**1. What is SQL, and what are its main components?**

**SQL (Structured Query Language)** is a standard programming language used to manage and manipulate relational databases. It allows users to create, retrieve, update, and delete data in databases.

**Main components of SQL:**

1. **DDL (Data Definition Language):** Defines and modifies database structure.
2. **DML (Data Manipulation Language):** Manipulates data within tables.
3. **DQL (Data Query Language):** Retrieves data from the database (primarily the SELECT statement).
4. **DCL (Data Control Language):** Controls access to the database.
5. **TCL (Transaction Control Language):** Manages database transactions.

**2. Explain the difference between DDL, DML, and DCL.**

| **Aspect** | **DDL** | **DML** | **DCL** |
| --- | --- | --- | --- |
| **Full Form** | Data Definition Language | Data Manipulation Language | Data Control Language |
| **Purpose** | Defines and modifies database schema. | Manipulates data in tables. | Controls user access to data. |
| **Examples** | CREATE, ALTER, DROP | INSERT, UPDATE, DELETE | GRANT, REVOKE |
| **Effect** | Changes structure (schema). | Alters data in existing tables. | Manages permissions. |
| **Commit Required** | Auto-committed (immediate effect). | Explicit commit needed. | Auto-committed. |

**4. How is a primary key different from a unique key?**

| **Aspect** | **Primary Key** | **Unique Key** |
| --- | --- | --- |
| **Uniqueness** | Ensures all values are unique. | Ensures all values are unique. |
| **NULL Values** | Does not allow NULL values. | Allows a single NULL value. |
| **Purpose** | Identifies each record uniquely. | Enforces data uniqueness but not for identification. |
| **Count per Table** | Only one primary key per table. | Multiple unique keys allowed. |

**5. What are constraints in SQL? Name some commonly used constraints.**

**Constraints** are rules enforced on table columns to ensure the integrity, accuracy, and reliability of the data.

**Commonly Used Constraints:**

1. **NOT NULL:** Ensures that a column cannot have NULL values.
2. **UNIQUE:** Ensures that all values in a column are unique.
3. **PRIMARY KEY:** Combines NOT NULL and UNIQUE, identifying each record uniquely.
4. **FOREIGN KEY:** Enforces referential integrity by linking a column to the primary key of another table.
5. **CHECK:** Ensures that all values in a column satisfy a specific condition.
6. **DEFAULT:** Sets a default value for a column when no value is provided.

**Example:**

**CREATE TABLE employees (**

**id INT PRIMARY KEY,**

**name VARCHAR(50) NOT NULL,**

**age INT CHECK (age >= 18),**

**email VARCHAR(100) UNIQUE**

**);**

**2. How does the GROUP BY clause work?**

The GROUP BY clause groups rows that have the same values in specified columns into summary rows, typically used with aggregate functions like COUNT, SUM, AVG, etc.

**Example:**

**SELECT department, COUNT(\*) AS total\_employees**

**FROM employees**

**GROUP BY department;**

**Result:**

| **Department** | **Total Employees** |
| --- | --- |
| Sales | 10 |
| HR | 5 |

**4. What is the use of the DISTINCT keyword in SQL?**

The DISTINCT keyword is used to return unique values from a column, eliminating duplicate rows.

**Example:**

**SELECT DISTINCT department**

**FROM employees;**

**1. What are Joins in SQL? Name the Types of Joins.**

**Joins in SQL are used to combine rows from two or more tables based on a related column. They are essential for querying data stored across multiple tables.**

**Types of Joins:**

1. **INNER JOIN: Returns only the matching rows from both tables.**
2. **LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and matching rows from the right table. Non-matching rows in the right table will show NULL.**
3. **RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and matching rows from the left table. Non-matching rows in the left table will show NULL.**
4. **FULL OUTER JOIN: Returns all rows from both tables, with NULL in non-matching columns.**
5. **CROSS JOIN: Returns the Cartesian product of both tables.**
6. **SELF JOIN: Joins a table to itself.**

**-- INNER JOIN Example**

**SELECT students.name, courses.course\_name**

**FROM students**

**INNER JOIN courses ON students.course\_id = courses.id;**

**-- LEFT JOIN Example**

**SELECT students.name, courses.course\_name**

**FROM students**

**LEFT JOIN courses ON students.course\_id = courses.id;**

**5. Explain SELF JOIN with an Example.**

**A SELF JOIN is a join in which a table is joined to itself. This is typically used when a table contains hierarchical data, such as an employee table where employees have managers.**

**Example:**

**SELECT e1.name AS employee, e2.name AS manager**

**FROM employees e1**

**LEFT JOIN employees e2 ON e1.manager\_id = e2.id;**

**1. What is an index in SQL, and why is it used?**

An **index** in SQL is a database structure that improves the speed of data retrieval operations on a table by creating a data structure that holds references to the original table rows. It works like a "table of contents" for the database, making searches faster.

-- Creating a clustered index

CREATE CLUSTERED INDEX idx\_clustered ON employees (id);

-- Creating a non-clustered index

CREATE NONCLUSTERED INDEX idx\_nonclustered ON employees (last\_name);

DROP INDEX idx\_employee\_name ON employees;

**1. What is a transaction in SQL?**

A **transaction** is a sequence of operations performed as a single logical unit of work. It ensures that the operations either complete entirely or not at all, maintaining database integrity.

 **Atomicity:**

* Ensures that all operations within a transaction are completed; if one fails, the entire transaction is rolled back.

 **Consistency:**

* Guarantees that a transaction takes the database from one valid state to another, preserving integrity.

 **Isolation:**

* Ensures that the intermediate state of a transaction is invisible to other transactions.

 **Durability:**

* Ensures that once a transaction is committed, the changes are permanent, even in case of a system failure.

**5. What is the difference between implicit and explicit transactions?**

| **Aspect** | **Implicit Transaction** | **Explicit Transaction** |
| --- | --- | --- |
| **Definition** | Automatically begins and commits after each SQL statement. | Requires manual control using BEGIN TRANSACTION, COMMIT, and ROLLBACK. |
| **Control** | No control over transaction boundaries. | Full control over when a transaction starts and ends. |
| **Example** | INSERT INTO table ... (implicitly commits). | BEGIN TRANSACTION; ... COMMIT; |

**1. What are stored procedures in SQL, and why are they used?**

A **stored procedure** is a precompiled collection of SQL statements and optional control-of-flow logic stored in the database. They are executed as a single unit and can accept input and output parameters.

**Why are they used?**

* **Performance:** Precompiled, reducing query parsing time.
* **Code Reusability:** Centralized logic for operations that can be reused across applications.
* **Security:** Can restrict direct access to tables, exposing only stored procedures.
* **Maintainability:** Easier to update and manage centralized logic.

CREATE PROCEDURE TransferFunds

@SourceAccountID INT,

@TargetAccountID INT,

@Amount DECIMAL(10, 2)

AS

BEGIN

BEGIN TRY

BEGIN TRANSACTION;

-- Deduct from source account

UPDATE Accounts

SET Balance = Balance - @Amount

WHERE ID = @SourceAccountID;

-- Add to target account

UPDATE Accounts

SET Balance = Balance + @Amount

WHERE ID = @TargetAccountID;

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION;

-- Log or return error details

PRINT ERROR\_MESSAGE();

THROW; -- Re-throw the error

END CATCH;

END;

-- Call the procedure

EXEC TransferFunds @SourceAccountID = 1, @TargetAccountID = 2, @Amount = 100;

**1. What is a view in SQL, and how is it created?**

A **view** is a virtual table in SQL that is based on the result set of a SELECT query. It does not store data itself but provides a way to simplify complex queries or provide a specific representation of data.

* Create View

CREATE VIEW HR\_Employees AS

SELECT id, name, salary

FROM employees

WHERE department = 'HR';

* Use View

SELECT \* FROM HR\_Employees;

| **Feature** | **CHAR** | **VARCHAR** |
| --- | --- | --- |
| **Storage Type** | Fixed length | Variable length |
| **Length** | Always stores the specified number of characters, padding with spaces if necessary. | Stores only the actual data, without padding. |
| **Efficiency** | Less efficient for variable-length data since it always uses the defined space. | More efficient for variable-length data because it only uses space for the actual data. |
| **Use Case** | Best for fixed-length strings (e.g., country codes, postal codes, etc.). | Best for strings with varying lengths (e.g., names, email addresses, etc.). |
| **Space Usage** | If you define CHAR(10) and store a 5-character string, it will pad with 5 spaces, using 10 bytes. | If you define VARCHAR(10) and store a 5-character string, it will use only 5 bytes (plus 1 or 2 bytes for length storage). |
| **Example** | CHAR(5) will always store 5 characters, even if the string is shorter. | VARCHAR(5) will store a string of length up to 5 characters, using only the needed space. |

**How to Optimize an SQL Query for Better Performance**

**Optimizing SQL queries is essential for improving performance, especially when working with large databases. Here are some best practices to optimize SQL queries:**

1. **Use Proper Indexing:**
   * **Ensure that the columns used in JOIN, WHERE, and ORDER BY clauses are indexed. Indexes can dramatically speed up data retrieval.**
2. **Avoid SELECT \*:**
   * **Instead of selecting all columns with SELECT \*, choose only the columns you need. This reduces the amount of data retrieved and improves performance.**

**SELECT name, age FROM employees; -- Instead of SELECT \***

1. **Limit Rows Returned:**
   * **Use LIMIT or TOP to reduce the number of rows returned if only a subset is required**

**SELECT \* FROM employees LIMIT 10;**

1. **Use Proper Joins:**
   * **Choose the correct type of JOIN (e.g., INNER JOIN, LEFT JOIN) based on your needs, and make sure you're joining on indexed columns.**
2. **Avoid N+1 Query Problem:**
   * **Instead of issuing multiple queries in a loop, try to combine them into a single query using JOIN or IN to reduce the number of queries sent to the database.**
3. **Use EXISTS instead of IN (when applicable):**
   * **EXISTS can be more efficient than IN when dealing with subqueries, especially when the subquery returns a large number of results.**
4. **Avoid Functions on Indexed Columns in WHERE Clause:**
   * **Avoid using functions on columns in the WHERE clause (e.g., UPPER(name)), as it may prevent the database from using the index effectively.**
5. **Optimize Subqueries:**
   * **Rewrite subqueries to avoid nesting if possible. Use JOIN or EXISTS instead.**
6. **Ensure Proper Data Types:**
   * **Ensure that the data types used for columns and values are appropriate to avoid unnecessary type casting during comparisons.**
7. **Use Proper Aggregations:**

* **Use aggregations like SUM, AVG, COUNT carefully, and ensure they are done on indexed columns if necessary.**

1. **To delete duplicate rows while keeping one, use a common table expression (CTE) or a subquery with ROW\_NUMBER().**

**DELETE FROM employees**

**WHERE id NOT IN (**

**SELECT MIN(id)**

**FROM employees**

**GROUP BY name**

**);**

| **Feature** | **DELETE** | **TRUNCATE** |
| --- | --- | --- |
| **Purpose** | **Deletes specific rows based on a WHERE condition.** | **Removes all rows from a table.** |
| **Condition** | **Can use a WHERE clause to filter rows.** | **Cannot use a WHERE clause.** |
| **Performance** | **Slower as it logs each row deletion.** | **Faster as it does not log individual row deletions.** |
| **Triggers** | **Fires DELETE triggers, if any.** | **Does not fire triggers.** |
| **Rollback** | **Can be rolled back if inside a transaction.** | **Cannot be rolled back in most databases.** |
| **Resets Identity Column** | **No, retains the identity value.** | **Yes, resets identity column.** |

**How Do You Implement Cascade Delete in SQL?**

**Cascade delete automatically deletes all related records in child tables when a record in the parent table is deleted. It is implemented using the ON DELETE CASCADE option in the foreign key constraint.**

**Use Case:**

* **If a Customer is deleted, all their Orders should also be deleted.**

**-- Create Customer table**

**CREATE TABLE Customer (**

**CustomerID INT PRIMARY KEY,**

**Name VARCHAR(50)**

**);**

**-- Create Orders table with ON DELETE CASCADE**

**CREATE TABLE Orders (**

**OrderID INT PRIMARY KEY,**

**CustomerID INT,**

**OrderDate DATE,**

**FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON DELETE CASCADE**

**);**

**-- Insert records**

**INSERT INTO Customer (CustomerID, Name) VALUES (1, 'Alice');**

**INSERT INTO Orders (OrderID, CustomerID, OrderDate) VALUES (101, 1, '2025-01-01');**

**-- Delete Customer**

**DELETE FROM Customer WHERE CustomerID = 1;**

**-- The corresponding order with OrderID = 101 will also be deleted.**

**First Normal Form (1NF):**

* **Ensures that the table has a primary key.**
* **All values in a column are atomic (no repeating groups or arrays).**

**------------- Non 1 NF----------------------**

**OrderID Products**

**1 Laptop, Mouse**

**Orders**

**-------------1 NF----------------------**

**OrderID Product**

**1 Laptop**

**2 Mouse**

**Second Normal Form (2NF):**

* **Satisfies 1NF.**
* **Removes partial dependency (a non-prime attribute depends only on part of a composite key).**
* **Example:**
  + **Non-2NF:**

**Orders(OrderID, ProductID, ProductName, CustomerID)**

* + **In 2NF:**

**Orders(OrderID, ProductID, CustomerID)**

**Products(ProductID, ProductName)**

** Third Normal Form (3NF):**

* **Satisfies 2NF.**
* **Removes transitive dependency (non-prime attributes depend on other non-prime attributes).**
* **Example:**
  + **Non-3NF:**

**Orders(OrderID, ProductID, CustomerName, CustomerAddress)**

* + **In 3NF:**

**Orders(OrderID, ProductID, CustomerID)**

**Customers(CustomerID, CustomerName, CustomerAddress)**

**Explain the Concept of Database Sharding**

**Database Sharding is a technique of horizontally partitioning a database to distribute data across multiple servers. Each shard holds a subset of the data, improving scalability and performance.**

**How Sharding Works:**

* **Data is split based on a shard key (e.g., UserID, Region).**
* **Each shard operates as an independent database.**

**Advantages:**

* **Scalability: Handles more data and traffic.**
* **Performance: Reduces query load by limiting the data searched.**

**Disadvantages:**

* **Complexity: Increases the complexity of design and maintenance.**
* **Rebalancing: Moving data between shards can be challenging.**

**Example:**

* **A global e-commerce platform shards its customer data by region:**
  + **Shard1: Customers in North America.**
  + **Shard2: Customers in Europe.**
  + **Shard3: Customers in Asia.**

**What is a Partitioned Table in SQL?**

**A partitioned table divides a large table into smaller, more manageable pieces called partitions. Each partition stores a subset of the table’s data based on a partition key, such as range, list, or hash.**

**Types of Partitioning:**

1. **Range Partitioning: Data is divided into partitions based on a range of values.**

**CREATE TABLE Sales (**

**SaleID INT,**

**SaleDate DATE,**

**Amount DECIMAL(10, 2)**

**) PARTITION BY RANGE (YEAR(SaleDate)) (**

**PARTITION p2019 VALUES LESS THAN (2020),**

**PARTITION p2020 VALUES LESS THAN (2021),**

**PARTITION p2021 VALUES LESS THAN (2022)**

**);**

1. **List Partitioning: Data is partitioned based on a list of values.**
2. **Hash Partitioning: Data is partitioned by applying a hash function on a column.**

| **Aspect** | **Sharding** | **Replication** |
| --- | --- | --- |
| **Purpose** | **Distributes data horizontally across multiple servers.** | **Maintains multiple copies of the same data across servers.** |
| **Data Distribution** | **Each shard holds a unique subset of data.** | **Each replica holds the entire dataset.** |
| **Scalability** | **Increases capacity by adding more shards (horizontal scaling).** | **Improves fault tolerance and read performance but not storage capacity.** |
| **Use Case** | **Large-scale applications with high write and read demands.** | **High availability and fault-tolerant systems.** |
| **Example** | **Sharding users by region (e.g., North America, Europe, Asia).** | **Creating read replicas for an e-commerce site to handle concurrent requests.** |

* 1. **How Do You Find All Employees in a Department with No Manager?**

**SELECT e.EmployeeID, e.Name, e.DepartmentID**

**FROM Employees e**

**LEFT JOIN Employees m ON e.ManagerID = m.EmployeeID**

**WHERE e.ManagerID IS NULL OR m.EmployeeID IS NULL;**

* 1. **Write a Query to Find Customers with No Orders in the Last 6 Months**

**SELECT c.CustomerID, c.Name**

**FROM Customers c**

**LEFT JOIN Orders o ON c.CustomerID = o.CustomerID AND o.OrderDate >= DATEADD(MONTH, -6, GETDATE())**

**WHERE o.OrderID IS NULL;**

**Alternate Query Using NOT EXISTS:**

**SELECT c.CustomerID, c.Name**

**FROM Customers c**

**WHERE NOT EXISTS (**

**SELECT 1**

**FROM Orders o**

**WHERE c.CustomerID = o.CustomerID AND o.OrderDate >= DATEADD(MONTH, -6, GETDATE())**

**);**