Attributes, E-R Diagrams.

The Entity-Relationship (E-R) model is a high-level conceptual data model that lays the groundwork for the conceptual design of databases. It provides a graphical representation to visualize the relationships between different sets of data.

1. Entity Relationship Model:

An **Entity Relationship (ER) Model** is a type of data model that visually represents the logical structure of databases. It uses different notations to depict entities, attributes, relationships, and their relevant constraints.

- **Entity**: Represents a real-world object or concept that holds data. For example, 'Student', 'Course', and 'Teacher' could all be entities in a university database.
- **Relationship**: Describes how entities are related to one another. For instance, a student 'enrolls in' a course.

2. Strong Entity Set:

A **Strong Entity Set** is an entity that can exist independently of any other entity in the schema. It has a primary key which uniquely identifies its instances. For example, 'Student' could be a strong entity if each student is uniquely identifiable by a 'Student_ID'.

3. Weak Entity Set:

A **Weak Entity Set**, on the other hand, cannot exist without a related entity. It doesn't have any primary key of its own but relies on the primary key of its linked strong entity. An example could be 'Section' of a course, which can't exist without the respective course.

4. Types of Attributes:

Attributes are properties or characteristics of entities. They are of various types:

- **Simple Attribute**: Atomic attributes that can't be divided further. For instance, 'StudentName'.
- **Composite Attribute**: Can be divided into smaller parts which represent more basic attributes with independent meanings. For instance, 'Address' can be divided into 'Street', 'City', 'State', and 'ZipCode'.

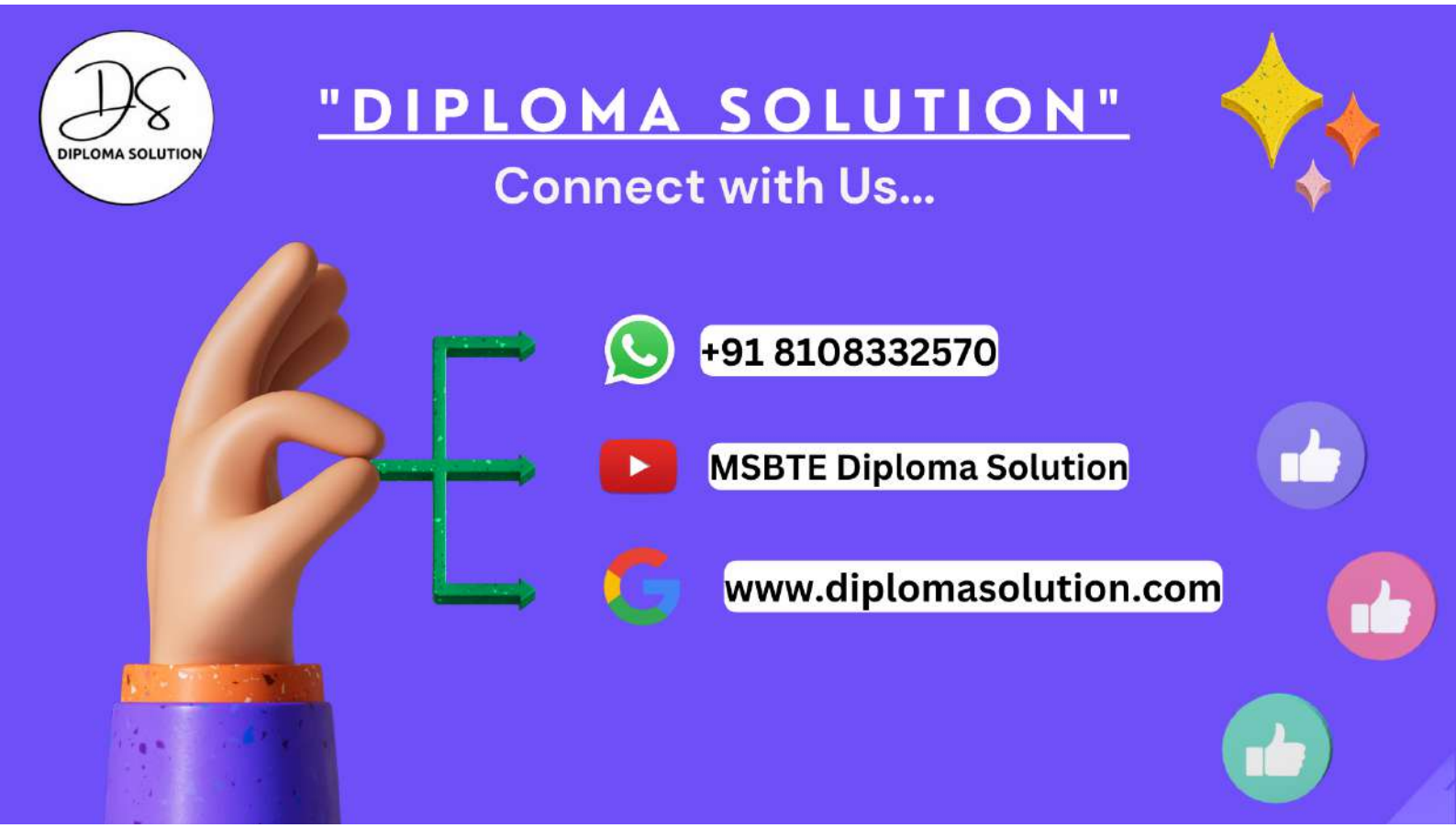
- **Derived Attribute**: Their values can be derived from other attributes in the database. For example, 'Age' can be derived from 'DateOfBirth'.
- **Multi-valued Attribute**: Can have multiple values. For instance, 'PhoneNumbers' for a student if students are allowed to have more than one phone number.
- **Key Attribute**: Attributes that uniquely identify an entity within an entity set.
- **Single-valued Attribute**: Holds a single value. For example, 'DateOfBirth'.

5. E-R Diagrams:

An **E-R Diagram** visually represents the structure of a database. It uses rectangles to represent entities, ovals to represent attributes, diamonds to denote relationships, and lines to show relationships between entities and their attributes.

In the E-R diagram:

- A line between two entities represents a relationship.
- Lines connecting entities to attributes indicate which attributes belong to which entity.
- Double ovals represent multi-valued attributes.
- Dashed ovals indicate derived attributes.
- Double rectangles denote weak entities.


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com

Unit- II Relational Data Model

2.1 Fundamentals of RDBMS- Record, fields, data types, tables and database

A Relational Database Management System (RDBMS) is a type of database management system that organizes data into structured tables. RDBMSs provide a set of concepts and functionalities that ensure data integrity, security, and efficiency. Let's delve into the key components and terms associated with RDBMS:

1. Record:

- A **record** (often referred to as a row) is a single data entry in a table that contains specific pieces of information. For instance, in a table named 'Students', each record might represent a single student's details.

2. Fields:

- Fields, often known as **columns** or attributes, represent specific categories of data within a table. In our 'Students' table, for example, the fields might include 'Student_ID', 'FirstName', 'LastName', and 'DateOfBirth'.

3. Data Types:

- Every field in a table has an associated **data type**. This dictates the kind of data the field can store. Common data types include:
  - **INTEGER**: For numeric values without decimal points.
  - **VARCHAR**: For variable-length alphanumeric characters.
  - **DATE**: For date values.
  - **FLOAT** or **DOUBLE**: For numbers with decimal points.
  - **BOOLEAN**: For true or false values.
  - **BLOB**: For binary data like images or multimedia.

4. Tables:

- A **table** is a collection of related data held in structured form within a database. Tables consist of rows (records) and columns (fields). Each table in an RDBMS has a unique name, and its structure is defined by its set of fields (data categories). A table's primary purpose is to store related data in a structured, easy-to-access manner. For instance, 'Courses', 'Teachers', and 'Enrollments' might all be separate tables within a university database.

5. Database:

- A **database** is a collection of related tables, queries, reports, views, and other objects. It serves as a centralized repository for data that can be easily accessed, managed, and updated. An RDBMS might manage multiple databases, each pertaining to different applications or functions within an organization.

2.2 Concept of RDBMS, E.F.Codd's Rule for RDBMS, Key concepts- Candidate key, Primary key, Foreign key.

RDBMS - Relational Database Management System:

An RDBMS is a database management system that organizes data into structured tables, ensuring relationships among these data tables using a set of integrity constraints. It provides a way to maintain, retrieve, and update the stored data efficiently and ensures data integrity, security, and consistency.

E.F. Codd's Rules for RDBMS:

Dr. Edgar F. Codd, the pioneer of the relational model, defined a set of thirteen rules (sometimes referred to as Codd's 12 rules, with rule zero included) that a database system must adhere to in order to be considered truly relational:

1. **Rule 0 (Foundation Rule)**: For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage databases entirely through its relational capabilities.
2. **Rule 1 (Information Rule)**: All information (including metadata) is to be represented as stored values in tables.
3. **Rule 2 (Guaranteed Access Rule)**: Every individual value in the database must be accessible by specifying a table name, primary key value, and a column/attribute name.
4. **Rule 3 (Systematic Treatment of Null Values)**: The DBMS must allow each field to remain null, which is distinct from an empty string or zero.
5. **Rule 4 (Database Description)**: The database description is itself stored in relational tables.
6. **Rule 5 (Comprehensive Data Sublanguage Rule)**: A relational system may support several languages, but it must fully support at least one relational language with a linear syntax.
7. **Rule 6 (View Updating Rule)**: All views that are theoretically updatable should be updatable by the system.
8. **Rule 7 (Set-Level Operations)**: The system must support set-level insertion, update, and deletion operations.
9. **Rule 8 (Physical Data Independence)**: Changes to the physical level (how data is stored) should not require changes at the logical level (tables, columns, etc.).
10. **Rule 9 (Logical Data Independence)**: Logical data changes, such as adding a column, should not require changes in application programs.
11. **Rule 10 (Integrity Independence)**: Integrity constraints must be defined separately from application programs and stored in the system catalog.
12. **Rule 11 (Distribution Independence)**: The end-user should not need to know the method of data distribution.
13. **Rule 12 (Nonsubversion Rule)**: The system must not allow bypassing the relational structure to alter data.

Key Concepts:

1. **Candidate Key**: An attribute or set of attributes that can uniquely identify a tuple (record) in a relation (table) is termed a candidate key. A relation may have multiple candidate keys.
2. **Primary Key**: Out of the candidate keys, one key is chosen as the main key to identify records uniquely within a relation. This chosen key is called the primary key. It cannot have null values, and its value must be unique.
3. **Foreign Key**: It is an attribute or a set of attributes in a table that serves as a link to the primary key of another table. It enforces referential integrity in the relationship between two tables.

2.3 Normalization: Normalization Concepts, Need of Normalization, Types of Normalization - 1NF,2NF,3NF

Normalization is a systematic approach for organizing data in databases. Its primary goal is to eliminate redundancy and prevent undesirable characteristics like insertion, update, and deletion anomalies. By distributing data into different tables and ensuring relationships among them, normalization ensures the efficient and consistent storage of data.

Normalization Concepts:

1. **Redundancy**: It refers to the unnecessary repetition of data in the database.
2. **Anomalies**: Problems that arise due to redundancy, including:
    - **Insertion Anomaly**: When certain data cannot be inserted without the presence of other data.
    - **Update Anomaly**: When particular data needs to be updated in multiple places.
    - **Deletion Anomaly**: When deleting certain data unintentionally results in loss of other data.

Need for Normalization:

1. **Eliminate Redundant Data**: Reducing or eliminating data duplication saves storage and improves data retrieval efficiency.
2. **Data Integrity**: Ensures data remains consistent and accurate throughout the database.
3. **Facilitate Efficient Data Search and Queries**: Optimized table structures allow faster and more accurate data retrieval.
4. **Database Scalability**: Normalized databases can scale more efficiently as data grows.
5. **Reduce Update Anomalies**: Streamlined data structures reduce the chances of anomalies during data operations.

Types of Normalization:

1. **First Normal Form (1NF)**:
   - Every column must contain atomic (indivisible) values.
   - Each entry in a column must be of the same data type.
   - Each column must have a unique name.
   - The order in which data is stored does not matter.
2. **Second Normal Form (2NF)**:
   - The table is in 1NF.
   - All non-key attributes must be functionally dependent on the primary key. This means that there should be no partial dependency of any column on the primary key.
3. **Third Normal Form (3NF)**:
   - The table is in 2NF.
   - There are no transitive dependencies of non-key attributes on the primary key. This means that non-key columns must be functionally dependent only on the primary key.

2.4 Introduction to Structured Query Language, Data Types in SQL, components of SQL- DDL, DML, DCL, DQL

SQL, or Structured Query Language, is a standardized programming language specifically designed for managing and manipulating relational databases. It provides the means to create, retrieve, update, and delete database records. SQL has become the industry standard for RDBMS (Relational Database Management Systems), allowing users to maintain and query large datasets in real-time.

Data Types in SQL:

Different databases support various data types, but some common SQL data types include:

1. **INTEGER**: Represents whole numbers.
2. **FLOAT** or **REAL**: Represents numbers with decimal points.
3. **VARCHAR** or **VARCHAR2**: Represents variable-length character strings.
4. **CHAR**: Represents fixed-length character strings.
5. **DATE**: Represents date values.
6. **TIMESTAMP**: Represents date and time values.
7. **BOOLEAN**: Represents true or false values.
8. **BLOB**: Represents binary data, like images.

Components of SQL:

1. **Data Definition Language (DDL)**:
   - **CREATE**: Used to create database objects like tables, indexes, or views.
   - **ALTER**: Used to modify existing database objects.
   - **DROP**: Used to delete database objects.
   - **TRUNCATE**: Used to remove all records from a table but retain the structure for future use.
2. **Data Manipulation Language (DML)**:
   - **SELECT**: Used to retrieve data from one or more tables.
   - **INSERT**: Used to add records to a table.
   - **UPDATE**: Used to modify records in a table.
   - **DELETE**: Used to remove records from a table.
3. **Data Control Language (DCL)**:
   - **GRANT**: Provides specific privileges to users or roles.
   - **REVOKE**: Removes specific privileges from users or roles.
4. **Data Query Language (DQL)**:
   - Primarily uses the **SELECT** statement to query and retrieve data from databases.

2.5 DDL Commands: CREATE, ALTER. DROP, TRUNCATE, DESC, RENAME

Data Definition Language (DDL) commands are utilized to define or modify the structure of database objects such as tables, indexes, and schemas. DDL commands do not manipulate the actual data, but instead, they affect the architecture of the database.

1. **CREATE**:

- This command is used to create a new table, index, or other database objects.
- Example:

```
CREATE TABLE Students ( StudentID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50), Age INT );
```

2. **ALTER**:

- The ALTER command is used to modify an existing database object like adding a column, deleting a column, or changing the data type of a column in a table.
- Example:

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

3. **DROP**:

- This command removes an existing database object. Once executed, the DROP command deletes the object and its data permanently.
- Example:

```
DROP TABLE Students;
```

4. **TRUNCATE**:

- The TRUNCATE command is used to delete all records from a table, but it keeps the structure of the table for future use. It's often faster than DELETE as it doesn't log individual row deletions.
- Example:

TRUNCATE TABLE Students;

5. **DESC (Describe)**:

- The DESC command provides a description of a table, such as its column names and data types. It's particularly useful when you need to understand the structure of a table.
- Example:

DESC Students;

6. **RENAME**:

- The RENAME command is used to rename an existing table or a column.
- Example (to rename a table):

RENAME TABLE Students TO Alumni;

2.6 Data Integrity Constraint: Types of Data Integrity Constraint: 1/0 constraint- Primary key, foreign key, Unique key constraint, Business Rule Constraint NulL Not Null and Check constraint.

Data integrity constraints in databases ensure that the information remains accurate, consistent, and reliable during its entire lifecycle. These constraints uphold the quality of the data and prevent accidental damage to the dataset. Let's explore the primary types of data integrity constraints:

1. **Entity Integrity (1/0 Constraint)**:
    - **Primary Key**: It uniquely identifies every record in a database table. A primary key does not accept null values and must contain a unique value for each row of data.
- Example:

CREATE TABLE Students (StudentID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50));

- **Unique Key Constraint**: Similar to a primary key, the unique key constraint ensures that all values in a column are unique. The difference is that a table can have multiple unique keys, but only one primary key, and unique keys can have null values.

- Example:

CREATE TABLE Users (UserID INT PRIMARY KEY, Email VARCHAR(100) UNIQUE);

2. **Referential Integrity**:
    - **Foreign Key**: This constraint ensures that the relationship between tables remains consistent. A foreign key in one table references the primary key of another table. It helps maintain referential integrity by ensuring that the relationship between two tables matches and is valid.

- Example:

CREATE TABLE Orders (OrderID INT PRIMARY KEY, StudentID INT FOREIGN KEY REFERENCES Students(StudentID));

3. **Domain Integrity**:
    - **Check Constraint**: It limits the values that a column can have. This constraint ensures that the values match the defined criteria.

- Example:

CREATE TABLE Employees (EmpID INT PRIMARY KEY, Age INT CHECK (Age >= 18));

4. **Null and Not Null**:
    - These constraints define whether a column can accept null values or not.

- **NULL**: Indicates that the column can have a null value.
- **NOT NULL**: Ensures that the column always has a value and cannot remain empty.
- Example:

CREATE TABLE Employees (EmpID INT NOT NULL, Phone VARCHAR(15) NULL);

5. **Business Rule Constraint**:
    - These are specific rules or constraints applied to the data that adhere to the business's logic or operations. For instance, a business rule might specify that an employee's salary cannot be below a certain threshold.

2.7 DML Commands: INSERT, UPDATE, DELETE

Data Manipulation Language (DML) commands deal with the manipulation of data stored in the database. They don't alter the schema or structure of the database but focus on inserting, modifying, or deleting data. Here are the primary DML commands:

1. **INSERT**:
    - The **INSERT** command is used to add new rows of data into a table. There are different ways to use the **INSERT** command based on how you want to add the data.

- **Inserting Full Rows**:

```
INSERT INTO Students (StudentID, FirstName, LastName) VALUES (1, 'John', 'Doe');
```

- **Inserting Specific Columns**:

```
INSERT INTO Students (FirstName, LastName) VALUES ('Jane', 'Doe');
```

2. **UPDATE**:
    - The **UPDATE** command modifies existing data within the table. It's typically used with the **WHERE** clause to specify which rows need updating.

- **Updating Specific Rows**:

```
UPDATE Students SET LastName = 'Smith' WHERE StudentID = 1;
```

- **Updating Multiple Columns**:

```
UPDATE Students SET FirstName = 'James', LastName = 'Johnson' WHERE StudentID = 2;
```

3. **DELETE**:
    - The **DELETE** command is used to remove rows from a table. Like the **UPDATE** command, **DELETE** is usually used with the **WHERE** clause to specify which rows to delete. It's important to use this command with caution, as it can remove data permanently.

- **Deleting Specific Rows**:

```
DELETE FROM Students WHERE StudentID = 1;
```

- **Deleting All Rows** (without removing the table structure):

```
DELETE FROM Students;
```


2.8 DCL Commands: COMMIT, SAVEPOINT, ROLLBACK, GRANT, and REVOKE.

Data Control Language (DCL) is a subset of SQL commands that handle the rights and permissions for the data and database objects. It also deals with the transactional control commands. Let's delve into the primary DCL commands:

1. **COMMIT**:
    - The **COMMIT** command is used to permanently save all the changes made during the current transaction to the database. Once you use **COMMIT**, the transaction is complete, and the changes cannot be undone.

- **Usage**:

```
COMMIT;
```

2. **SAVEPOINT**:
    - The **SAVEPOINT** command allows you to mark a specific point within a transaction. This point can later be used to roll back changes to that particular point, rather than rolling back the entire transaction.

- **Usage**:

```
SAVEPOINT savepoint_name;
```

3. **ROLLBACK**:
    - The **ROLLBACK** command is used to undo the transactions that have not been saved to the database. If used without a savepoint, it undoes all the changes in the ongoing transaction. If specified with a savepoint, it undoes the changes up to that savepoint.

- **Usage without Savepoint**:

```
ROLLBACK;
```

- **Usage with Savepoint**:

```
ROLLBACK TO savepoint_name;
```

4. **GRANT**:
    - The **GRANT** command is used to provide specific privileges to users or roles. These privileges can relate to creating tables, inserting data, updating data, etc.

- **Usage**:

```
GRANT SELECT, INSERT ON Students TO user_name;
```

5. **REVOKE**:
    - The **REVOKE** command is used to remove specific privileges granted to users or roles. It's the opposite of the **GRANT** command.

- **Usage**:

```
REVOKE INSERT ON Students FROM user_name;
```


2.9 DQL Commands: SELECT.

Data Query Language (DQL) is a subset of SQL commands used for querying the database to fetch information. The primary, and often the only command in DQL, is the **SELECT** command.

1. **SELECT**:
    - The **SELECT** command is utilized to retrieve data from one or more tables in the database. The retrieved data is presented in the form of a result table or result set.

- **Basic Usage**:

SELECT column1, column2 FROM table_name;

This fetches specific columns from a table.

- **Fetch All Columns**:

SELECT * FROM table_name;

This fetches all columns from a table.

- **With Conditions** (Using the WHERE Clause):

SELECT column1, column2 FROM table_name WHERE condition;

This fetches specific columns based on a condition.

- **Sorting Results** (Using the ORDER BY Clause):

SELECT column1, column2 FROM table_name ORDER BY column1 ASC;

This sorts the results based on **column1** in ascending order.

- **Aggregation**:

SELECT COUNT(column1), AVG(column2) FROM table_name;

This fetches the count of **column1** values and the average of **column2** values.

- **Grouping Results** (Using the GROUP BY Clause):

SELECT column1, COUNT(column2) FROM table_name GROUP BY column1;

This groups the results by unique values in **column1** and counts the **column2** values for each group.

- **Joining Tables**:

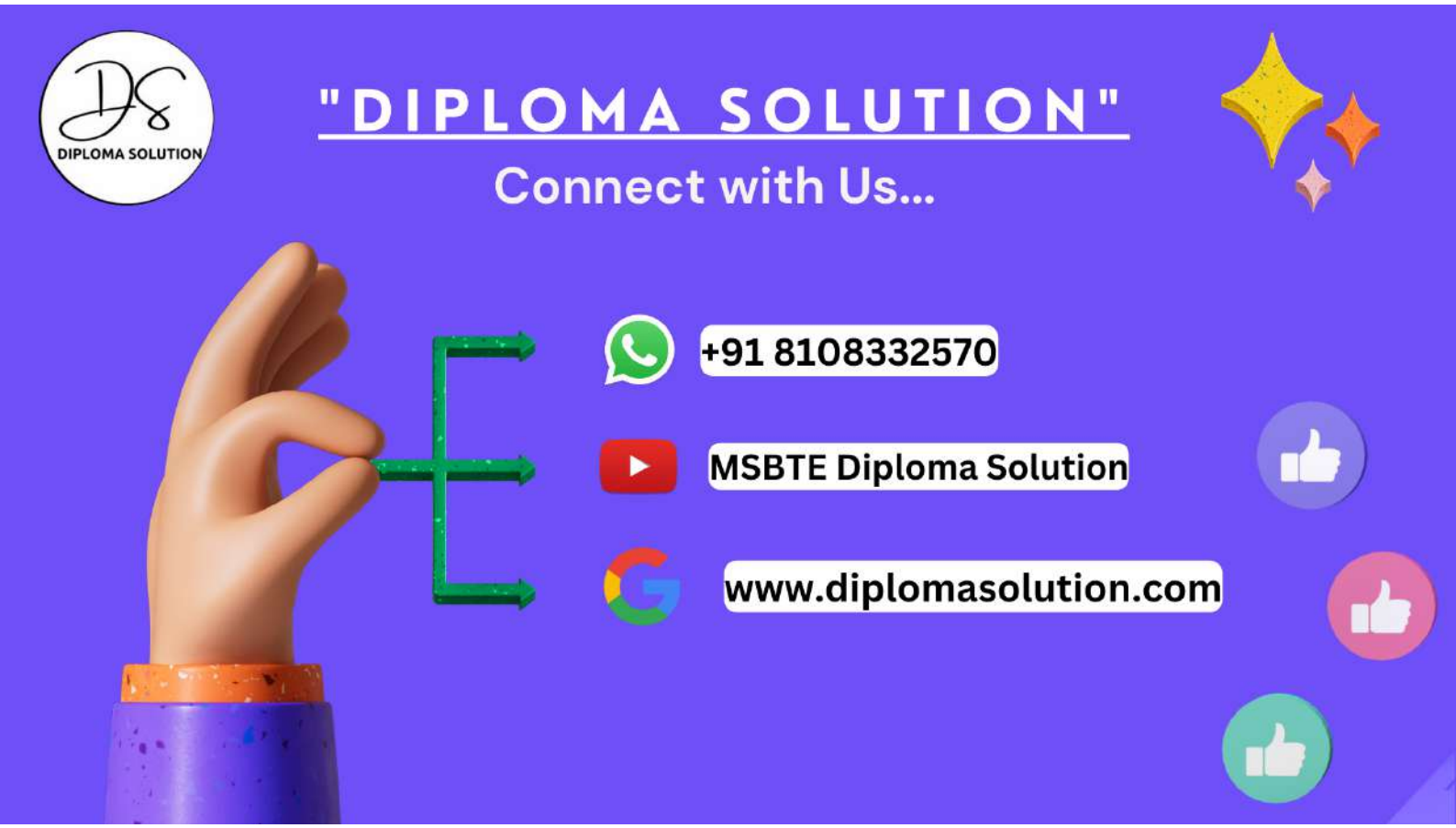
SELECT table1.column1, table2.column2 FROM table1 JOIN table2 ON table1.common_column = table2.common_column;

This fetches data from two tables by joining them on a common column.

2.10 SQL Operators: Arithmetic Operators, Comparison Operators, Logical Operators. Set Operators. Range Searching operators- Between, Pattern matching operators-Like.

SQL operators are used to perform specific operations on data within the database. They can be broadly categorized into arithmetic, comparison, logical, set, range searching, and pattern matching operators. Let's dive into each category:

1. **Arithmetic Operators**:
    - These operators perform mathematical operations.
        - **+** : Addition


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com

- **-** : Subtraction
- ***** : Multiplication
- **/** : Division
- **%** : Modulus (remainder after division)

Example:

```
SELECT salary, salary + 500 AS incremented_salary FROM employees;
```

2. **Comparison Operators**:
   - Used for comparing two values.
     - **=** : Equal to
     - **!= or <>** : Not equal to
     - **>** : Greater than
     - **<** : Less than
     - **>=** : Greater than or equal to
     - **<=** : Less than or equal to

Example:

```
SELECT name FROM students WHERE age >= 18;
```

3. **Logical Operators**:
   - Used to combine multiple conditions.
     - **AND** : True if both conditions are true
     - **OR** : True if at least one condition is true
     - **NOT** : True if the condition is false

Example:

```
SELECT id FROM users WHERE age > 18 AND city = 'New York';
```

4. **Set Operators**:
   - Used to combine rows from two or more tables.
     - **UNION** : Returns distinct values from both tables
     - **UNION ALL** : Returns all values, including duplicates
     - **INTERSECT** : Returns rows common to both tables
     - **EXCEPT** : Returns rows from the first table that aren't present in the second

Example:

```
SELECT name FROM table1 UNION SELECT name FROM table2;
```

5. **Range Searching Operators**:
   - **BETWEEN**: Used to filter values within a specific range.

Example:

```
SELECT product FROM inventory WHERE price BETWEEN 10 AND 50;
```

6. **Pattern Matching Operators**:

- **LIKE**: Used in conjunction with wildcards to search for patterns within data.
  - **%** : Represents zero or multiple characters
  - **_** : Represents a single character

Example:

```
SELECT firstname FROM users WHERE lastname LIKE 'Smith%';
```

Unit III- Interactive SQL and Advance SQL: SQL Performance Tuning

3.1 In-built Functions: String, Arithmetic,

SQL offers a wide range of in-built functions to perform various operations on data. These functions can be categorized based on their primary purpose. Let's explore the string and arithmetic functions in SQL:

1. **String Functions**:

These functions allow you to manipulate and retrieve information about string data.

- **CONCAT(str1, str2)**: Combines two strings.
- **LENGTH(str)**: Returns the number of characters in a string.
- **UPPER(str)** and **LOWER(str)**: Convert the string to upper or lower case respectively.
- **TRIM(str)**: Removes leading and trailing spaces from a string.
- **SUBSTRING(str, start, length)**: Extracts a portion of a string.
- **CHAR_LENGTH(str)** or **CHARACTER_LENGTH(str)**: Returns the number of characters in a string.
- **POSITION(substring IN string)**: Returns the position of the first occurrence of a substring in a string.
- **REPLACE(str, old_string, new_string)**: Replaces all occurrences of a substring with another substring.

Example:

```
SELECT UPPER(firstname) FROM users;
```

2. **Arithmetic Functions**:

These functions allow you to perform mathematical operations on data.

- **ABS(n)**: Returns the absolute value of n.
- **ROUND(n, d)**: Rounds the number n to d decimal places.
- **CEIL(n)** or **CEILING(n)**: Returns the smallest integer greater than or equal to n.
- **FLOOR(n)**: Returns the largest integer less than or equal to n.
- **POW(n, p)** or **POWER(n, p)**: Returns the value of n raised to the power of p.
- **SQRT(n)**: Returns the square root of n.
- **MOD(n, m)**: Returns the remainder of n divided by m.
- **EXP(n)**: Raises the number e (approximately equal to 2.718) to the power of n.
- **LN(n)**: Returns the natural logarithm of n.

Example:

```
SELECT ROUND(salary) FROM employees;
```

3.2 Date and time, Aggregate functions.

In SQL, there are various in-built functions designed to work with date and time values, as well as aggregate functions which operate on a set of values and return a single value. Here's a detailed overview:

1. **Date and Time Functions**:

These functions allow you to manipulate, format, and retrieve information related to date and time data.

- **NOW()**: Returns the current date and time.
- **CURDATE()**: Returns the current date.
- **CURTIME()**: Returns the current time.
- **DATE(datetime)**: Extracts the date part of a date or datetime expression.
- **YEAR(date)**, **MONTH(date)**, **DAY(date)**: Extracts the year, month, or day from a date.
- **HOUR(time)**, **MINUTE(time)**, **SECOND(time)**: Extracts the hour, minute, or second from a time.
- **DATEDIFF(date1, date2)**: Returns the difference between two dates.
- **DATE_ADD(date, INTERVAL value type)**: Adds a specified time interval to a date.
- **DATE_FORMAT(date, format)**: Displays date/time data in a specified format.

Example:

```
SELECT DATE_ADD(CURDATE(), INTERVAL 10 DAY) AS 'Date After 10 Days';
```

2. **Aggregate Functions**:

These functions perform calculations on a set of values and return a single summarizing value.

- **COUNT(column)**: Returns the number of rows for the specified column.
- **SUM(column)**: Returns the sum of all values in the specified column.
- **AVG(column)**: Returns the average value of the specified column.
- **MIN(column)**: Returns the smallest value in the specified column.
- **MAX(column)**: Returns the largest value in the specified column.
- **GROUP_CONCAT(column)**: Returns a concatenated string of values from multiple rows.
- **STD(column)**: Returns the standard deviation of the specified column.
- **VAR_POP(column)** and **VAR_SAMP(column)**: Return the population and sample variance of the specified column, respectively.

Note: These functions are typically used with the **GROUP BY** clause to aggregate data based on one or more columns.

Example:

```
SELECT department, COUNT(employee_id) AS 'Number of Employees' FROM employees GROUP BY department;
```

3.3 Queries using Group by, having, and Order by clause, Joins: Inner and Outer Join, Sub queries.

Delving deeper into SQL, we encounter advanced query structures that enable us to extract intricate data patterns and insights. Some essential elements include the **GROUP BY**, **HAVING**, and **ORDER BY** clauses, as well as various types of joins and sub-queries. Here's a detailed look:

1. **GROUP BY Clause**:

The **GROUP BY** clause is used in collaboration with aggregate functions to group rows that have the same values in specified columns.

Example:

```
SELECT department, COUNT(employee_id) AS 'Number of Employees' FROM employees GROUP BY department;
```

2. **HAVING Clause**:

The **HAVING** clause is used to filter the results of a **GROUP BY** query based on a condition applied to the result of an aggregate function.

Example:

```
SELECT department, AVG(salary) AS 'Average Salary' FROM employees GROUP BY department HAVING AVG(salary) > 50000;
```

3. **ORDER BY Clause**:

The **ORDER BY** clause is used to sort the result set based on one or more columns, either in ascending (**ASC**) or descending (**DESC**) order.

Example:

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

4. **Joins**:

Joins are used to combine rows from two or more tables based on a related column.

- **Inner Join**: Returns rows when there is a match in both tables.

```
SELECT employees.name, departments.name FROM employees INNER JOIN departments ON employees.department_id = departments.id;
```

- **Outer Join**: Can be a left, right, or full outer join. Returns rows when there is a match in one of the tables.

```
SELECT students.name, courses.name FROM students LEFT OUTER JOIN courses ON students.course_id = courses.id;
```

5. **Sub-queries**:

Sub-queries or nested queries are queries embedded within another SQL query. They can be used in various places within a query, including the **WHERE**, **FROM**, and **SELECT** clauses.

Example:

```
SELECT name FROM employees WHERE department_id IN (SELECT id FROM departments WHERE name = 'Sales');
```

3.4 Views: Concept of View, The Create View Command, Updating Views, Views and Joins, and Sub queries, Dropping Views.

Views in SQL are powerful tools that allow you to simplify complex queries, enhance security, and manage data more efficiently. Let's explore the concept of views, how to create and update them, their relationship with joins and subqueries, and how to drop views.

1. Concept of View:

A view is a virtual table created by a query. It allows you to present the data from one or more tables in a structured, user-friendly format without duplicating the actual data. Views can be used for simplifying complex queries, enforcing security, and providing a customized perspective on the data.

2. The CREATE VIEW Command:

To create a view, you use the **CREATE VIEW** command. Here's the basic syntax:

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

Example:

CREATE VIEW high_salary_employees AS SELECT name, salary FROM employees WHERE salary > 50000;

3. Updating Views:

Views are typically read-only, meaning you can't use them to update the underlying tables directly. However, you can update a view if it's defined with the **WITH CHECK OPTION** clause. This allows you to insert, update, or delete records through the view under certain conditions.

4. Views and Joins:

Views can be created using joins to combine data from multiple tables. This is especially useful when you need to present a unified view of related data without exposing the underlying complexity.

Example:

CREATE VIEW employee_department AS SELECT employees.name, departments.name FROM employees INNER JOIN departments ON employees.department_id = departments.id;

5. Views and Subqueries:

Views can be built on top of subqueries, making it easier to manage complex queries. You can create a view that encapsulates a subquery, allowing you to reuse it throughout your database.

6. Dropping Views:

To remove a view, you use the **DROP VIEW** command. Be cautious when dropping views, as it permanently deletes them.

Example:

DROP VIEW high_salary_employees;

3.5 Sequences: Creating Sequences, Altering Sequences, Dropping Sequences.

In database management systems, sequences are database objects used to generate unique numbers. They are commonly employed to produce primary keys automatically. This article will dive into the specifics of creating, altering, and dropping sequences in SQL.

1. Creating Sequences:

To produce a sequence in SQL, you use the **CREATE SEQUENCE** command. The syntax generally looks like:

CREATE SEQUENCE sequence_name START WITH initial_value INCREMENT BY increment_value MINVALUE minimum_value MAXVALUE maximum_value CYCLE | NOCYCLE CACHE cache_size;

- **START WITH**: Determines the starting value of the sequence.
- **INCREMENT BY**: Specifies the value by which the sequence will increase.
- **MINVALUE** and **MAXVALUE**: Define the boundaries of the sequence.
- **CYCLE**: Allows the sequence to restart from the MINVALUE once it reaches the MAXVALUE.
- **NO CYCLE**: Prevents the sequence from cycling back.
- **CACHE**: Specifies the number of sequence numbers to be cached in memory for faster access.

Example:

CREATE SEQUENCE emp_seq START WITH 1 INCREMENT BY 1 MINVALUE 1 MAXVALUE 10000 NOCYCLE CACHE 20;

2. Altering Sequences:

If you wish to modify a sequence after it has been created, the **ALTER SEQUENCE** command comes into play.

ALTER SEQUENCE sequence_name [options];

The options you can modify include the increment value, max value, min value, whether the sequence should cycle, and more.

Example:

```
ALTER SEQUENCE emp_seq INCREMENT BY 2 MAXVALUE 20000;
```

3. Dropping Sequences:

To delete a sequence from the database, the **DROP SEQUENCE** command is utilized.

```
DROP SEQUENCE sequence_name;
```

Be cautious when executing this command, as once a sequence is dropped, it cannot be retrieved, and the sequence values may also be lost.

Example:

```
DROP SEQUENCE emp_seq;
```


3.6 Indexes: Index Types. Creating of an Index: Simple Unique, and

In the realm of databases, indexes play a pivotal role in expediting the retrieval of data. Essentially, an index serves as a lookup table that the database system uses to speed up the data search process. Let's delve into the different types of indexes and how they can be created.

1. Index Types:

- **Single-level Index**: A simple index that consists of two columns. One column contains a copy of the primary or candidate key of a table, and the second column contains a set of pointers holding the address of the disk block where that specific key value can be found.
- **Multilevel Index**: This type of index is similar to a single-level index but can have multiple levels. A multilevel index reduces the number of disk IO operations required to find an item.
- **Composite Index (Compound Index)**: An index on two or more columns of a table. It's beneficial when queries filter or sort by these columns.
- **Unique Index**: Enforces the uniqueness of the values in a set of columns, so the combination of values in these columns are unique across all the rows in the table.
- **Clustered Index**: Determines the physical order of data in a table. A table can have only one clustered index.
- **Non-clustered Index**: A separate structure from the data table that reorders one or more selected columns' data. A table can have multiple non-clustered indexes.

2. Creating Indexes:

The general syntax for creating an index is:

CREATE INDEX index_name ON table_name (column1, column2, ...);

- **Simple Index**:

CREATE INDEX idx_column_name ON table_name (column_name);

- **Unique Index**: Enforces uniqueness for the values in the specified column(s).

CREATE UNIQUE INDEX idx_unique_column_name ON table_name (column_name);

- **Composite Index**:

CREATE INDEX idx_composite_name ON table_name (column1, column2);

3.7 Composite Index, Dropping Indexes

Composite Indexes:

A composite index, also known as a compound or concatenated index, encompasses two or more columns of a database table. Such indexes are useful when queries frequently filter, sort, or join using multiple columns.

The primary reason to utilize composite indexes is to improve query performance. However, as with all indexes, they add some overhead to data-modification operations (like insert, update, or delete), as the index also needs to be updated when modifications occur.

Creating a Composite Index:

To create a composite index, list multiple columns in the **CREATE INDEX** statement:

CREATE INDEX composite_idx_name ON table_name (column1, column2, ...);

For example, if you have a **orders** table and you frequently query on both **customer_id** and **order_date**, you might create a composite index like:

CREATE INDEX idx_orders_customer_date ON orders (customer_id, order_date);

Dropping Indexes:

As data evolves or application requirements change, you might find that an index is no longer beneficial, or it might even impede performance. In such scenarios, it's essential to remove these unnecessary indexes. This is where the **DROP INDEX** command becomes useful.

The syntax to drop an index is:

DROP INDEX index_name ON table_name;

For example, to drop the composite index we created above:

DROP INDEX idx_orders_customer_date ON orders;

Remember that dropping an index would eliminate the overhead associated with maintaining it during data modifications. However, the queries that benefitted from the index might become slower.

3.8 Synonyms: Creating Synonyms, Dropping Synonyms.

In the database world, particularly in SQL, a synonym is an alias or alternative name for a table, view, sequence, or other schema objects. Synonyms provide both convenience and a layer of security by abstracting the actual name and ownership of an object. They also come in handy for providing backward compatibility.
Let's delve into the creation and deletion of synonyms.

Creating Synonyms:
To create a synonym, the basic syntax is as follows:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM synonym_name FOR object_name [@dblink];
```

- **OR REPLACE**: Allows you to modify the synonym if it already exists.
- **PUBLIC**: Indicates the synonym is a public synonym and is accessible to all users. Remember though, the underlying object might have permissions that restrict its accessibility.
- **synonym_name**: The name of the synonym you are creating.
- **object_name**: The name of the database object for which the synonym is being created.
- **@dblink**: (Optional) The database link to a remote database where the object is located.

Example:
If you want to create a synonym for the **employees** table:

```
CREATE SYNONYM emp_synonym FOR employees;
```

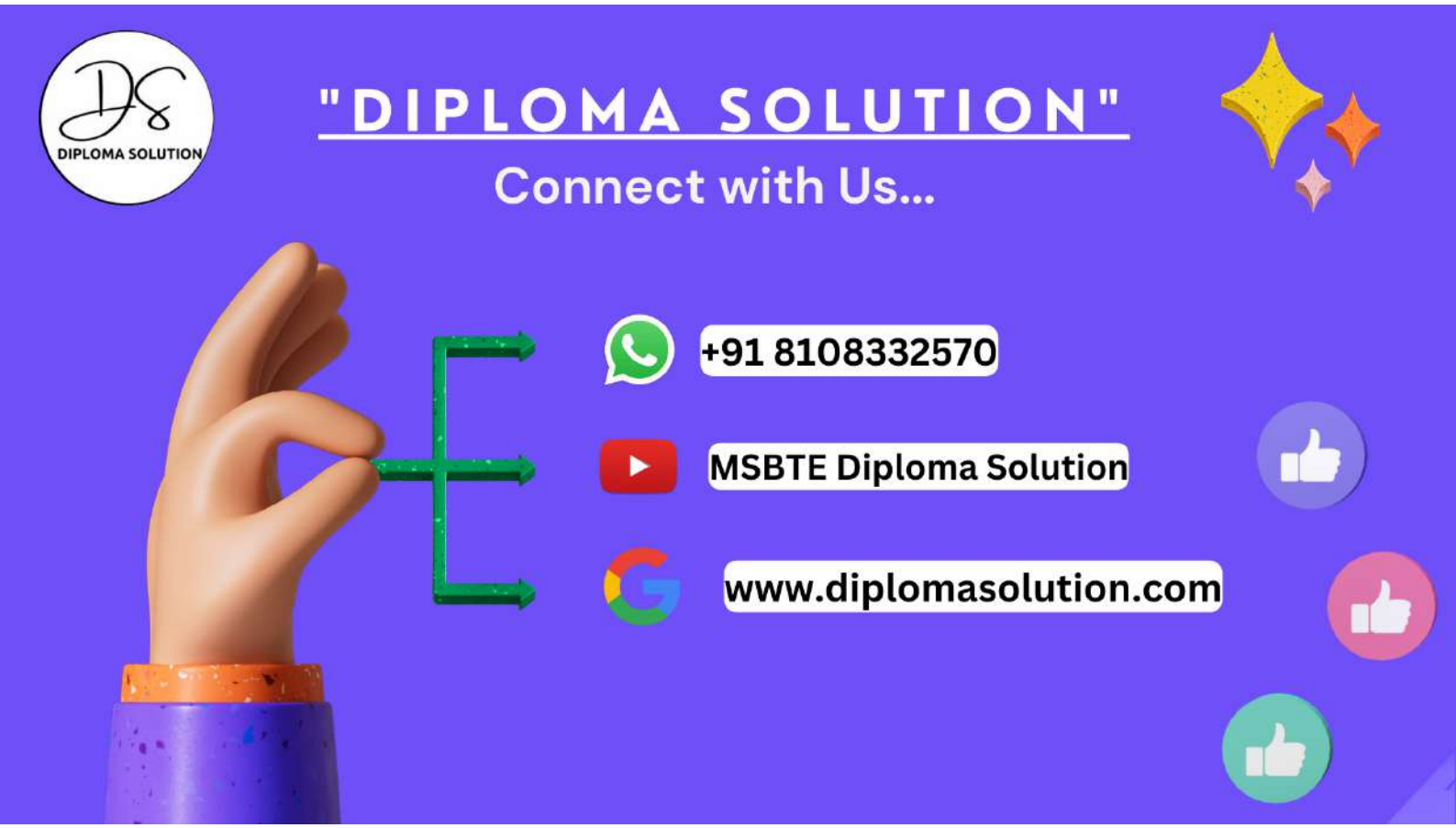
Dropping Synonyms:
There might be situations where a synonym is no longer required. In such cases, it's necessary to drop the synonym to maintain the cleanliness of the schema.
To drop a synonym, the syntax is:

```
DROP [PUBLIC] SYNONYM synonym_name [force];
```

- **PUBLIC**: Indicates that the synonym is a public synonym.
- **synonym_name**: The name of the synonym you want to drop.
- **force**: Drops the synonym even if it has dependent tables or user-defined types.

Example:
To drop the previously created synonym:

```
DROP SYNONYM emp_synonym;
```


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com

Unit IV - PL/SQL Programming

4.1 Introduction of PL/SQL, Advantages of PL/SQL, The PL/SQl" Block Structure, PL/SQL execution environment, PL/SQL data Types, Variables, Co11stants.

PL/SQL, which stands for "Procedural Language extensions to SQL", is Oracle's proprietary procedural extension to the SQL language. Essentially, it merges the power of SQL (a language designed for managing data in relational databases) with procedural constructs, delivering the means to create robust and scalable database applications.

Advantages of PL/SQL:

1. **Performance**: PL/SQL allows for entire blocks of SQL statements to be sent to the database at once, reducing network traffic and enhancing performance.
2. **Error Handling**: PL/SQL provides robust error-handling capabilities, making it easier to handle exceptions and debug.
3. **Security**: With PL/SQL, one can encapsulate business logic, which can prevent direct access to the database and its tables, thus enhancing security.
4. **Reusable Code**: With stored procedures and functions, code can be stored centrally and reused throughout the application.
5. **Portability**: PL/SQL is available on all Oracle platforms, making applications easily portable from one system to another.

The PL/SQL Block Structure:

A PL/SQL block consists of three sections:

1. **Declaration Section**: Variables, types, and constants are declared here but not initialized.
2. **Execution Section**: This is where the actual code, including SQL statements, is written and executed.
3. **Exception Section**: This is where you handle any errors that might occur in the execution section.

The basic structure looks like this:

DECLARE -- Declaration Section BEGIN -- Execution Section EXCEPTION -- Exception Handling Section END;

PL/SQL Execution Environment:

PL/SQL code can be executed in several environments, such as:

1. **Oracle Server**: It runs stored procedures, triggers, etc.
2. **Oracle Developer Tools**: Such as Oracle Forms or Oracle Application Express.
3. **Third-party Applications**: They can interface with the Oracle database.

PL/SQL Data Types:

PL/SQL supports a spectrum of data types, from basic types like NUMBER, VARCHAR2, and DATE to more complex types like records and tables. There are also LOB types for large objects and user-defined types.

Variables and Constants in PL/SQL:

- **Variables**: These are storage locations, having a name and a data type. They must be declared in the declaration section of a block. For instance:

DECLARE v_name VARCHAR2(50); BEGIN ... END;

- **Constants**: These are similar to variables, but once initialized, their value cannot change. For instance:

DECLARE c_pi CONSTANT NUMBER := 3.14159265359; BEGIN ... END;

4.2 Control Structure: Conditional Control, Iterative Control, Sequential Control.

Control structures guide the execution of a program. By determining the order in which statements or blocks of code run, they play a pivotal role in crafting efficient and logical programs. The three primary types of control structures are: conditional, iterative, and sequential.

1. Conditional Control:

Conditional control structures evaluate a condition (or conditions) and then, based on whether that condition is true or false, decide the next course of action.

- **IF-THEN-ELSE**: This is the most common conditional structure. The program checks a condition; if the condition is true, it executes one block of code, if false, another block of code is executed.

IF condition THEN -- code to execute if condition is true ELSE -- code to execute if condition is false END IF;

- **CASE**: This is useful when there are multiple conditions to evaluate.

CASE variable WHEN value1 THEN -- code for condition1 WHEN value2 THEN -- code for condition2 ... ELSE -- code if none of the conditions are met END CASE;

2. Iterative Control (Loops):

Iterative control structures execute a block of code multiple times, as long as a specified condition is met.

- **FOR Loop**: Iterates a specific number of times.

FOR counter IN initial_value..final_value LOOP -- code to be executed END LOOP;

- **WHILE Loop**: Executes as long as a condition remains true.

WHILE condition LOOP -- code to be executed END LOOP;

- **LOOP-EXIT**: A general loop that uses the EXIT clause to break out of the loop based on a condition.

LOOP -- code to be executed EXIT WHEN condition; END LOOP;

3. Sequential Control:

Sequential control is the inherent nature of programs where instructions are executed in the order they are written, from top to bottom. There are no checks or conditions; the program simply runs the statements one after the other.

However, in PL/SQL, you can have constructs like **GOTO**, which can alter the normal sequential flow, but its use is generally discouraged due to the complications it can introduce, making code harder to read and maintain.

4.3 Exception handling: Predefined Exception, User defined Exception.

Exception handling is an integral part of programming, allowing developers to manage unexpected or undesirable events during program execution. In PL/SQL, exceptions can be broadly categorized into two types: predefined exceptions and user-defined exceptions.

1. Predefined Exceptions:

These are exceptions that are already defined by PL/SQL. They get raised automatically when a corresponding error occurs during program execution.
Some common predefined exceptions are:

- **NO_DATA_FOUND**: Raised when a SELECT statement returns no rows.
- **TOO_MANY_ROWS**: Raised when a SELECT statement returns more than one row, but only one is expected.
- **ZERO_DIVIDE**: Raised when an attempt is made to divide a number by zero.
- **INVALID_CURSOR**: Raised when operations are attempted on an invalid cursor, like closing a cursor that's not open.
- **DUP_VAL_ON_INDEX**: Raised when there's an attempt to insert or update a value that would result in duplicate values in a unique indexed column.

Example:

```
BEGIN -- Some operations ... EXCEPTION WHEN NO_DATA_FOUND THEN -- Handle the exception ... END;
```

2. User-Defined Exceptions:

Sometimes, the predefined exceptions are not sufficient to handle specific errors relevant to the business logic or specific use cases. In such scenarios, PL/SQL allows developers to define their own exceptions.
Steps to handle user-defined exceptions:

1. **Declare the Exception**: The exception must be declared in the declaration section.
2. **Raise the Exception**: In the execution or business logic section, the exception can be raised based on some condition using the **RAISE** keyword.
3. **Handle the Exception**: In the exception section, you define what action should be taken once the exception is raised.

Example:

DECLARE ex_custom_exception EXCEPTION; BEGIN -- Some operations IF some_condition THEN RAISE ex_custom_exception; END IF; ... EXCEPTION WHEN ex_custom_exception THEN -- Handle the exception ... END;

In addition to this, PL/SQL provides the **RAISE_APPLICATION_ERROR** procedure, which allows developers to issue user-friendly error messages from the database layer.

4.4 Cursors: Implicit and Explicit Cursors, Declaring, opening ai1d Closing a Cursor, Fetching a Record from Cursor, Cursor for loops, Parameterized Cursors.

In PL/SQL, a cursor is a control structure that allows you to traverse and fetch records from a result set, one record at a time. Cursors come in handy when working with multi-row SQL statements, providing the means to process each row sequentially.

1. Implicit Cursors:

Implicit cursors are automatically created and managed by PL/SQL for DML statements (like INSERT, UPDATE, and DELETE) and singular SELECT statements that fetch only one row. You don't have to declare or control these cursors, as PL/SQL takes care of them behind the scenes.

Some attributes related to implicit cursors:

- **%FOUND**: Returns **TRUE** if the DML affected at least one record, otherwise **FALSE**.
- **%NOTFOUND**: Opposite of **%FOUND**.
- **%ROWCOUNT**: Returns the number of rows affected by the DML.

2. Explicit Cursors:

Explicit cursors offer more control compared to implicit cursors. They must be declared, opened, fetched from, and eventually closed by the developer.

Basic steps for using an explicit cursor:

1. **Declare the Cursor**: Define the cursor with the SQL statement it will handle.
2. **Open the Cursor**: Open it to initialize and allocate memory.
3. **Fetch from the Cursor**: Retrieve rows one at a time.
4. **Close the Cursor**: Release allocated memory.

DECLARE CURSOR cur_sample IS SELECT column_name FROM table_name; variable_name table_name.column_name%TYPE; BEGIN OPEN cur_sample; LOOP FETCH cur_sample INTO variable_name; EXIT WHEN cur_sample%NOTFOUND; -- Process fetched row ... END LOOP; CLOSE cur_sample; END;

- **Cursor For Loops**: Simplify the process by handling the opening, fetching, and closing automatically.

DECLARE CURSOR cur_sample IS SELECT column_name FROM table_name; BEGIN FOR record_name IN cur_sample LOOP -- Process fetched row ... END LOOP; END;

3. Parameterized Cursors:

Parameterized cursors accept parameters, allowing the cursor's result set to be determined at runtime. It provides flexibility in determining the data fetched by the cursor.

DECLARE CURSOR cur_sample(p_parameter TYPE) IS SELECT column_name FROM table_name WHERE some_column = p_parameter; BEGIN FOR record_name IN cur_sample(value) LOOP -- Process fetched row ... END LOOP; END;

4.5 Procedures: Advantages, Creating, Executing and Deleting a Stored Procedure.

Stored procedures in PL/SQL are named PL/SQL blocks that can be created and stored in the database for repetitive usage. They can accept parameters and are invoked using a CALL statement or invoked directly from a PL/SQL block.

Advantages of Stored Procedures:

1. **Modularity**: Procedures allow for modular programming. You can write the logic once and call it wherever required.
2. **Performance**: Since they are precompiled, repeated execution is faster. The database needs to interpret the procedure just once, and then it can execute it multiple times without reinterpreting it.
3. **Maintainability**: Changes can be made in a single place (the procedure) rather than in all the applications or scripts that use the functionality.
4. **Security**: Direct access to tables can be restricted, allowing users to access data only through procedures, thus providing a controlled environment.
5. **Reusability**: Procedures allow developers to encapsulate logic that can be reused in multiple applications or scripts.

Creating a Stored Procedure:

CREATE OR REPLACE PROCEDURE procedure_name (parameter_list) IS -- Declarations BEGIN -- Procedure body ... END procedure_name;

Executing a Stored Procedure:

A stored procedure can be executed in various ways:

1. From an anonymous PL/SQL block:

BEGIN procedure_name(arguments); END;

2. Using the **EXECUTE** or **EXEC** command (commonly used in SQL*Plus):

EXEC procedure_name(arguments);

3. From another stored procedure or function.

Deleting a Stored Procedure:

To delete a stored procedure, the **DROP** command is used:

DROP PROCEDURE procedure_name;

4.6 Functions: Advantages. Creating, Executing and Deleting a Functi0n.

In PL/SQL, a function is a named PL/SQL block that can take parameters, perform an action, and return a value. Functions are similar to procedures but with a key distinction: they must return a value.

Advantages of Functions:

1. **Modularity**: Like procedures, functions promote modular programming. The logic is written once and can be invoked wherever required.
2. **Performance**: Functions are precompiled, which ensures faster execution on repeated calls.
3. **Maintainability**: If there's a need for a change, it can be made in the function, avoiding the necessity to modify every instance where the function is called.
4. **Reusability**: Encapsulate frequently used computations or operations into functions, making them reusable across multiple applications or scripts.
5. **Security**: Access to data can be limited by allowing users to interact with data only via functions.
6. **Consistency**: By using functions, you ensure that the same logic is applied every time the operation is performed.

Creating a Function:

CREATE OR REPLACE FUNCTION function_name (parameter_list) RETURN return_type IS -- Declarations BEGIN -- Function logic ... RETURN value; END function_name;

Executing a Function:

A function can be executed in various ways:

1. As part of an SQL statement:

SELECT function_name(arguments) FROM dual;

2. From an anonymous PL/SQL block:

DECLARE variable_name return_type; BEGIN variable_name := function_name(arguments); END;

3. From another stored procedure or function.

Deleting a Function:

To delete a function, use the **DROP** command:

DROP FUNCTION function_name;

4.7 Database Triggers: Use of Database Triggers, how to apply database Triggers, Types of Triggers, Syntax for Creating Trigger, Deleting Trigger.

Database triggers are specialized stored procedures that automatically execute or fire when certain events occur within a database. These events might include changes in data via INSERT, UPDATE, or DELETE statements or even changes in schema or server operations.

Use of Database Triggers:

1. **Data Validation**: Implementing complex business rules or checks that can't be defined using standard constraint mechanisms.
2. **Auditing**: Automatically logging changes or operations on data for security and tracking purposes.
3. **Automated Response**: Taking automatic actions in response to certain changes in data, such as sending notifications.
4. **Cascade Operations**: Implementing cascading deletes or updates which aren't directly supported by referential integrity constraints.
5. **Maintaining Derived Data**: For instance, maintaining summary or aggregate data in response to individual row changes.

How to Apply Database Triggers:

Triggers can be associated with tables or views and will fire in response to the specified events on those objects.

Types of Triggers:

1. **Before vs. After Triggers**: Define whether the trigger fires before or after the triggering event.
2. **Row-level vs. Statement-level Triggers**: Row-level triggers fire once for each row affected by the triggering statement, whereas statement-level triggers fire once for the entire triggering statement.

3. **Instead of Triggers**: Used primarily with views to make a view updatable.
4. **DML Triggers**: Triggered by data modification events - INSERT, UPDATE, DELETE.
5. **DDL Triggers**: Triggered by schema-level events like CREATE, ALTER, DROP.
6. **Database Event Triggers**: Triggered by database-level events like startup or shutdown.

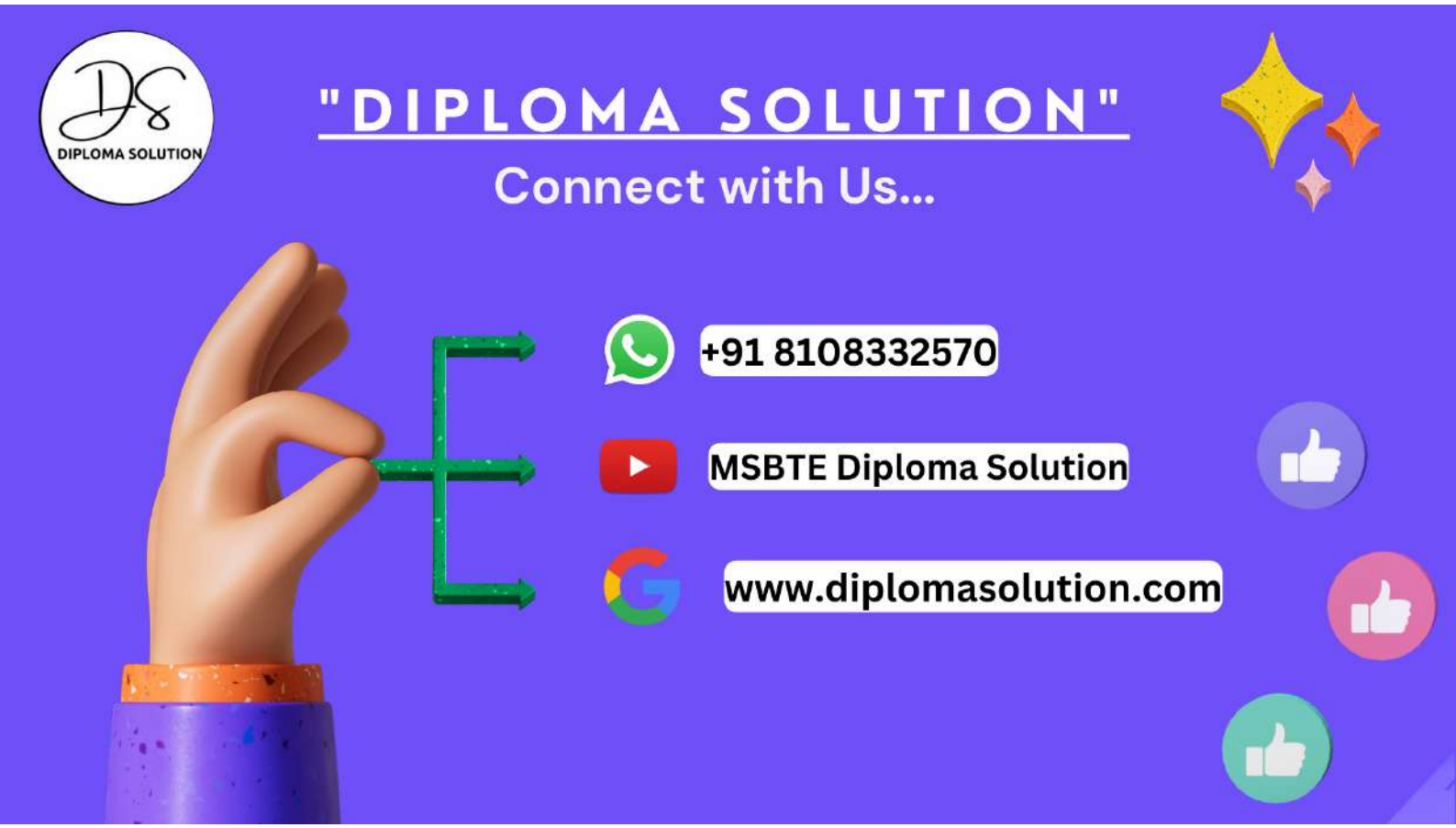
Syntax for Creating a Trigger:

```
CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE | AFTER | INSTEAD OF] [event]
[ON table_name] [FOR EACH ROW] [WHEN (condition)] DECLARE -- Variable declarations
BEGIN -- Trigger logic END; /
```

Deleting a Trigger:

To delete a trigger, use the **DROP** command:

```
DROP TRIGGER trigger_name;
```


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com

Unit V- Database Security and Transaction Processing

5.1 Database security: Introduction to database security, Data security Requirements, Types of Database Users-Creating, altering and Deleting Users.

In an era where data breaches can lead to significant financial and reputational damage, ensuring the security of databases has become paramount. Database security refers to the multiple layers of protection measures taken to guard databases against threats, unauthorized access, and malicious activities.

Introduction to Database Security:

Database security isn't just about protecting the data from external threats but also about ensuring data integrity, availability, and confidentiality. With the increasing volume of data stored and the value associated with this data, organizations are prime targets for cybercriminals. Thus, a robust security framework is essential.

Data Security Requirements:

To maintain the sanctity of data, certain security requirements are paramount:

1. **Authentication**: Ensure that only verified users can access the database.
2. **Authorization**: Grant specific permissions to users based on their roles to ensure they only access what they need.
3. **Confidentiality**: Protect sensitive data from unauthorized access.
4. **Integrity**: Ensure data remains accurate and consistent throughout its lifecycle.
5. **Availability**: Ensure that data remains accessible to authorized users whenever required.

Types of Database Users:

Database users can be broadly categorized into several types based on their roles and responsibilities:

1. **Database Administrators (DBAs)**: They manage and oversee the entire database system. DBAs have the highest level of access, allowing them to make structural changes, manage users, and perform backups and recovery.
2. **Application Developers**: They create applications that interact with the database. Typically, they need permissions to read from and write to specific parts of the database but don't need full administrative rights.
3. **End-users**: These are individuals who access the database using applications. Their rights are usually limited to querying data, with little to no permissions for altering the structure of the database.
4. **Data Analysts**: These users access the database to pull reports, analyze data, and generate insights. Their permissions often allow for querying and reading but restrict writing or altering data.
5. **Guest Users**: Often have the most limited permissions, only accessing certain public parts of the database.

Creating, Altering, and Deleting Users:
Managing user accounts and their privileges is an essential aspect of database security. Here's a general process:

1. **Creating Users**: Database administrators can create new users by specifying a username and password. Depending on the database system, there might be additional parameters like default tablespaces or profile settings.
2. **Altering Users**: Once created, user accounts might need modifications. Maybe their roles change, or they require additional access. DBAs can alter user accounts to change passwords, adjust privileges, or modify any other settings associated with the account.
3. **Deleting Users**: If a user no longer needs access to the database, perhaps due to a role change or leaving the organization, their account should be deleted or deactivated to ensure they can't access the database.

5.2 Protecting the data within database-Database Privileges: Systems privileges and object Privileges, Granting and Revoking Privileges: Grant and Revoke command.

Ensuring the security and integrity of data stored in databases is of paramount importance. Database privileges provide mechanisms to grant specific rights or permissions to users or roles, ensuring only authorized personnel can access or modify data.

System Privileges vs. Object Privileges:

1. **System Privileges**:
   - These are privileges that relate to particular actions on the database environment. They are not associated with a particular object but instead with broad operations a user or role can perform.
   - Examples: **CREATE TABLE** (allows user to create tables), **CREATE SESSION** (permits the user to connect to a database), **ALTER ANY TABLE** (allows altering of any table in the database).

2. **Object Privileges**:
   - These privileges are specific to particular database objects like tables, views, sequences, or procedures.
   - Examples: **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **EXECUTE**, **REFERENCES**.

Granting and Revoking Privileges:
To manage the distribution of these privileges, SQL provides **GRANT** and **REVOKE** commands.

1. **GRANT Command**:

The **GRANT** command is used to provide specific privileges to a user or a role.
Syntax:

GRANT privilege_name ON object_name TO {user_name | PUBLIC | role_name} [WITH GRANT OPTION];

- **WITH GRANT OPTION** allows the user to grant the privilege to other users.

Example:
GRANT SELECT, UPDATE ON employees TO user1, user2;

2. **REVOKE Command**:

If you need to withdraw privileges granted previously, you can use the **REVOKE** command.
Syntax:
REVOKE privilege_name ON object_name FROM {user_name | PUBLIC | role_name};

Example:
REVOKE DELETE ON employees FROM user1;

5.3 Transaction: Concept, Properties and States of Transaction.

A transaction is a sequence of one or more SQL operations executed as a single logical unit of work. It ensures data integrity and consistency. Transactions in databases are used to manage actions that can alter the data, such as creating, updating, or deleting records.

Properties of Transactions:
Transactions are governed by four key properties, often referred to by the acronym ACID:

1. **Atomicity**:
   - This ensures that all operations within a transaction are completed successfully; if not, none of the operations are applied. It's an all-or-nothing scenario.
   - If a transaction is interrupted (for example, due to a system failure), any changes made up to that point are rolled back to their previous state.
2. **Consistency**:
   - This ensures that each transaction brings the database from one consistent state to another.
   - Any data written to the database must be valid according to all defined rules, including constraints, cascades, and triggers.
3. **Isolation**:
   - This ensures that concurrent execution of transactions yields consistent results.

- Transactions are shielded from each other, meaning that operations within one transaction are invisible to other concurrent transactions until they are completed.

4. **Durability**:
   - This ensures that once a transaction has been committed, it remains so. This is true even in the event of power loss, crashes, or errors.
   - Completed transactions are recorded in persistent storage (like hard drives).

States of Transaction:
A transaction progresses through several states during its lifetime:

1. **Active**:
   - The initial state of every transaction.
   - At this point, the transaction is being executed.
2. **Partial Commit**:
   - A state where the initial part of a transaction has been executed but it is not yet finalized.
3. **Committed**:
   - The transaction has completed and changes are permanently saved to the database.
4. **Failed**:
   - The system has recognized that the transaction cannot continue executing due to some internal error or violation of integrity constraints.
5. **Aborted**:
   - The transaction has been rolled back and the database remains unchanged.
   - After aborting, the system can either start the transaction all over again or kill it.

5.4 Database Backup -Types of Failures, Causes of failures. Database Backup Introduction. Types of Database Backups-Physical and Logical.

Data is one of the most valuable assets of any organization. Ensuring its safety and integrity is paramount, especially in the face of potential failures. One of the fundamental steps in data protection is creating backups of the database.

Types of Failures:
Understanding potential failures is the first step towards crafting an effective backup strategy.

1. **Transaction Failures**:
   - Occurs due to logical errors, like division by zero, or integrity constraint violations.
2. **System Failures**:
   - Result from software bugs, power outages, or system crashes.

3. **Media Failures**:
    - Caused by physical damage to storage media, such as hard drive malfunctions or disk corruption.
4. **Network Failures**:
    - Arise from issues in network connections, hindering communication between database systems.
5. **Concurrency Control Failures**:
    - Occur when multiple users try to access the database simultaneously, leading to data inconsistency or deadlock situations.

Causes of Failures:
The root causes of database failures can be numerous:

- Hardware malfunctions or wear and tear.
- Software bugs or incompatibility issues.
- Human errors, such as accidental data deletion.
- Natural disasters, like fires or floods.
- Malicious attacks, including cyberattacks, hacking, or virus infections.

Database Backup Introduction:
Database backup refers to the process of creating a copy of the database, which can be restored in case of any unexpected failures or data loss incidents. Effective backup strategies allow for timely data recovery, minimizing downtime and data loss.

Types of Database Backups:
There are principally two types of backups:

1. **Physical Backups**:
    - Involves copying the actual physical files that constitute the database. This includes data files, control files, and archived redo logs.
    - Physical backups are typically taken when the database is offline or in a quiet state to ensure consistency.
    - Restoration is usually faster since it involves direct file replacement.

2. **Logical Backups**:
    - Extracts logical data (tables, procedures, etc.) from the database using tools like SQL's **EXPORT** or MySQL's **mysqldump**.
    - Results in a file containing SQL statements that, when executed, recreate the database.
    - Provides a way to backup a subset of the database, like specific tables.
    - Restoration involves executing the SQL statements, which can be slower than restoring physical backups.

5.5 Database Recovery - Recovery concept, Recovery Techniques - Roll forward, Rollback,

Database systems are prone to failures due to a myriad of reasons ranging from system crashes to human errors. To maintain data integrity and availability, it's imperative to have recovery mechanisms in place. Database recovery refers to the strategy and processes involved in restoring a database to a correct state following failures.

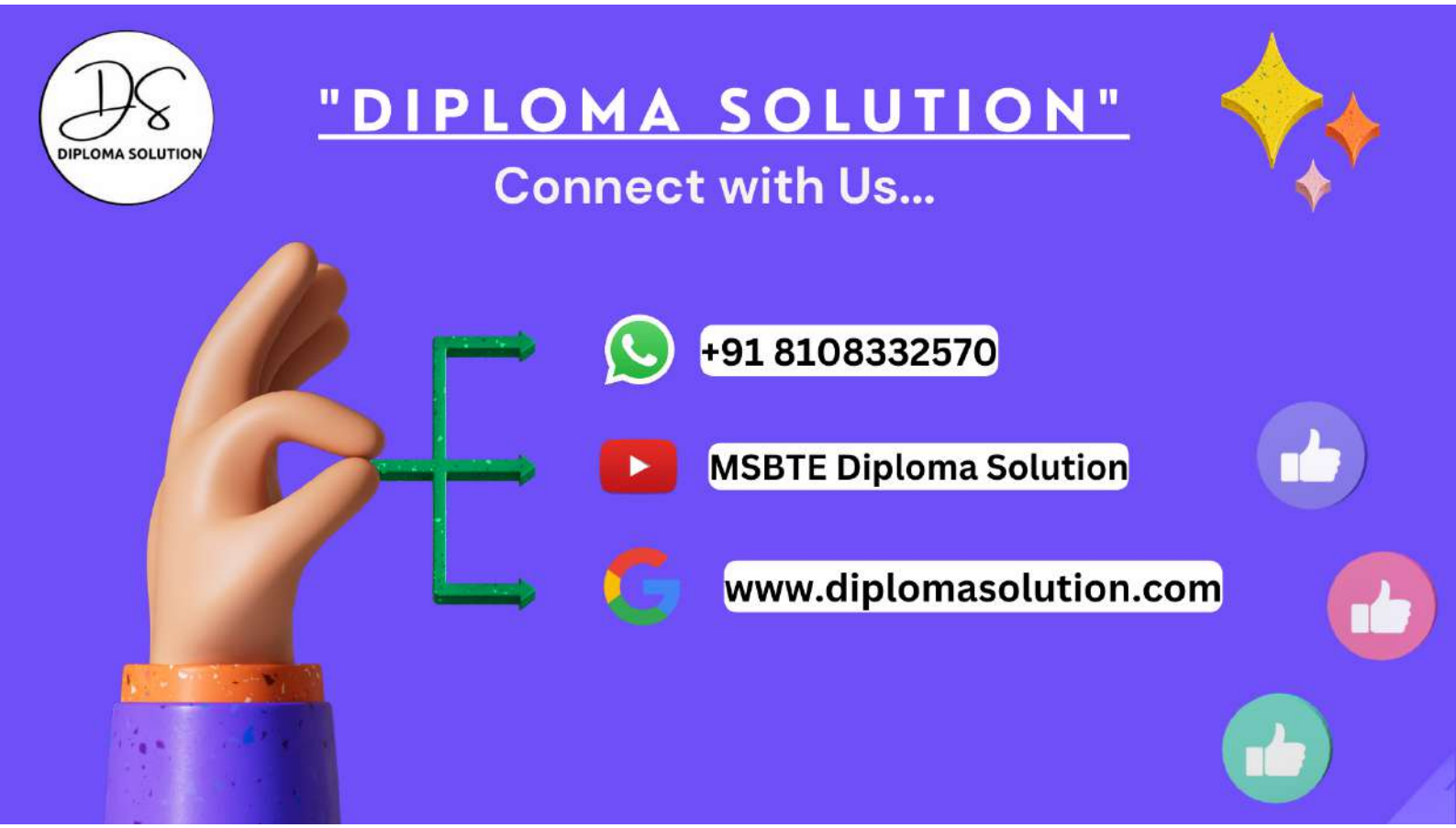
Recovery Concept:
Recovery in the context of databases involves:

- Restoring the database to a state that ensures data integrity, often to the most recent consistent state before the failure.
- Repairing the structural and logical integrity of the database without data loss.
- Minimizing downtime to ensure data availability and minimize disruption.

Recovery Techniques:
When it comes to actual recovery techniques, there are mainly two mechanisms: Roll forward and Rollback.

1. **Roll Forward**:
    - Also known as "Redo," this technique involves moving the database from a past state to a more recent, desired state.
    - After restoring a previous backup, the system applies logged changes (from the transaction logs) to move the database forward in time. It re-applies all the transactions from logs to the backup data.
    - The goal is to incorporate all valid transactions that occurred after the backup.
    - Roll forward is typically used when the physical database files are corrupted but transaction logs are intact.

2. **Rollback**:
    - Also known as "Undo," this technique involves reverting the database to a previous state by undoing certain transactions.
    - When a transaction is identified as problematic (for instance, it's causing an error or it hasn't been completed due to system crash), the system can use the logs to undo the changes made by that transaction.
    - It returns the database to a state before the transaction began.
    - The rollback mechanism is integral to maintaining the ACID properties of databases, particularly the atomicity property which ensures that transactions are either fully completed or fully undone.


DIPLOMA SOLUTION
"DIPLOMA SOLUTION"
Connect with Us...
+91 8108332570
MSBTE Diploma Solution
www.diplomasolution.com