DATABASE

- ## Introduction to Database

  A database is an organized collection of data stored in a computer system and usually controlled by a database management system (DBMS). Structured query language (SQL) is commonly used for data querying and writing.

- ## Why databases are needed?

  Organization: Databases help store data in a structured way, making it easy to find, update, or remove information.
  Speed: They allow for fast retrieval of data, even when dealing with large amounts of information.
  Security: Databases provide protection against unauthorized access .
  Accuracy: They ensure data consistency and integrity by preventing errors like duplicate entries or invalid information.
  Multiple Users: Databases let many users access and update data simultaneously without causing conflicts or errors.

- ## Where are the database maintained?

  Databases are typically maintained on **servers** or **data storage systems** that are designed to store, retrieve, and manage large amounts of structured data. These can be

physically located in various places depending on the type of database setup:

1. **On-Premises Databases**:

- These databases are hosted on physical servers located within an organization's own facilities or data centers.

2. **Cloud Databases**:

- Cloud databases are hosted by third-party providers like AWS, Azure, or Google Cloud. These databases are scalable, flexible, and managed remotely by the service provider.

3. **Distributed databases:**

- Distributed databases spread data across multiple servers or locations. They help ensure high availability and fault tolerance.

4. **Hybrid Databases**:

- Some organizations use a combination of on-premises, cloud, and edge databases to meet different needs. For instance, sensitive data might be kept in on-premises databases, while less-sensitive data could be stored in the cloud.

- **Database Management System**

A Database Management System (DBMS) is a software system that is designed to manage and organize data in a structured manner. It allows users to create, modify, and query a database, as well as manage the security and access controls for that database. DBMS provides an environment to store and retrieve data in convenient and efficient manner.

Types:

1.**Relational DBMS:**

A Relational Database (RDBMS) stores data in a structured way using tables Each table contains rows (records) and columns (attributes). These databases are built on the concept of relations between tables, where data can be linked or related to other data in different tables. The language used to interact with relational databases is typically SQL (Structured Query Language).

Examples: MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server

2. **NoSQL Database:**

A category of databases that store data in formats other than tables. These databases are designed for scalability and flexibility, handling unstructured or semi-structured data. It is ideal for handling large amounts of unstructured or semi-structured data.

Key-Value:

A Key-Value Database is a type of NoSQL database that stores data as pairs of a key and a value. In a Key-Value Database Each key is unique. Each key points to a value, which could be any type of data, like a string, a number, or even a list of items.
key-value databases are used in situations where you need to store and retrieve data very quickly.
Example:
Redis:
A very fast, open-source key-value store, often used for caching and real-time applications.

Column based:

A column-based database is a type of database where data is stored in columns rather than in rows. So instead of storing all of the information about a single person together in one place as in row-based, the database stores all of the values of each attribute like Name, Age, Country in separate columns.
For the same data, a columnar database would store it like this:
Column 1 (ID): [1, 2, 3]
Column 2 (Name): ["John", "Alice", "Bob"]
Column 3 (Age): [30, 25, 35]
Column 4 (Country): ["USA", "Canada", "UK"]

Example: Imagine you're working with a huge dataset of millions of users, and you want to calculate the average age of users. If you have the data in a column-based database, it's easier to

access just the Age column and calculate the average quickly, without needing to scan through all the rows.

Document Based

A document-based database also known as document store is a type of NoSQL database designed to store, retrieve, and manage document-oriented information. Instead of storing data in rows and columns like traditional relational databases, document databases store data as documents, which are typically in formats like JSON, BSON, or XML.

Graph Based

Graph database is designed to handle data whose relationships are as important as the data itself. A graph database represents data as nodes and edges, where:
Nodes represent entities or objects (e.g., users, products, locations). Edges represent the relationships between these nodes (e.g., "friend of," "purchased," "located at").

SQL

SQL is the standard language used to interact with relational databases. It allows you to create, read, update, and delete data often referred to as CRUD operations.

1. DDL (Data Definition Language)

DDL is used to define and manage the structure of a database, such as creating, altering, and deleting tables, indexes, and other database objects.

Key DDL commands include:

CREATE: Used to create new database objects, like tables, indexes, or schemas.

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

ALTER: Used to modify the structure of an existing database object, such as adding, deleting, or modifying columns in a table.

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

DROP: Used to delete database objects, such as a table or index, permanently from the database.

```
DROP Table Employees;
```

TRUNCATE: Command is used to delete all rows from a table, but it does not remove the table structure.

```
TRUNCATE TABLE Employees;
```

2.DML (Data Manipulation Language)

DML is used to manipulate the data within the tables in a database. These commands allow users to insert, update, delete, and retrieve data.

INSERT: Used to add new records (rows) to a table.

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Department)
VALUES (1, 'John', 'Doe', 'Sales');
```

UPDATE: Used to modify existing records in a table.

```
UPDATE Employees
SET Department = 'Marketing'
WHERE EmployeeID = 1;
```

DELETE: Used to delete records from a table without affecting the table structure.

```
DELETE FROM Employees
WHERE EmployeeID = 1;
```

SELECT: Used to retrieve data from one or more tables based on specific conditions.

```
SELECT * FROM Employees;
```

3.DCL

DCL deals with permissions and access control to the database. The commands are used to grant or revoke privileges like who can view, insert, update, or delete data.

The main DCL commands are:

a) **GRANT – Give Permissions**

The GRANT command is used to give specific permissions to users or roles on database. It allows a user to perform certain actions like selecting data, inserting data, etc.

Example: Giving a user named Arya permission to select and insert data into the employees table:

```
GRANT SELECT, INSERT ON employees TO Arya;
```

b) **REVOKE – Remove Permissions**

The REVOKE command is used to remove previously granted permissions from a user or role.

Example: Removing SELECT and INSERT permissions from Arya on the employees table:

REVOKE SELECT, INSERT ON employees FROM Arya;

4.TCL

TCL commands help you manage transactions in a database. A transaction in a database is a sequence of one or more operations that are executed as a single unit, ensuring consistency, isolation, and durability (ACID properties).

1. COMMIT

The COMMIT statement is used to save all changes made during the current transaction. Once a COMMIT is executed, all changes are permanently applied to the database.

```
BEGIN TRANSACTION;
INSERT INTO employees (id, name) VALUES (1, 'Alice');
INSERT INTO employees (id, name) VALUES (2, 'Bob');
COMMIT;       -- Save the changes permanently.
```

2. ROLLBACK

The ROLLBACK statement is used to undo all changes made during the current transaction. If a ROLLBACK is issued, the database is reverted to its state before the transaction started.

```
BEGIN TRANSACTION;
INSERT INTO employees (id, name) VALUES (1, 'Alice');
```

INSERT INTO employees (id, name) VALUES (2, 'Bob');
ROLLBACK; - - Undo the changes.

What is a Schema?

A schema in a database is like a blueprint that defines how data is organized and structured inside the database.
It shows:
What data is stored (e.g., employees, orders, products).
How that data is organized (e.g., tables, fields).
How the data is related (e.g., how employees are linked to departments).

Tables:
A schema defines tables, which store your data. Each table has rows and columns .
Example: A table called Employees might have columns like EmployeeID, Name, Age, and Department.
Columns:
Columns in a table are the attributes or characteristics of the data. Each column has a specific data type like text, numbers, or dates.
Example: The EmployeeID column in the Employees table would contain unique numbers for each employee.

Relationships:

A schema also defines relationships between tables. For example, an Employees table might be connected to a Departments table to show which employee belongs to which department.
Example: An Employee might have a DepartmentID that links them to a department in the Departments table.

Constraints:

Constraints are rules that help ensure the data is accurate and consistent.
Example: A primary key ensures that each EmployeeID is unique. A foreign key ensures that every DepartmentID in the Employees table matches a valid department in the Departments table.

What are Views?

A view is a virtual table in a database that provides a way to present data in a specific format, without storing any data. Views do not store data and instead fetch it from the underlying tables whenever you query them.

1. Employees Table:

EmployeeID	Name	Age	DepartmentID	Salary
1	Alice	30	101	50000
2	Bob	25	102	40000
3	Charlie	35	101	55000

2. Departments Table:

DepartmentID	DepartmentName
101	HR
102	IT

```
CREATE VIEW EmployeeInfo AS
SELECT Employees.Name, Employees.Age, Employees.Salary,
Departments.DepartmentName
FROM Employees
JOIN Departments ON Employees.DepartmentID
Departments.DepartmentID;
```

This creates a view named EmployeeInfo. It pulls data from the Employees and Departments tables, and it shows the employee's name, age, salary, and their department.

What are Triggers?

Triggers are special rules that automatically run when certain actions like INSERT, UPDATE, DELETE happen in the database.
Example- To create a trigger that automatically logs whenever an INSERT operation is performed on the Employees table.

```
CREATE TRIGGER LogEmployeeInsert
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
  INSERT INTO Employee_Audit (Action, EmployeeID, Name)
  VALUES ('INSERT', NEW.EmployeeID, NEW.Name);
```

What are Indexes?

- Indexes in a database are special data structures that improve the speed of data retrieval operations on a table. It improves SELECT queries but might slow down INSERT, UPDATE, and DELETE operations because the index must be updated.
- An index in a relational database function like a **key-value store**. Each value in the index corresponds to a **key**, and that key points to the **location** of the actual data in the table.

  CREATE INDEX idx_employee_id ON employees (employee_id);

  Visual Representation of the Index:

Employee ID (Key)	Pointer to Row (Value)
101	Row 1
102	Row 2
103	Row 3

What are Stored procedures?

A **stored procedure** is a set of SQL queries saved and stored in the database. Once a stored procedure is defined, it can be reused multiple times without the need to rewrite the SQL commands.

For example:

Consider the task of adding a new employee record to a database. Instead of writing the INSERT statement each time a new employee needs to be added, a stored procedure can be created to handle the insertion. After defining the stored procedure, it can be called with the required employee details, thereby simplifying the process and promoting code reuse.

```
CREATE PROCEDURE AddEmployee
   @FirstName VARCHAR (50),
   @LastName VARCHAR (50),
   @Position VARCHAR (50)
AS
BEGIN
   INSERT INTO Employees (FirstName, LastName, Position)
   VALUES (@FirstName, @LastName, @Position);
END;
-- First Employee: John Doe as Manager
EXEC AddEmployee @FirstName = 'John', @LastName = 'Doe',
@Position = 'Manager';


-- Second Employee: Alice Smith as Developer
```

EXEC AddEmployee @FirstName = 'Alice', @LastName = 'Smith', @Position = 'Developer';

NULL

NULL is a special marker used in SQL to indicate that a value is unknown or missing. It is different from an empty string, 0, or any other default value. NULL represents absence of data, not a zero or an empty value.

Example:

SELECT * FROM Employees WHERE Salary IS NULL;

This query will return all employees whose salary is missing (NULL).

JOINTS

A **JOIN** combines rows from two or more tables based on a related column between them. There are different types of joins in SQL:

Types of Joins:

- **INNER JOIN**: Returns only the rows where there is a match in both tables.
  Employees Table:

Employee ID	Name
1	John
2	Alice
3	Bob

Departments table:

Department ID	Employee ID	Department Name
101	1	HR
102	2	IT
103	4	Sales

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
INNER JOIN Departments
ON Employees.EmployeeID = Departments.EmployeeID;
```

Result:

Name	Department
John	HR
Alice	IT

- **LEFT JOIN (or LEFT OUTER JOIN)**: Returns all rows from the left table, and matching rows from the right table. If no match, NULL is returned for columns from the right table.

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments
```

ON Employees.EmployeeID = Departments.EmployeeID;

Name	Department
John	HR
Alice	IT
Bob	NULL

- **RIGHT JOIN (or RIGHT OUTER JOIN)**: Returns all rows from the right table, and matching rows from the left table. If no match, NULL is returned for columns from the left table.

  SELECT Employees.Name, Departments.DepartmentName
  FROM Employees
  RIGHT JOIN Departments
  ON Employees.EmployeeID = Departments.EmployeeID;

Name	Department
John	HR
Alice	IT
NULL	Sales

- **FULL JOIN (or FULL OUTER JOIN)**: Returns rows when there is a match in either the left or the right table.

  SELECT Employees.Name, Departments.DepartmentName
  FROM Employees
  FULL JOIN Departments
  ON Employees.EmployeeID = Departments.EmployeeID;

Name	Department
John	HR
Alice	IT
Bob	NULL
NULL	Sales

AGGREGATIONS

Aggregation functions allow you to perform calculations on a set of values. Common aggregation functions include:

- COUNT (): Counts the number of rows.
- SUM (): Adds up the values of a column.
- AVG (): Calculates the average value of a column.
- MAX (): Returns the maximum value in a column.
- MIN (): Returns the minimum value in a column.

```
SELECT DepartmentID, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY DepartmentID;
```

Aliases

An **alias** is a temporary name given to a table or column for easier reference in SQL queries.

- **Table alias**: Used to give a short name to a table.
- **Column alias**: Used to rename columns in the result set.

Example:

```
SELECT E.FirstName, E.LastName, D.DepartmentName
FROM Employees AS E
INNER JOIN Departments AS D
ON E.DepartmentID = D.DepartmentID;
```

GROUP BY

The **GROUP BY** clause is used to group rows that have the same values in specified columns into summary rows. Often used with aggregation functions like COUNT (), SUM (), AVG (), etc.

Example:

```
SELECT DepartmentID, COUNT (*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID;
```

This query returns the number of employees in each department by grouping the rows based on DepartmentID.

HAVING

The **HAVING** clause is used to filter records after grouping them with GROUP BY. It is like the WHERE clause, but WHERE filters rows before grouping, while HAVING filters after grouping.

Example:

SELECT DepartmentID, AVG(Salary) AS AverageSalary

FROM Employees

GROUP BY DepartmentID

HAVING AVG(Salary) > 50000;

This query returns only departments where the average salary is greater than 50,000.

Complex Queries

Complex queries often combine multiple SQL features, like joins, aggregations, and filtering. They might involve subqueries, multiple joins, or nested queries.

Example:

SELECT E.FirstName, E.LastName, D.DepartmentName, AVG(E.Salary) AS AverageSalary

FROM Employees AS E

```
INNER JOIN Departments AS D ON E.DepartmentID =
D.DepartmentID

WHERE E.HireDate > '2020-01-01'

GROUP BY D.DepartmentName, E.FirstName, E.LastName

HAVING AVG(E.Salary) > 60000;
```

- Joins Employees and Departments.
- Filters employees hired after January 1st, 2020.
- Groups by employee names and department names.
- Returns employees who have an average salary greater than 60,000.

QUERY TUNING

Query tuning is the process of improving the performance of a query. Here are some common techniques to optimize your queries:

Techniques for Query Tuning:

1. **Indexes**: Create indexes on columns that are frequently used in WHERE, JOIN, or ORDER BY clauses. This speeds up data retrieval.

2. **Avoiding SELECT ***: Select only the columns you need rather than using SELECT *, which retrieves all columns and may be inefficient.
3. **Using Proper Joins**: Be mindful of the type of joins you use. For example, if you need only matching rows, use INNER JOIN. Avoid using LEFT JOIN unless necessary.
4. **Limit the Data**: Use LIMIT or TOP to restrict the number of rows returned, especially for large datasets.
5. **Avoid Complex Subqueries**: Subqueries in SELECT, WHERE, or FROM clauses can be inefficient. Try to simplify or rewrite them as joins.