SQL Interview Questions

Last Updated : 11 Jun, 2025

Are you preparing for a **SQL interview**? **SQL** is a standard database language used for **accessing** and **manipulating data** in databases. It stands for **Structured Query Language** and was developed by **IBM** in the 1970's, SQL allows us to **create**, **read**, **update**, and **delete** data with simple yet effective commands. For both **newcomers** and **seasoned professionals**, mastering SQL is a must-have skill in today's data-driven job market

Here, we cover **100 SQL Interview Questions with answers** asked in SQL developer interviews at MAANG and other high-paying companies. Whether we're preparing for our first **data-related role** or seeking to **advance our career**, this guide will walk us through the most commonly asked **SQL interview questions** to help us stand out in **competitive job interviews.**

Table of Content

SQL Basic Interview Questions

"**SQL Basic Interview Questions**" covers fundamental concepts that are essential for anyone preparing for a SQL **interview**. Explore these essential questions to build a strong understanding and boost our confidence in **SQL basics**.

1. What is SQL?

SQL (Structured Query Language) is a standard programming language used to communicate with **relational databases**. It allows users to create, read, update, and delete data, and provides commands to define database schema and manage database security.

2. What is a database?

A database is an **organized collection of data** stored electronically, typically structured in tables with rows and columns. It is managed by a **database management system** (DBMS), which allows for efficient **storage**, **retrieval**, and **manipulation** of data.

3. What are the main types of SQL commands?

SQL commands are broadly classified into:

- **DDL (Data Definition Language):** CREATE, ALTER, DROP, TRUNCATE.
- **DML (Data Manipulation Language):** SELECT, INSERT, UPDATE, DELETE.
- **DCL (Data Control Language):** GRANT, REVOKE.
- **TCL (Transaction Control Language):** COMMIT, ROLLBACK, SAVEPOINT.

4. What is the difference between CHAR and VARCHAR2 data types?

- **CHAR:** Fixed-length storage. If the defined length is not fully used, it is padded with spaces.
- **VARCHAR2:** Variable-length storage. Only the actual data is stored, saving space when the full length is not needed.

5. What is a primary key?

A primary key is a unique identifier for each record in a table. It ensures that no two rows have the same value in the primary key column(s), and it does not allow NULL values.

6. What is a foreign key?

A foreign key is a column (or set of columns) in one table that refers to the primary key in another table. It establishes and enforces a relationship between the two tables, ensuring data integrity.

7. What is the purpose of the DEFAULT constraint?

The DEFAULT constraint assigns a default value to a column when no value is provided during an **INSERT operation**. This helps maintain consistent data and simplifies data entry.

A foreign key is a column (or set of columns) in one table that refers to the primary key in another table. It establishes and enforces a relationship between the two tables, ensuring data integrity.

7. What is the purpose of the DEFAULT constraint?

The DEFAULT constraint assigns a default value to a column when no value is provided during an **INSERT operation**. This helps maintain consistent data and simplifies data entry.

8. What is normalization in databases?

Normalization is the process of organizing data in a database to **reduce redundancy** and **improve data integrity**. This involves dividing large tables into smaller, related tables and defining relationships between them to ensure consistency and avoid anomalies.

9. What is denormalization, and when is it used?

Denormalization is the process of combining **normalized tables** into larger tables for performance reasons. It is used when **complex queries** and joins slow down data retrieval, and the performance benefits outweigh the **drawbacks of redundancy**.

10. What is a query in SQL?

A query is a SQL statement used to retrieve, update, or manipulate data in a **database**. The most common type of query is a SELECT statement, which fetches data from one or more tables based on specified conditions.

11. What are the different operators available in SQL?

- **Arithmetic Operators:** +, -, *, /, %
- **Comparison Operators:** =, !=, <>, >, <, >=, <=
- **Logical Operators:** AND, OR, NOT
- **Set Operators:** UNION, INTERSECT, EXCEPT
- **Special Operators:** BETWEEN, IN, LIKE, IS NULL

12. What is a view in SQL?

A view is a **virtual table** created by a **SELECT query**. It does not store data itself, but presents data from one or more tables in a structured way. Views simplify complex queries, improve readability, and enhance security by restricting access to specific rows or columns.

13. What is the purpose of the UNIQUE constraint?

The UNIQUE constraint ensures that all values in a column (or combination of columns) are **distinct**. This prevents duplicate values and helps maintain data integrity.

14. What are the different types of joins in SQL?

- **INNER JOIN:** Returns rows that have matching values in both tables.
- **LEFT JOIN (LEFT OUTER JOIN):** Returns all rows from the left table, and matching rows from the right table.
- **RIGHT JOIN (RIGHT OUTER JOIN):** Returns all rows from the right table, and matching rows from the left table.
- **FULL JOIN (FULL OUTER JOIN):** Returns all rows when there is a match in either table.
- **CROSS JOIN:** Produces the Cartesian product of two tables.

15. What is the difference between INNER JOIN and OUTER JOIN?

- **INNER JOIN:** Returns only rows where there is a match in both tables.
- **OUTER JOIN:** Returns all rows from one table (LEFT, RIGHT, or FULL), and the matching rows from the other table. If there is no match, NULL values are returned for the non-matching side.

16. What is the purpose of the GROUP BY clause?

The GROUP BY clause is used to arrange **identical data** into **groups**. It is typically used with aggregate functions (such as COUNT, SUM, AVG) to perform calculations on each group rather than on the entire dataset.

17. What are aggregate functions in SQL?

from the other table. If there is no match, NULL values are returned for the non-matching side.

16. What is the purpose of the GROUP BY clause?

The GROUP BY clause is used to arrange **identical data** into **groups**. It is typically used with aggregate functions (such as COUNT, SUM, AVG) to perform calculations on each group rather than on the entire dataset.

17. What are aggregate functions in SQL?

Aggregate functions perform calculations on a set of values and return a single value. Common aggregate functions include:

- COUNT(): Returns the number of rows.
- SUM(): Returns the total sum of values.
- AVG(): Returns the average of values.
- MIN(): Returns the smallest value.
- MAX(): Returns the largest value.

18. What is a subquery?

A subquery is a query nested within another query. It is often used in the **WHERE clause** to filter data based on the results of another query, making it easier to handle complex conditions.

19. What is the difference between the WHERE and HAVING clauses?

- **WHERE:** Filters rows before any grouping takes place.
- **HAVING:** Filters grouped data after the GROUP BY clause has been applied.
  In short, WHERE applies to individual rows, while HAVING applies to groups.

20. What are indexes, and why are they used?

Indexes are **database objects** that improve query performance by allowing **faster retrieval of rows**. They function like a book's index, making it quicker to find specific data without scanning the entire table. However, indexes require **additional storage** and can slightly slow down **data modification** operations.

21. What is the difference between DELETE and TRUNCATE commands?

- **DELETE:** Removes rows one at a time and records each deletion in the transaction log, allowing rollback. It can have a WHERE clause.
- **TRUNCATE:** Removes all rows at once without logging individual row deletions. It cannot have a WHERE clause and is faster than DELETE for large data sets.

22. What is the purpose of the SQL ORDER BY clause?

The ORDER BY clause sorts the result set of a query in either **ascending** (default) or **descending order**, based on one or more columns. This helps present the data in a more meaningful or readable sequence.

23. What are the differences between SQL and NoSQL databases?

- **SQL Databases:**
  - Use structured tables with rows and columns.
  - Rely on a fixed schema.
  - Offer **ACID** properties.
- **NoSQL Databases:**
  - Use flexible, schema-less structures (e.g., key-value pairs, document stores).
  - Are designed for horizontal scaling.
  - Often focus on performance and scalability over strict consistency.

24. What is a table in SQL?

A table is a **structured collection** of related data organized into rows and columns. Columns define the type of data stored, while rows contain individual records.

25. What are the types of constraints in SQL?

Common constraints include:

- Are designed for horizontal scaling.
- Often focus on performance and scalability over strict consistency.

24. What is a table in SQL?

A table is a **structured collection** of related data organized into rows and columns. Columns define the type of data stored, while rows contain individual records.

25. What are the types of constraints in SQL?

Common constraints include:

- **NOT NULL:** Ensures a column cannot have NULL values.
- **UNIQUE:** Ensures all values in a column are distinct.
- **PRIMARY KEY:** Uniquely identifies each row in a table.
- **FOREIGN KEY:** Ensures referential integrity by linking to a primary key in another table.
- **CHECK:** Ensures that all values in a column satisfy a specific condition.
- **DEFAULT:** Sets a default value for a column when no value is specified.

26. What is a cursor in SQL?

A cursor is a database object used to **retrieve**, **manipulate**, and traverse through rows in a result set one row at a time. Cursors are helpful when performing operations that must be processed sequentially rather than in a set-based manner.

27. What is a trigger in SQL?

A trigger is a set of SQL statements that automatically execute in response to certain events on a table, such as **INSERT**, **UPDATE**, or **DELETE**. Triggers help maintain **data consistency**, enforce business rules, and implement complex integrity constraints.

28. What is the purpose of the SQL SELECT statement?

The SELECT statement retrieves data from one or more tables. It is the most commonly used command in SQL, allowing users to filter, sort, and display data based on specific criteria.

29. What are NULL values in SQL?

NULL represents a missing or unknown value. It is different from zero or an empty string. NULL values indicate that the data is not available or applicable.

30. What is a stored procedure?

A stored procedure is a precompiled set of SQL statements stored in the **database**. It can take input parameters, perform logic and queries, and return output values or result sets. Stored procedures improve **performance** and **maintainability** by centralizing business logic.

SQL Intermediate Interview Questions

This section covers moderately complex **SQL topics** like **advanced queries**, **multi-table joins**, **subqueries**, and **basic optimization techniques**. These questions help enhance skills for both **database developers** and **administrators**, preparing us for more **technical SQL challenges** in the field.

31. What is the difference between DDL and DML commands?

1. DDL (Data Definition Language):

These commands are used to **define** and **modify the structure of database** objects such as **tables**, **indexes**, and **views**. For example, the **`CREATE` command** creates a new table, the **`ALTER` command** modifies an existing table, and the **`DROP` command** removes a table entirely. DDL commands primarily focus on the schema or structure of the database.

Example:

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

2. DML (Data Manipulation Language):

These commands deal with the **actual data stored** within database objects. For instance, the **`INSERT` command** adds rows of data to a table, the **`UPDATE`** command modifies existing data, and the **`DELETE`** command removes rows from a table. In short, DML commands allow you to query and manipulate the data itself rather than the structure.

Example:

database developers and **administrators**, preparing us for more **technical SQL challenges** in the field.

31. What is the difference between DDL and DML commands?

1. DDL (Data Definition Language):

These commands are used to **define** and **modify the structure of database** objects such as **tables**, **indexes**, and **views**. For example, the **`CREATE` command** creates a new table, the **`ALTER` command** modifies an existing table, and the **`DROP` command** removes a table entirely. DDL commands primarily focus on the schema or structure of the database.

Example:

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

2. DML (Data Manipulation Language):

These commands deal with the **actual data stored** within database objects. For instance, the **`INSERT` command** adds rows of data to a table, the **`UPDATE`** command modifies existing data, and the **`DELETE`** command removes rows from a table. In short, DML commands allow you to query and manipulate the data itself rather than the structure.

Example:

```
INSERT INTO Employees (ID, Name) VALUES (1, 'Alice');
```

32. What is the purpose of the ALTER command in SQL?

The `ALTER` command is used to **modify the structure** of an existing database object. This command is essential for adapting our **database schema** as requirements evolve.

- Add or drop a column in a table.
- Change a column's data type.
- Add or remove constraints.
- Rename columns or tables.
- Adjust indexing or storage settings.

33. What is a composite primary key?

A **composite primary key** is a primary key made up of two or more columns. Together, these columns must form a unique combination for each row in the table. It's used when a single column isn't sufficient to uniquely identify a record.

Example:

Consider an Orders table where `OrderID` and `ProductID` together uniquely identify each record because multiple orders might include the same product, but not within the same order.

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

34. How is data integrity maintained in SQL databases?

Data integrity refers to the **accuracy**, **consistency**, and **reliability** of the data stored in the database. SQL databases maintain data integrity through several mechanisms:

- **Constraints:** Ensuring that certain conditions are always met. For example, `NOT NULL` ensures a column cannot have missing values, `FOREIGN KEY` ensures a valid relationship between tables, and `UNIQUE` ensures no duplicate values.
- **Transactions:** Ensuring that a series of operations either all succeed or all fail, preserving data consistency.
- **Triggers:** Automatically enforcing rules or validations before or after changes to data.
- **Normalization:** Organizing data into multiple related tables to minimize redundancy and prevent anomalies.

  These measures collectively ensure that the data remains reliable and meaningful over time.

35. What are the advantages of using stored procedures?

- Improved Performance: Stored procedures are compiled and cached in the database, making their execution faster than sending multiple individual queries.

```
);
```

34. How is data integrity maintained in SQL databases?

Data integrity refers to the **accuracy**, **consistency**, and **reliability** of the data stored in the database. SQL databases maintain data integrity through several mechanisms:

- **Constraints:** Ensuring that certain conditions are always met. For example, `NOT NULL` ensures a column cannot have missing values, `FOREIGN KEY` ensures a valid relationship between tables, and `UNIQUE` ensures no duplicate values.
- **Transactions:** Ensuring that a series of operations either all succeed or all fail, preserving data consistency.
- **Triggers:** Automatically enforcing rules or validations before or after changes to data.
- **Normalization:** Organizing data into multiple related tables to minimize redundancy and prevent anomalies.

  These measures collectively ensure that the data remains reliable and meaningful over time.

35. What are the advantages of using stored procedures?

- **Improved Performance:** Stored procedures are precompiled and cached in the database, making their execution faster than sending multiple individual queries.
- **Reduced Network Traffic:** By executing complex logic on the server, fewer round trips between the application and database are needed.
- **Enhanced Security:** Stored procedures can restrict direct access to underlying tables, allowing users to execute only authorized operations.
- **Reusability and Maintenance:** Once a procedure is written, it can be reused across multiple applications. If business logic changes, you only need to update the stored procedure, not every application that uses it.

36. What is a UNION operation, and how is it used?

The `UNION` operator combines the result sets of two or more `SELECT` queries into a single result set, removing **duplicate rows**. The result sets must have the same number of columns and compatible data types for corresponding columns.

Example:

```
SELECT Name FROM Customers
UNION
SELECT Name FROM Employees;
```

37. What is the difference between UNION and UNION ALL?

- **UNION:** Combines result sets from two queries and removes **duplicate rows**, ensuring only unique records are returned.
- **UNION ALL:** Combines the result sets without removing duplicates, meaning all records from both queries are included.
- Performance-wise, `UNION ALL` is faster than UNION because it doesn't perform the additional operation of eliminating duplicates.

Example:

```
SELECT Name FROM Customers
UNION ALL
SELECT Name FROM Employees;
```

38. How does the CASE statement work in SQL?

The `CASE` statement is SQL's way of implementing **conditional logic** in queries. It evaluates conditions and returns a value based on the first condition that evaluates to true. If no condition is met, it can return a default value using the **ELSE clause**.

Example:

```
SELECT ID,
       CASE
           WHEN Salary > 100000 THEN 'High'
           WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'
           ELSE 'Low'
       END AS SalaryLevel
FROM Employees;
```

```
SELECT Name FROM Employees;
```

38. How does the CASE statement work in SQL?

The `CASE` statement is SQL's way of implementing **conditional logic** in queries. It evaluates conditions and returns a value based on the first condition that evaluates to true. If no condition is met, it can return a default value using the **ELSE clause**.

Example:

```
SELECT ID,
       CASE
           WHEN Salary > 100000 THEN 'High'
           WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'
           ELSE 'Low'
       END AS SalaryLevel
FROM Employees;
```

39. What are scalar functions in SQL?

Scalar functions operate on individual values and return a single value as a result. They are often used for formatting or converting data. Common examples include:

- **LEN():** Returns the length of a string.
- **ROUND():** Rounds a numeric value.
- **CONVERT():** Converts a value from one data type to another.

Example:

```
SELECT LEN('Example') AS StringLength;
```

40. What is the purpose of the COALESCE function?

The **`COALESCE` function** returns the first non-NULL value from a list of expressions. It's commonly used to provide default values or handle missing data gracefully.

Example:

```
SELECT COALESCE(NULL, NULL, 'Default Value') AS Result;
```

41. What are the differences between SQL's COUNT() and SUM() functions?

1. COUNT(): Counts the number of rows or non-NULL values in a column.

Example:

```
SELECT COUNT(*) FROM Orders;
```

2. SUM(): Adds up all numeric values in a column.

Example:

```
SELECT SUM(TotalAmount) FROM Orders;
```

42. What is the difference between the NVL and NVL2 functions?

- **NVL():** Replaces a NULL value with a specified replacement value. **Example**: `NVL(Salary, 0)` will replace `NULL` with `0`.
- **NVL2():** Evaluates two arguments:
    - If the first argument is **NOT NULL**, returns the second argument.
    - If the first argument is **NULL**, returns the third argument.

Example:

```
SELECT NVL(Salary, 0) AS AdjustedSalary FROM Employees;  -- Replaces NULL with
0

SELECT NVL2(Salary, Salary, 0) AS AdjustedSalary FROM Employees;  -- If Salary
is NULL, returns 0; otherwise, returns Salary.
```

43. How does the RANK() function differ from DENSE_RANK()?

```
SELECT SUM(TotalAmount) FROM Orders;
```

42. What is the difference between the NVL and NVL2 functions?

- **NVL():** Replaces a NULL value with a specified replacement value. **Example**: `NVL(Salary, 0)` will replace `NULL` with `0`.
- **NVL2():** Evaluates two arguments:
  - If the first argument is **NOT NULL**, returns the second argument.
  - If the first argument is **NULL**, returns the third argument.

Example:

```
SELECT NVL(Salary, 0) AS AdjustedSalary FROM Employees;  -- Replaces NULL with
0

SELECT NVL2(Salary, Salary, 0) AS AdjustedSalary FROM Employees;  -- If Salary
is NULL, returns 0; otherwise, returns Salary.
```

43. How does the RANK() function differ from DENSE_RANK()?

- **RANK():** Assigns a rank to each row, with gaps if there are ties.
- **DENSE_RANK():** Assigns consecutive ranks without any gaps.

Example:

```
SELECT Name, Salary, RANK() OVER (ORDER BY Salary DESC) AS Rank
FROM Employees;
```

If two employees have the same salary, they get the same rank, but `RANK()` will skip a number for the next rank, while `DENSE_RANK()` will not.

44. What is the difference between ROW_NUMBER() and RANK()?

- **ROW_NUMBER():** Assigns a unique number to each row regardless of ties.
- **RANK():** Assigns the same number to tied rows and leaves gaps for subsequent ranks.

Example:

```
SELECT Name, ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum
FROM Employees;
```

45. What are common table expressions (CTEs) in SQL?

A CTE is a temporary result set defined within a query. It improves query readability and can be referenced multiple times.

Example:

```
WITH TopSalaries AS (
    SELECT Name, Salary
    FROM Employees
    WHERE Salary > 50000
)
SELECT * FROM TopSalaries WHERE Name LIKE 'A%';
```

46. What are window functions, and how are they used?

Window functions allow you to perform calculations across a set of table rows that are related to the current row within a result set, without collapsing the result set into a single row. These functions can be used to compute running totals, moving averages, rank rows, etc.

Example: Calculating a running total

```
SELECT Name, Salary,
    SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal
FROM Employees;
```

47. What is the difference between an index and a key in SQL?

1. Index

- An index is a database object created to **speed up data retrieval**. It stores a sorted reference to table data, which helps the database engine find rows more quickly than scanning the entire

```
    SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal
FROM Employees;
```

47. What is the difference between an index and a key in SQL?

1. Index

- An index is a database object created to **speed up data retrieval**. It stores a sorted reference to table data, which helps the database engine find rows more quickly than scanning the entire table.
- **Example:** A non-unique index on a column like `LastName` allows quick lookups of rows where the last name matches a specific value.

2. **Key**

- A key is a logical concept that enforces rules for uniqueness or relationships in the data.
- For instance, a **PRIMARY KEY** uniquely identifies each row in a table and ensures that no duplicate or NULL values exist in the key column(s).
- A **FOREIGN KEY** maintains referential integrity by linking rows in one table to rows in another.

48. How does indexing improve query performance?

Indexing allows the database to locate and access the rows corresponding to a **query condition** much faster than scanning the entire table. Instead of reading each row sequentially, the database uses the index to **jump directly** to the relevant data pages. This reduces the number of disk **I/O operations** and speeds up query execution, especially for large tables.

Example:

```
CREATE INDEX idx_lastname ON Employees(LastName);
SELECT * FROM Employees WHERE LastName = 'Smith';
```

The index on `LastName` lets the database quickly find all rows matching 'Smith' without scanning every record.

49. What are the trade-offs of using indexes in SQL databases?

Advantages

- Faster query performance, especially for SELECT queries with WHERE clauses, JOIN conditions, or ORDER BY clauses.
- Improved sorting and filtering efficiency.

Disadvantages:

- Increased storage space for the index structures.
- Additional overhead for write operations (INSERT, UPDATE, DELETE), as indexes must be updated whenever the underlying data changes.
- Potentially **slower bulk data load**s or batch inserts due to the need to maintain index integrity.

In short, indexes make read operations faster but can slow down write operations and increase storage requirements.

50. What is the difference between clustered and non-clustered indexes?

1. Clustered Index:

- Organizes the physical data in the table itself in the order of the indexed column(s).
- A table can have only one clustered index.
- Improves range queries and queries that sort data.
- Example: If `EmployeeID` is the clustered index, the rows in the table are stored physically sorted by `EmployeeID`.

2. Non-Clustered Index:

- Maintains a separate structure that contains a reference (or pointer) to the physical data in the table.
- A table can have multiple non-clustered indexes.
- Useful for specific query conditions that aren't related to the primary ordering of the data.
- Example: A non-clustered index on `LastName` allows fast lookups by last name even if the table is sorted by another column.

51. What are temporary tables, and how are they used?

Temporary tables are tables that exist only for the duration of a **session** or a **transaction**. They are useful for storing intermediate results, simplifying complex queries, or performing operations on subsets of data without modifying the main table.

1. Local Temporary Tables:

sorted by another column.

51. What are temporary tables, and how are they used?

Temporary tables are tables that exist only for the duration of a **session** or a **transaction**. They are useful for storing intermediate results, simplifying complex queries, or performing operations on subsets of data without modifying the main tables.

1. Local Temporary Tables:

- Prefixed with # (e.g., `#TempTable`).
- Only visible to the session that created them.
- Automatically dropped when the session ends.

2. Global Temporary Tables:

- Prefixed with ## (e.g., `##GlobalTempTable`).
- Visible to all sessions.
- Dropped when all sessions that reference them are closed.

Example:

```
CREATE TABLE #TempResults (ID INT, Value VARCHAR(50));
INSERT INTO #TempResults VALUES (1, 'Test');
SELECT * FROM #TempResults;
```

52. What is a materialized view, and how does it differ from a standard view?

- **Standard View:**
  - A virtual table defined by a query.
  - Does not store data; the underlying query is executed each time the view is referenced.
  - A standard view shows real-time data.
- **Materialized View:**
  - A physical table that stores the result of the query.
  - Data is precomputed and stored, making reads faster.
  - Requires periodic refreshes to keep data up to date.
  - materialized view is used to store aggregated sales data, updated nightly, for fast reporting.

53. What is a sequence in SQL?

A sequence is a database object that generates a series of **unique numeric values**. It's often used to produce unique identifiers for primary keys or other columns requiring sequential values.

Example:

```
CREATE SEQUENCE seq_emp_id START WITH 1 INCREMENT BY 1;
SELECT NEXT VALUE FOR seq_emp_id; -- Returns 1
SELECT NEXT VALUE FOR seq_emp_id; -- Returns 2
```

54. What are the advantages of using sequences over identity columns?

1. Greater Flexibility:

- Can specify start values, increments, and maximum values.
- Can be easily reused for multiple tables.

2. Dynamic Adjustment: Can alter the sequence without modifying the table structure.

3. Cross-Table Consistency: Use a single sequence for multiple related tables to ensure unique identifiers across them.
In short, sequences offer more control and reusability than identity columns.

55. How do constraints improve database integrity?

Constraints enforce rules that the data must follow, preventing invalid or inconsistent data from being entered:

- **NOT NULL:** Ensures that a column cannot contain NULL values.
- **UNIQUE:** Ensures that all values in a column are distinct.
- **PRIMARY KEY:** Combines NOT NULL and UNIQUE, guaranteeing that each row is uniquely identifiable.
- **FOREIGN KEY:** Ensures referential integrity by requiring values in one table to match primary key values in another.
- **CHECK:** Validates that values meet specific [illegible]

3. Cross-Table Consistency: Use a single sequence for multiple related tables to ensure unique identifiers across them.
In short, sequences offer more control and reusability than identity columns.

55. How do constraints improve database integrity?

Constraints enforce rules that the data must follow, preventing invalid or inconsistent data from being entered:

- **NOT NULL:** Ensures that a column cannot contain NULL values.
- **UNIQUE:** Ensures that all values in a column are distinct.
- **PRIMARY KEY:** Combines NOT NULL and UNIQUE, guaranteeing that each row is uniquely identifiable.
- **FOREIGN KEY:** Ensures referential integrity by requiring values in one table to match primary key values in another.
- **CHECK:** Validates that values meet specific criteria (e.g., `CHECK (Salary > 0)`).
  By automatically enforcing these rules, constraints maintain data reliability and consistency.

56. What is the difference between a local and a global temporary table?

- **Local Temporary Table:**
  - Prefixed with # (e.g., `#TempTable`).
  - Exists only within the session that created it.
  - Automatically dropped when the session ends.
- **Global Temporary Table:**
  - Prefixed with ## (e.g., `##GlobalTempTable`).
  - Visible to all sessions.
  - Dropped only when all sessions referencing it are closed.

Example:

```
CREATE TABLE #LocalTemp (ID INT);
CREATE TABLE ##GlobalTemp (ID INT);
```

57. What is the purpose of the SQL MERGE statement?

The **`MERGE` statement** combines multiple operations INSERT, UPDATE, and DELETE into one. It is used to synchronize two tables by:

- Inserting rows that don't exist in the target table.
- Updating rows that already exist.
- Deleting rows from the target table based on conditions

Example:

```
MERGE INTO TargetTable T
USING SourceTable S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.Value = S.Value
WHEN NOT MATCHED THEN
    INSERT (ID, Value) VALUES (S.ID, S.Value);
```

58. How can you handle duplicates in a query without using DISTINCT?

1. GROUP BY: Aggregate rows to eliminate duplicates

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

2. **ROW_NUMBER():** Assign a unique number to each row and filter by that

```
WITH CTE AS (
    SELECT Column1, Column2, ROW_NUMBER() OVER (PARTITION BY Column1 ORDER BY
Column2) AS RowNum
    FROM TableName
)
SELECT * FROM CTE WHERE RowNum = 1;
```

59. What is a correlated subquery?

A correlated subquery is a subquery that references columns from the outer query. It is re-executed

```
WITH CTE AS (
    SELECT Column1, Column2, ROW_NUMBER() OVER (PARTITION BY Column1 ORDER BY
Column2) AS RowNum
    FROM TableName
)
SELECT * FROM CTE WHERE RowNum = 1;
```

59. What is a correlated subquery?

A correlated subquery is a subquery that references columns from the outer query. It is re-executed for each row processed by the outer query. This makes it more dynamic, but potentially less efficient.

Example:

```
SELECT Name,
       (SELECT COUNT(*)
        FROM Orders
        WHERE Orders.CustomerID = Customers.CustomerID) AS OrderCount
FROM Customers;
```

60. What are partitioned tables, and when should we use them?

Partitioned tables divide data into **smaller**, more **manageable segments** based on a column's value (e.g., date or region). Each partition is stored separately, making queries that target a specific partition more efficient. It is used when

- Large tables with millions or billions of rows.
- Scenarios where queries frequently filter on partitioned columns (e.g., year, region).
- To improve maintenance operations, such as archiving older partitions without affecting the rest of the table.

SQL Advanced Interview Questions

This section covers **complex SQL topics**, including **performance tuning**, **complex indexing strategies**, **transaction isolation levels**, and **advanced query optimization techniques**. By tackling these challenging questions, we'll gain a deeper understanding of SQL, preparing us for **senior-level roles** and **technical interviews**.

61. What are the ACID properties of a transaction?

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability—four key properties that ensure database transactions are processed reliably.

1. **Atomicity:**

- A transaction is treated as a single unit of work, meaning all operations must succeed or fail as a whole.
- If any part of the transaction fails, the entire transaction is rolled back.

2. **Consistency:**

- A transaction must take the database from one valid state to another, maintaining all defined rules and constraints.
- This ensures data integrity is preserved throughout the transaction process.

3. **Isolation:**

- Transactions should not interfere with each other.
- Even if multiple transactions occur simultaneously, each must operate as if it were the only one in the system until it is complete.

4. **Durability:**

- Once a transaction is committed, its changes must persist, even in the event of a system failure.
- This ensures the data remains stable after the transaction is successfully completed.

62. What are the differences between isolation levels in SQL?

Isolation levels define the extent to which the operations in one transaction are isolated from those in other transactions. They are critical for **managing concurrency** and ensuring data integrity. Common isolation levels include:

1. Read Uncommitted:

- Allows reading uncommitted changes from other transactions.
- Can result in dirty reads, where a transaction reads data that might later be rolled back.

- Once a transaction is committed, its changes must persist, even in the event of a system failure.
- This ensures the data remains stable after the transaction is successfully completed.

62. What are the differences between isolation levels in SQL?

Isolation levels define the extent to which the operations in one transaction are isolated from those in other transactions. They are critical for **managing concurrency** and ensuring data integrity. Common isolation levels include:

1. Read Uncommitted:

- Allows reading uncommitted changes from other transactions.
- Can result in dirty reads, where a transaction reads data that might later be rolled back.

2. Read Committed:

- Ensures a transaction can only read committed data.
- Prevents dirty reads but does not protect against non-repeatable reads or phantom reads.

3. Repeatable Read:

- Ensures that if a transaction reads a row, that row cannot change until the transaction is complete.
- Prevents dirty reads and non-repeatable reads but not phantom reads.

4. Serializable:

- The highest level of isolation.
- Ensures full isolation by effectively serializing transactions, meaning no other transaction can read or modify data that another transaction is using.
- Prevents dirty reads, non-repeatable reads, and phantom reads, but may introduce performance overhead due to locking and reduced concurrency.

63. What is the purpose of the WITH (NOLOCK) hint in SQL Server?

- The **WITH (NOLOCK)** hint allows a query to read data without acquiring shared locks, effectively reading uncommitted data.
- It can improve performance by **reducing contention for locks**, especially on large tables that are frequently updated.
- Results may be inconsistent or unreliable, as the data read might change or be rolled back.

Example:

```
SELECT *
FROM Orders WITH (NOLOCK);
```

This query fetches data from the `Orders` table without waiting for other transactions to release their locks.

64. How do you handle deadlocks in SQL databases?

Deadlocks occur when two or more transactions hold resources that the other transactions need, resulting in a cycle of dependency that prevents progress. Strategies to handle deadlocks include:

1. **Deadlock detection and retry:**

- Many database systems have mechanisms to detect deadlocks and terminate one of the transactions to break the cycle.
- The terminated transaction can be retried after the other transactions complete.

2. **Reducing lock contention:**

- Use indexes and optimized queries to minimize the duration and scope of locks.
- Break transactions into smaller steps to reduce the likelihood of conflicts.

3. **Using proper isolation levels:**

- In some cases, lower isolation levels can help reduce locking.
- Conversely, higher isolation levels (like Serializable) may ensure a predictable order of operations, reducing deadlock risk.

4. **Consistent ordering of resource access:**

- Ensure that transactions acquire resources in the same order to prevent cyclical dependencies.

65. What is a database snapshot, and how is it used?

A database snapshot is a read-only, static view of a database at a specific point in time.

- Reporting: Allowing users to query a consistent dataset without affecting live operations.
- Backup and recovery: Snapshots can serve as a point-in-time recovery source if changes go wrong.

- Ensure that transactions acquire resources in the same order to prevent cyclical dependencies.

65. What is a database snapshot, and how is it used?

A database snapshot is a read-only, static view of a database at a specific point in time.

- **Reporting:** Allowing users to query a consistent dataset without affecting live operations.
- **Backup and recovery:** Snapshots can serve as a point-in-time recovery source if changes need to be reversed.
- **Testing:** Providing a stable dataset for testing purposes without the risk of modifying the original data.

Example:

```
CREATE DATABASE MySnapshot ON
(
    NAME = MyDatabase_Data,
    FILENAME = 'C:\Snapshots\MyDatabase_Snapshot.ss'
)
AS SNAPSHOT OF MyDatabase;
```

66. What are the differences between OLTP and OLAP systems?

1. **OLTP (Online Transaction Processing)**

- Handles large volumes of simple transactions (e.g., order entry, inventory updates).
- Optimized for fast, frequent reads and writes.
- Normalized schema to ensure data integrity and consistency.
- Examples: e-commerce sites, banking systems.

2. **OLAP (Online Analytical Processing)**

- Handles complex queries and analysis on large datasets.
- Optimized for read-heavy workloads and data aggregation.
- Denormalized schema (e.g., star or snowflake schemas) to support faster querying.
- Examples: Business intelligence reporting, data warehousing.

67. What is a live lock, and how does it differ from a deadlock?

1. Live Lock

- Occurs when two or more transactions keep responding to each other's changes, but no progress is made.
- Unlike a deadlock, the transactions are not blocked; they are actively running, but they cannot complete.

2. **Deadlock**

- A **deadlock** occurs when two or more transactions are waiting on each other's resources indefinitely, blocking all progress.
- No progress can be made unless one of the transactions is terminated

68. What is the purpose of the SQL EXCEPT operator?

The `EXCEPT` operator is used to return rows from one query's result set that are not present in another query's result set. It effectively performs a set difference, showing only the data that is **unique** to the first query.

Example:

```
SELECT ProductID FROM ProductsSold
EXCEPT
SELECT ProductID FROM ProductsReturned;
```

Use Case:

- To find discrepancies between datasets.
- To verify that certain data exists in one dataset but not in another.

Performance Considerations:

- **`EXCEPT` works** best when the datasets involved have appropriate indexing and when the result sets are relatively small.
- Large datasets without indexes may cause slower performance because the database has to compare each row.

69. How do you implement dynamic SQL, a[redacted] what are its advantages and risks?

indefinitely, blocking all progress.
- No progress can be made unless one of the transactions is terminated

68. What is the purpose of the SQL EXCEPT operator?

The `EXCEPT operator` is used to return rows from one query's result set that are not present in another query's result set. It effectively performs a set difference, showing only the data that is **unique** to the first query.

Example:

```
SELECT ProductID FROM ProductsSold
EXCEPT
SELECT ProductID FROM ProductsReturned;
```

Use Case:

- To find discrepancies between datasets.
- To verify that certain data exists in one dataset but not in another.

Performance Considerations:

- **`EXCEPT` works** best when the datasets involved have appropriate indexing and when the result sets are relatively small.
- Large datasets without indexes may cause slower performance because the database has to compare each row.

69. How do you implement dynamic SQL, and what are its advantages and risks?

Dynamic SQL is SQL code that is constructed and executed at **runtime** rather than being fully defined and static. In SQL Server: Use **`sp_executesql`** or `EXEC`. In other databases: Concatenate query strings and execute them using the respective command for the database platform.

Syntax:

```
DECLARE @sql NVARCHAR(MAX)
SET @sql = 'SELECT * FROM ' + @TableName
EXEC sp_executesql @sql;
```

Advantages:

- **Flexibility:** Dynamic SQL can adapt to different conditions, tables, or columns that are only known at runtime.
- **Simplifies Complex Logic:** Instead of writing multiple queries, a single dynamically constructed query can handle multiple scenarios.

Risks:

- **SQL Injection Vulnerabilities:** If user input is not sanitized, attackers can inject malicious SQL code.
- **Performance Overhead:** Because dynamic SQL is constructed at runtime, it may not benefit from cached execution plans, leading to slower performance.
- **Complexity in Debugging:** Dynamic queries can be harder to read and troubleshoot.

70. What is the difference between horizontal and vertical partitioning?

Partitioning is a database technique used to divide data into smaller, more manageable pieces.

- **Horizontal Partitioning:**
  - Divides the rows of a table into multiple partitions based on values in a specific column.
  - Example: Splitting a customer table into separate partitions by geographic region or by year.
  - **Use Case:** When dealing with large datasets, horizontal partitioning can improve performance by limiting the number of rows scanned for a query.
- **Vertical Partitioning:**
  - Divides the columns of a table into multiple partitions.
  - Example: Storing infrequently accessed columns (e.g., large text or binary fields) in a separate table or partition.
  - **Use Case:** Helps in optimizing storage and query performance by separating commonly used columns from less frequently accessed data.
- **Key Difference:**
  - Horizontal partitioning is row-based, focusing on distributing the dataset's rows across partitions.
  - Vertical partitioning is column-based, aiming to separate less-used columns into different partitions or tables.

cached execution plans, leading to slower performance.
- **Complexity in Debugging:** Dynamic queries can be harder to read and troubleshoot.

70. What is the difference between horizontal and vertical partitioning?

Partitioning is a database technique used to divide data into smaller, more manageable pieces.

- **Horizontal Partitioning:**
  - Divides the rows of a table into multiple partitions based on values in a specific column.
  - Example: Splitting a customer table into separate partitions by geographic region or by year.
  - **Use Case:** When dealing with large datasets, horizontal partitioning can improve performance by limiting the number of rows scanned for a query.
- **Vertical Partitioning:**
  - Divides the columns of a table into multiple partitions.
  - Example: Storing infrequently accessed columns (e.g., large text or binary fields) in a separate table or partition.
  - **Use Case:** Helps in optimizing storage and query performance by separating commonly used columns from less frequently accessed data.
- **Key Difference:**
  - Horizontal partitioning is row-based, focusing on distributing the dataset's rows across partitions.
  - Vertical partitioning is column-based, aiming to separate less-used columns into different partitions or tables.

71. What are the considerations for indexing very large tables?

1. Indexing Strategy:

- Focus on the most frequently queried columns or those involved in JOIN and WHERE conditions.
- Avoid indexing every column, as it increases storage and maintenance costs.

2. Index Types:

- Use clustered indexes for primary key lookups and range queries.
- Use non-clustered indexes for filtering, ordering, and covering specific queries.

3. Partitioned Indexes:

- If the table is partitioned, consider creating **local indexes** for each partition. This improves manageability and can speed up queries targeting specific partitions.

4. Maintenance Overhead:

- Index rebuilding and updating can be resource-intensive. Plan for regular index maintenance during off-peak hours.
- Monitor index fragmentation and rebuild indexes as necessary to maintain performance.

5. Monitoring and Tuning:

- Continuously evaluate query performance using execution plans and statistics.
- Remove unused or rarely accessed indexes to reduce maintenance costs.

6. Indexing large tables requires a careful approach to ensure that performance gains from faster queries outweigh the costs of increased storage and maintenance effort.

72. What is the difference between database sharding and partitioning?

1. Sharding

- Sharding involves splitting a database into multiple smaller, **independent databases** (shards). Each shard operates on a subset of the overall data and can be hosted on separate servers.
- Sharding is a horizontal scaling strategy that distributes data across multiple databases, typically to handle massive data volumes and high traffic.
- **Purpose:** Horizontal scaling to handle large volumes of data and high query loads.
- **Example:** A global user database might be divided into shards by region, such as a shard for North America, Europe, and Asia.
- **Key Benefit:** Each shard can be queried independently, reducing the load on any single server.

2. **Partitioning**

- Partitioning splits a single table into smaller, logical pieces, usually within the same database.
- Partitioning is a **logical organization of data** within a single database to optimize performance and manageability.
- **Purpose:** Improve query performance by reducing the amount of data scanned, and simplify maintenance tasks such as archiving or purging old data.
- Example: A sales table could be partitioned by month so that queries targeting recent sales do not

6. Indexing large tables requires a careful approach to ensure that performance gains from faster queries outweigh the costs of increased storage and maintenance effort.

72. What is the difference between database sharding and partitioning?

1. Sharding

- Sharding involves splitting a database into multiple smaller, **independent databases** (shards). Each shard operates on a subset of the overall data and can be hosted on separate servers.
- Sharding is a horizontal scaling strategy that distributes data across multiple databases, typically to handle massive data volumes and high traffic.
- **Purpose:** Horizontal scaling to handle large volumes of data and high query loads.
- **Example:** A global user database might be divided into shards by region, such as a shard for North America, Europe, and Asia.
- **Key Benefit:** Each shard can be queried independently, reducing the load on any single server.

2. **Partitioning**

- Partitioning splits a single table into smaller, logical pieces, usually within the same database.
- Partitioning is a **logical organization of data** within a single database to optimize performance and manageability.
- **Purpose:** Improve query performance by reducing the amount of data scanned, and simplify maintenance tasks such as archiving or purging old data.
- **Example:** A sales table could be partitioned by year so that queries targeting recent sales do not need to scan historical data.

73. What are the best practices for writing optimized SQL queries?

1. Write Simple, Clear Queries:

- Avoid overly complex joins and subqueries.
- Use straightforward, well-structured SQL that is easy to read and maintain.

2. Filter Data Early:

- Apply WHERE clauses as early as possible to reduce the amount of data processed.
- Consider using indexed columns in WHERE clauses for faster lookups.

3. **Avoid SELECT *:

- Retrieve only the columns needed. This reduces I/O and improves performance.

4. Use Indexes Wisely:

- Create indexes on columns that are frequently used in WHERE clauses, JOIN conditions, and ORDER BY clauses.
- Regularly review index usage and remove unused indexes.

5. Leverage Query Execution Plans:

- Use execution plans to identify bottlenecks, missing indexes, or inefficient query patterns.

6. Use Appropriate Join Types:

- Choose INNER JOIN, LEFT JOIN, or OUTER JOIN based on the data relationships and performance requirements.

7. Break Down Complex Queries:

- Instead of a single monolithic query, use temporary tables or CTEs to process data in stages.

8. Optimize Aggregations:

- Use GROUP BY and aggregate functions efficiently.
- Consider pre-aggregating data if queries frequently require the same computations.

9. Monitor Performance Regularly:

- Continuously analyze query performance and fine-tune as data volumes grow or usage patterns change.

74. How can you monitor query performance in a production database?

1. Use Execution Plans:

Review the execution plan of queries to understand how the database is retrieving data, which indexes are being used, and where potential bottlenecks exist.

2. Analyze Wait Statistics:

Identify where queries are waiting, such as on locks, I/O, or CPU, to pinpoint the cause of slowdowns.

- Continuously analyze query performance and fine-tune as data volumes grow or usage patterns change.

74. How can you monitor query performance in a production database?

1. Use Execution Plans:

Review the execution plan of queries to understand how the database is retrieving data, which indexes are being used, and where potential bottlenecks exist.

2. Analyze Wait Statistics:

Identify where queries are waiting, such as on locks, I/O, or CPU, to pinpoint the cause of slowdowns.

3. Leverage Built-in Monitoring Tools:

- SQL Server: Use Query Store, DMVs (Dynamic Management Views), and performance dashboards.
- MySQL: Use `EXPLAIN`, `SHOW PROFILE`, and the Performance Schema.
- PostgreSQL: Use `EXPLAIN (ANALYZE)`, `pg_stat_statements`, and log-based monitoring.

4. **Set Up Alerts and Baselines:**

- Monitor key performance metrics (query duration, IOPS, CPU usage) and set thresholds.
- Establish baselines to quickly identify when performance degrades.

5. **Continuous Query Tuning:**

- Regularly revisit and tune queries as data grows or application requirements change.
- Remove unused or inefficient indexes and re-evaluate the indexing strategy.

75. What are the trade-offs of using indexing versus denormalization?

1. Indexing

- **Advantages:**
  - Speeds up read operations and improves query performance without changing the data structure.
  - Can be applied incrementally and is reversible if not effective.
  - Consider indexing when you need faster lookups without altering the data model.
- **Disadvantages:**
  - Slows down write operations as indexes need to be maintained.
  - Requires additional storage.

2. **Denormalization**

- **Advantages:**
  - Simplifies query logic by storing pre-joined or aggregated data.
  - Can improve performance for read-heavy workloads where complex joins are frequent.
  - Consider denormalization when complex joins or repeated aggregations significantly slow down queries
- **Disadvantages:**
  - Introduces data redundancy, which can lead to inconsistencies.
  - Increases storage requirements.
  - Makes updates more complex, as redundant data must be synchronized.

76. How does SQL handle recursive queries?

SQL handles **recursive queries** using **Common Table Expressions** (CTEs). A recursive CTE repeatedly references itself to process hierarchical or tree-structured data.

Key Components:

- **Anchor Member:** The initial query that starts the recursion.
- **Recursive Member:** A query that references the CTE to continue building the result set.
- **Termination Condition:** Ensures that recursion stops after a certain depth or condition is met.

Example:

```
WITH RecursiveCTE (ID, ParentID, Depth) AS (
    SELECT ID, ParentID, 1 AS Depth
    FROM Categories
    WHERE ParentID IS NULL
    UNION ALL
    SELECT c.ID, c.ParentID, r.Depth + 1
    FROM Categories c
    INNER JOIN RecursiveCTE r
```

- Increases storage requirements.
- Makes updates more complex, as redundant data must be synchronized.

76. How does SQL handle recursive queries?

SQL handles **recursive queries** using **Common Table Expressions** (CTEs). A recursive CTE repeatedly references itself to process hierarchical or tree-structured data.

Key Components:

- **Anchor Member:** The initial query that starts the recursion.
- **Recursive Member:** A query that references the CTE to continue building the result set.
- **Termination Condition:** Ensures that recursion stops after a certain depth or condition is met.

Example:

```
WITH RecursiveCTE (ID, ParentID, Depth) AS (
    SELECT ID, ParentID, 1 AS Depth
    FROM Categories
    WHERE ParentID IS NULL
    UNION ALL
    SELECT c.ID, c.ParentID, r.Depth + 1
    FROM Categories c
    INNER JOIN RecursiveCTE r
    ON c.ParentID = r.ID
)
SELECT * FROM RecursiveCTE;
```

77. What are the differences between transactional and analytical queries?

1. Transactional Queries:

- Focus on individual, short-term operations such as inserts, updates, and deletes.
- Optimize for high-throughput and low-latency.
- Often used in OLTP (Online Transaction Processing) systems.

2. Analytical Queries:

- Involve complex aggregations, multi-dimensional analysis, and data transformations.
- Typically read-heavy, processing large amounts of historical or aggregated data.
- Often used in OLAP (Online Analytical Processing) systems.

3. Key Differences:

- **Transactional queries** support day-to-day operations and maintain data integrity.
- **Analytical queries** support decision-making by providing insights from large datasets

78. How can you ensure data consistency across distributed databases?

1. Use Distributed Transactions: Implement two-phase commit (2PC) to ensure all participating databases commit changes simultaneously or roll back if any part fails.

2. Implement Eventual Consistency: If strong consistency isn't required, allow data to become consistent over time. This approach is common in distributed systems where high availability is a priority.

3. Conflict Resolution Mechanisms: Use versioning, timestamps, or conflict detection rules to resolve inconsistencies.

4. Data Replication and Synchronization: Use reliable replication strategies to ensure that changes made in one database are propagated to others.

5. Regular Audits and Validation: Periodically verify that data remains consistent across databases and fix discrepancies as needed.

79. What is the purpose of the SQL PIVOT operator?

The PIVOT operator transforms rows into columns, making it easier to summarize or rearrange data for reporting.

Example:

Converting a dataset that lists monthly sales into a format that displays each month as a separate column.

```
SELECT ProductID, [2021], [2022]
FROM (
    SELECT ProductID, YEAR(SaleDate) AS SaleYear, Amount
    FROM Sales
) AS SourceTable
PIVOT (
```

made in one database are propagated to others.

5. Regular Audits and Validation: Periodically verify that data remains consistent across databases and fix discrepancies as needed.

79. What is the purpose of the SQL PIVOT operator?

The PIVOT operator transforms rows into columns, making it easier to summarize or rearrange data for reporting.

Example:

Converting a dataset that lists monthly sales into a format that displays each month as a separate column.

```
SELECT ProductID, [2021], [2022]
FROM (
    SELECT ProductID, YEAR(SaleDate) AS SaleYear, Amount
    FROM Sales
) AS Source
PIVOT (
    SUM(Amount)
    FOR SaleYear IN ([2021], [2022])
) AS PivotTable;
```

80. What is a bitmap index, and how does it differ from a B-tree index?

1. Bitmap Index:

- Represents data with bitmaps (arrays of bits) to indicate the presence or absence of a value in each row.
- Efficient for low-cardinality columns, such as “gender” or “yes/no” fields.
- Can perform fast logical operations (AND, OR, NOT) on multiple columns simultaneously.

2. B-tree Index:

- Uses a balanced tree structure to store indexed data in a sorted order.
- Suitable for high-cardinality columns (e.g., unique identifiers, large ranges of values).
- Supports range-based queries efficiently.

3. Key Difference:

- Bitmap indexes excel with low-cardinality data and complex boolean conditions.
- B-tree indexes are better for unique or high-cardinality data and range queries.

Query Based SQL Interview Questions

This section is dedicated to questions that focus on **writing** and **understanding SQL queries**. By practicing these examples, we’ll learn **how to retrieve**, **manipulate**, and **analyze data** effectively, building the **problem-solving** skills needed for **real-world scenarios**.

81. Write a query to find the second-highest salary of an employee in a table.

```
SELECT MAX(Salary) AS SecondHighestSalary
FROM Employee
WHERE Salary < (SELECT MAX(Salary) FROM Employee);
```

Explanation:

This query identifies the second-highest salary by selecting the maximum salary that is less than the overall highest salary. The subquery determines the top salary, while the outer query finds the next highest value.

82. Write a query to retrieve employees who earn more than the average salary.

```
SELECT *
FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```

Explanation:

This query fetches details of employees whose salary exceeds the average salary. The subquery calculates the average salary, and the main query filters rows based on that result.

83. Write a query to fetch the duplicate values from a column in a table.

```
SELECT ColumnName, COUNT(*)
FROM TableName
```

3. Key Difference:

- Bitmap indexes excel with low-cardinality data and complex boolean conditions.
- B-tree indexes are better for unique or high-cardinality data and range queries.

Query Based SQL Interview Questions

This section is dedicated to questions that focus on **writing** and **understanding SQL queries**. By practicing these examples, we'll learn **how to retrieve**, **manipulate**, and **analyze data** effectively, building the **problem-solving** skills needed for **real-world scenarios**.

81. Write a query to find the second-highest salary of an employee in a table.

```
SELECT MAX(Salary) AS SecondHighestSalary
FROM Employee
WHERE Salary < (SELECT MAX(Salary) FROM Employee);
```

Explanation:

This query identifies the second-highest salary by selecting the maximum salary that is less than the overall highest salary. The subquery determines the top salary, while the outer query finds the next highest value.

82. Write a query to retrieve employees who earn more than the average salary.

```
SELECT *
FROM Employee
WHERE Salary > (SELECT AVG(Salary) FROM Employee);
```

Explanation:

This query fetches details of employees whose salary exceeds the average salary. The subquery calculates the average salary, and the main query filters rows based on that result.

83. Write a query to fetch the duplicate values from a column in a table.

```
SELECT ColumnName, COUNT(*)
FROM TableName
GROUP BY ColumnName
HAVING COUNT(*) > 1;
```

Explanation:

The query uses `GROUP BY` to group identical values and `HAVING COUNT(*) > 1` to identify values that appear more than once in the specified column.

84. Write a query to find the employees who joined in the last 30 days.

```
SELECT *
FROM Employee
WHERE JoiningDate > DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

Explanation:

By comparing the `JoiningDate` to the current date minus 30 days, this query retrieves all employees who joined within the last month.

85. Write a query to fetch top 3 earning employees.

```
SELECT *
FROM Employee
ORDER BY Salary DESC
LIMIT 3;
```

Explanation:

The query sorts employees by salary in descending order and uses `LIMIT 3` to return only the top three earners.

86. Write a query to delete duplicate rows in a table without using the ROWID keyword.

```
DELETE FROM Employee
WHERE EmployeeID NOT IN (
```

Explanation:

The query sorts employees by salary in descending order and uses `LIMIT 3` to return only the top three earners.

86. Write a query to delete duplicate rows in a table without using the ROWID keyword.

```
DELETE FROM Employee
WHERE EmployeeID NOT IN (
   SELECT MIN(EmployeeID)
   FROM Employee
   GROUP BY Column1, Column2
);
```

Explanation:

This query retains only one row for each set of duplicates by keeping the row with the smallest `EmployeeID`. It identifies duplicates using `GROUP BY` and removes rows not matching the minimum ID.

87. Write a query to fetch common records from two tables.

```
SELECT *
FROM TableA
INNER JOIN TableB ON TableA.ID = TableB.ID;
```

Explanation:

An `INNER JOIN` is used to find rows present in both tables by matching a common column (in this case, `ID`).

88. Write a query to fetch employees whose names start and end with 'A'.

```
SELECT *
FROM Employee
WHERE Name LIKE 'A%' AND Name LIKE '%A';
```

Explanation:

The query uses `LIKE` with wildcard characters to filter rows where the `Name` column starts and ends with the letter 'A'.

89. Write a query to display all departments along with the number of employees in each.

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

Explanation:

By grouping employees by their `DepartmentID` and counting rows in each group, the query produces a list of departments along with the employee count.

90. Write a query to find employees who do not have managers.

```
SELECT *
FROM Employee
WHERE ManagerID IS NULL;
```

Explanation:

This query selects employees whose `ManagerID` column is `NULL`, indicating they don't report to a manager.

91. Write a query to fetch the 3rd and 4th highest salaries.

```
WITH SalaryRank AS (
    SELECT Salary, RANK() OVER (ORDER BY Salary DESC) AS Rank
    FROM Employee
)
SELECT Salary
FROM SalaryRank
WHERE Rank IN (3, 4);
```

Explanation:

This query selects employees whose `ManagerID` column is `NULL`, indicating they don't report to a manager.

91. Write a query to fetch the 3rd and 4th highest salaries.

```
WITH SalaryRank AS (
    SELECT Salary, RANK() OVER (ORDER BY Salary DESC) AS Rank
    FROM Employee
)
SELECT Salary
FROM SalaryRank
WHERE Rank IN (3, 4);
```

Explanation:

This query uses the **RANK()** window function to rank the salaries in descending order. The outer query then selects the 3rd and 4th highest salaries by filtering for those ranks.

92. Write a query to transpose rows into columns.

```
SELECT
   MAX(CASE WHEN ColumnName = 'Condition1' THEN Value END) AS Column1,
   MAX(CASE WHEN ColumnName = 'Condition2' THEN Value END) AS Column2
FROM TableName;
```

Explanation:

This query converts specific row values into columns using conditional aggregation with `CASE`. Each column's value is determined based on a condition applied to rows.

93. Write a query to fetch records updated within the last hour.

```
SELECT *
FROM TableName
WHERE UpdatedAt >= NOW() - INTERVAL 1 HOUR;
```

Explanation:

By comparing the `UpdatedAt` timestamp to the current time minus one hour, the query retrieves rows updated in the last 60 minutes.

94. Write a query to list employees in departments that have fewer than 5 employees.

```
SELECT *
FROM Employee
WHERE DepartmentID IN (
    SELECT DepartmentID
    FROM Employee
    GROUP BY DepartmentID
    HAVING COUNT(*) < 5
);
```

Explanation:

The subquery counts employees in each department, and the main query uses those results to find employees working in departments with fewer than 5 members.

95. Write a query to check if a table contains any records.

```
SELECT CASE
       WHEN EXISTS (SELECT * FROM TableName) THEN 'Has Records'
       ELSE 'No Records'
       END AS Status;
```

Explanation:

The query uses `EXISTS` to determine if any rows exist in the table, returning a status of 'Has Records' or 'No Records' based on the result.

96. Write a query to find employees whose salaries are higher than their managers.

Explanation:

The subquery counts employees in each department, and the main query uses those results to find employees working in departments with fewer than 5 members.

95. Write a query to check if a table contains any records.

```
SELECT CASE
       WHEN EXISTS (SELECT * FROM TableName) THEN 'Has Records'
       ELSE 'No Records'
       END AS Status;
```

Explanation:

The query uses `EXISTS` to determine if any rows exist in the table, returning a status of 'Has Records' or 'No Records' based on the result.

96. Write a query to find employees whose salaries are higher than their managers.

```
SELECT e.EmployeeID, e.Salary
FROM Employee e
JOIN Employee m ON e.ManagerID = m.EmployeeID
WHERE e.Salary > m.Salary;
```

Explanation:

This query joins the `Employee` table with itself to compare employee salaries to their respective managers' salaries, selecting those who earn more.

97. Write a query to fetch alternating rows from a table.

```
WITH RowNumbered AS (
    SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum
    FROM Employee
)
SELECT *
FROM RowNumbered
WHERE RowNum % 2 = 0;
```

Explanation:

This query assigns a sequential number to each row using `ROW_NUMBER()`, then selects rows where the row number is even, effectively fetching alternating rows. The `ORDER BY (SELECT NULL)` is used to avoid any specific ordering and just apply a sequential numbering.

98. Write a query to find departments with the highest average salary.

```
SELECT DepartmentID
FROM Employee
GROUP BY DepartmentID
ORDER BY AVG(Salary) DESC
LIMIT 1;
```

Explanation:

Grouping by `DepartmentID` and ordering by the average salary in descending order, the query returns the department with the highest average.

99. Write a query to fetch the nth record from a table.

```
WITH OrderedEmployees AS (
    SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum
    FROM Employee
)
SELECT *
FROM OrderedEmployees
WHERE RowNum = n;
```

Explanation:

This query uses **ROW_NUMBER()** to generate a sequential number for each row. The outer query then retrieves the row where the number matches the desired `nth` position. The approach is portable across most databases.

100. Write a query to find employees hired in the same month of any year.

98. Write a query to find departments with the highest average salary.

```
SELECT DepartmentID
FROM Employee
GROUP BY DepartmentID
ORDER BY AVG(Salary) DESC
LIMIT 1;
```

Explanation:

Grouping by `DepartmentID` and ordering by the average salary in descending order, the query returns the department with the highest average.

99. Write a query to fetch the nth record from a table.

```
WITH OrderedEmployees AS (
    SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS RowNum
    FROM Employee
)
SELECT *
FROM OrderedEmployees
WHERE RowNum = n;
```

Explanation:

This query uses **ROW_NUMBER()** to generate a sequential number for each row. The outer query then retrieves the row where the number matches the desired `nth` position. The approach is portable across most databases.

100. Write a query to find employees hired in the same month of any year.

```
SELECT *
FROM Employee
WHERE MONTH(JoiningDate) = MONTH(CURDATE());
```

Explanation:

By comparing the month of `JoiningDate` to the current month, the query selects all employees who were hired in that month regardless of the year.

Conclusion

A solid understanding of SQL is essential for anyone aspiring to work in **data analysis**, database administration, or **software development**. By reviewing and practicing these **100 SQL interview questions**, we will gain the **confidence** and **knowledge** needed to answer even the **toughest queries** during our **next interview**. Remember, **preparation** is the key because each question we master brings us **closer to securing** that **dream job** in the tech industry.

SQL Interview Questions

1. What is Pattern Matching in SQL?

SQL pattern matching provides for pattern search in data if you have no clue as to what that word should be. This kind of SQL query uses wildcards to match a string pattern, rather than writing the exact word. The LIKE operator is used in conjunction with **SQL Wildcards** to fetch the required information.

- **Using the % wildcard to perform a simple search**

The % wildcard matches zero or more characters of any type and can be used to define wildcards both before and after the pattern. Search a student in your database with first name beginning with the letter K:

```
SELECT *
FROM students
WHERE first_name LIKE 'K%'
```

- **Omitting the patterns using the NOT keyword**

Use the NOT keyword to select records that don't match the pattern. This query returns all students whose first name does not begin with K.

```
SELECT *
FROM students
WHERE first_name NOT LIKE 'K%'
```

- **Matching a pattern anywhere using the % wildcard twice**

Search for a student in the database where he/she has a K in his/her first name.

```
SELECT *
FROM students
WHERE first_name LIKE '%Q%'
```

- **Using the _ wildcard to match pattern at a specific position**

The _ wildcard matches exactly one character of any type. It can be used in conjunction with % wildcard. This query fetches all students with letter K at the third position in their first name.

```
SELECT *
FROM students
WHERE first_name LIKE '__K%'
```

- **Matching patterns for a specific length**

The _ wildcard plays an important role as a limitation when it matches exactly one character. It limits the length and position of the matched results. For example -

```
SELECT *    /* Matches first names with three or more letters */
FROM students
WHERE first_name LIKE '___%'

SELECT *    /* Matches first names with exactly four characters */
FROM students
WHERE first_name LIKE '____'
```

2. How to create empty tables with the same structure as another table?

Creating empty tables with the same structure can be done smartly by fetching the records of one table into a new table using the INTO operator while fixing a WHERE clause to be false for all records. Hence, SQL prepares the new table with a duplicate structure to accept the fetched records but since no records get fetched due to the WHERE clause in action, nothing is inserted into the new table.

```
SELECT * INTO Students_copy
FROM Students WHERE 1 = 2;
```

3. What is a Recursive Stored Procedure?

A stored procedure that calls itself until a boundary condition is reached, is called a recursive stored procedure. This recursive function helps the programmers to deploy the same set of code several times as and when required. Some SQL programming languages limit the recursion depth to prevent an infinite loop of procedure calls from causing a stack overflow, which slows down the system and may lead to system crashes.

```
DELIMITER $$      /* Set a new delimiter => $$ */
CREATE PROCEDURE calctotal( /* Create the procedure */
    IN number INT,    /* Set Input and Ouput variables */
    OUT total INT
) BEGIN
```

to accept the fetched records but since no records get fetched due to the WHERE clause in action, nothing is inserted into the new table.

```
SELECT * INTO Students_copy
FROM Students WHERE 1 = 2;
```

3. What is a Recursive Stored Procedure?

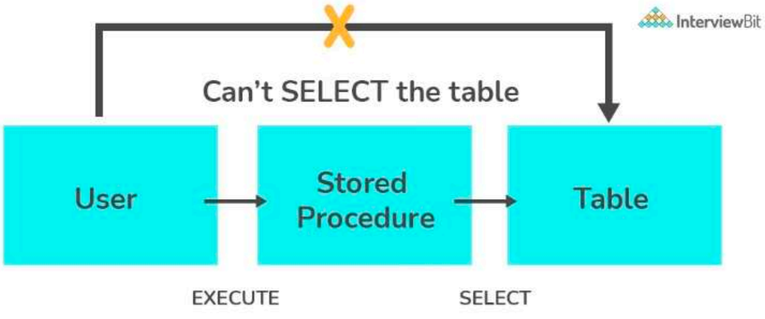
A stored procedure that calls itself until a boundary condition is reached, is called a recursive stored procedure. This recursive function helps the programmers to deploy the same set of code several times as and when required. Some SQL programming languages limit the recursion depth to prevent an infinite loop of procedure calls from causing a stack overflow, which slows down the system and may lead to system crashes.

```
DELIMITER $$      /* Set a new delimiter => $$ */
CREATE PROCEDURE calctotal( /* Create the procedure */
   IN number INT,    /* Set Input and Ouput variables */
   OUT total INT
) BEGIN
DECLARE score INT DEFAULT NULL;    /* Set the default value => "score" */
SELECT awards FROM achievements    /* Update "score" via SELECT query */
WHERE id = number INTO score;
IF score IS NULL THEN SET total = 0;    /* Termination condition */
ELSE
CALL calctotal(number+1);    /* Recursive call */
SET total = total + score;    /* Action after recursion */
END IF;
END $$      /* End of procedure */
DELIMITER ;      /* Reset the delimiter */
```

4. What is a Stored Procedure?

A stored procedure is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary. The sole disadvantage of stored procedure is that it can be executed nowhere except in the database and occupies more memory in the database server. It also provides a sense of security and functionality as users who can't access the data directly can be granted access via stored procedures.

```
DELIMITER $$
CREATE PROCEDURE FetchAllStudents()
BEGIN
SELECT *  FROM myDB.students;
END $$
DELIMITER ;
```



5. What is Collation? What are the different types of Collation Sensitivity?

Collation refers to a set of rules that determine how data is sorted and compared. Rules defining the correct character sequence are used to sort the character data. It incorporates options for specifying case sensitivity, accent marks, kana character types, and character width. Below are the different types of collation sensitivity:

- **Case sensitivity:** **A** and **a** are treated differently.
- **Accent sensitivity:** **a** and **á** are treated differently.
- **Kana sensitivity:** Japanese kana characters Hiragana and Katakana are treated differently.
- **Width sensitivity:** Same character represented in single-byte (half-width) and double-byte (full-width) are treated differently.

6. What are the differences between OLTP and OLAP?

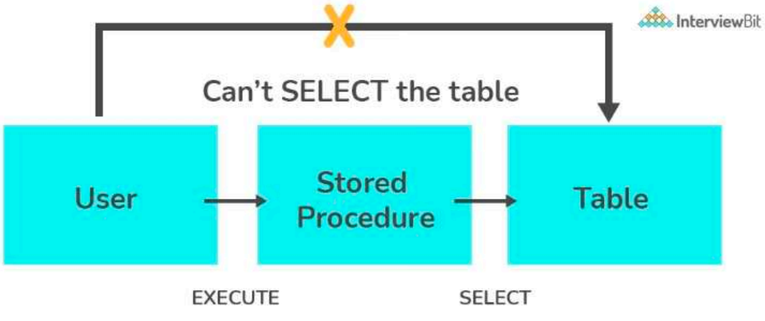
OLTP stands for **Online Transaction Processing**, is a class of software applications capable of supporting transaction-oriented programs. An important attribute of an OLTP system is its ability to maintain conc d architecture to avoid single points of failure. These systems are generally designe short transactions. Queries involved in such databases are generally simple, need

```
DELIMITER ;      /* Reset the delimiter */
```

4. What is a Stored Procedure?

A stored procedure is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary. The sole disadvantage of stored procedure is that it can be executed nowhere except in the database and occupies more memory in the database server. It also provides a sense of security and functionality as users who can't access the data directly can be granted access via stored procedures.

```
DELIMITER $$
CREATE PROCEDURE FetchAllStudents()
BEGIN
SELECT *  FROM myDB.students;
END $$
DELIMITER ;
```



5. What is Collation? What are the different types of Collation Sensitivity?

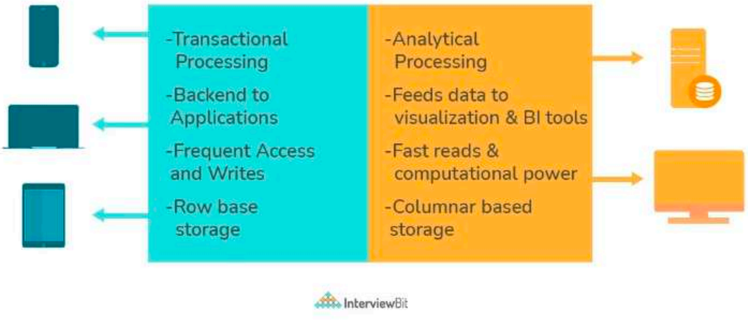
Collation refers to a set of rules that determine how data is sorted and compared. Rules defining the correct character sequence are used to sort the character data. It incorporates options for specifying case sensitivity, accent marks, kana character types, and character width. Below are the different types of collation sensitivity:

- **Case sensitivity:** **A** and **a** are treated differently.
- **Accent sensitivity:** **a** and **á** are treated differently.
- **Kana sensitivity:** Japanese kana characters Hiragana and Katakana are treated differently.
- **Width sensitivity:** Same character represented in single-byte (half-width) and double-byte (full-width) are treated differently.

6. What are the differences between OLTP and OLAP?

OLTP stands for **Online Transaction Processing**, is a class of software applications capable of supporting transaction-oriented programs. An important attribute of an OLTP system is its ability to maintain concurrency. OLTP systems often follow a decentralized architecture to avoid single points of failure. These systems are generally designed for a large audience of end-users who conduct short transactions. Queries involved in such databases are generally simple, need fast response times, and return relatively few records. A number of transactions per second acts as an effective measure for such systems.

OLAP stands for **Online Analytical Processing**, a class of software programs that are characterized by the relatively low frequency of online transactions. Queries are often too complex and involve a bunch of aggregations. For OLAP systems, the effectiveness measure relies highly on response time. Such systems are widely used for data mining or maintaining aggregated, historical data, usually in multi-dimensional schemas.

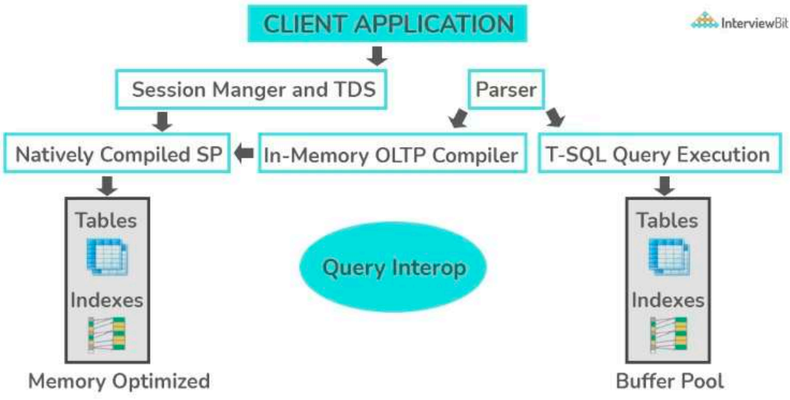


7. What is OLTP?

OLTP stands for Online Transaction Processing, is a class of software application
programs. An essential attribute of an OLTP system is its ability to maintain conc

7. What is OLTP?

OLTP stands for Online Transaction Processing, is a class of software applications capable of supporting transaction-oriented programs. An essential attribute of an OLTP system is its ability to maintain concurrency. To avoid single points of failure, OLTP systems are often decentralized. These systems are usually designed for a large number of users who conduct short transactions. Database queries are usually simple, require sub-second response times, and return relatively few records. Here is an insight into the working of an OLTP system *[Note - The figure is not important for interviews]* -



8. What is User-defined function? What are its various types?

The user-defined functions in SQL are like functions in any other programming language that accept parameters, perform complex calculations, and return a value. They are written to use the logic repetitively whenever required. There are two types of SQL user-defined functions:

- Scalar Function: As explained earlier, user-defined scalar functions return a single scalar value.
- Table-Valued Functions: User-defined table-valued functions return a table as output.
  - **Inline:** returns a table data type based on a single SELECT statement.
  - **Multi-statement:** returns a tabular result-set but, unlike inline, multiple SELECT statements can be used inside the function body.

9. What is a UNIQUE constraint?

A UNIQUE constraint ensures that all values in a column are different. This provides uniqueness for the column(s) and helps identify each row uniquely. Unlike primary key, there can be multiple unique constraints defined per table. The code syntax for UNIQUE is quite similar to that of PRIMARY KEY and can be used interchangeably.

```
CREATE TABLE Students (    /* Create table with a single field as unique */
   ID INT NOT NULL UNIQUE
   Name VARCHAR(255)
);

CREATE TABLE Students (    /* Create table with multiple fields as unique */
   ID INT NOT NULL
   LastName VARCHAR(255)
   FirstName VARCHAR(255) NOT NULL
   CONSTRAINT PK_Student
   UNIQUE (ID, FirstName)
);

ALTER TABLE Students    /* Set a column as unique */
ADD UNIQUE (ID);
ALTER TABLE Students    /* Set multiple columns as unique */
ADD CONSTRAINT PK_Student    /* Naming a unique constraint */
UNIQUE (ID, FirstName);
```

10. What is a Query?

A query is a request for data or information from a database table or combination of tables. A database query can be either a select query or an action query.

```
SELECT fname, lname     /* select query */
FROM myDb.students
WHERE student_id = 1;

UPDATE myDB.students     /* action query */
SET fname = 'Captain', lname = 'America'
WHERE student_id = 1;
```

```
ADD UNIQUE (ID);
ALTER TABLE Students    /* Set multiple columns as unique */
ADD CONSTRAINT PK_Student    /* Naming a unique constraint */
UNIQUE (ID, FirstName);
```

10. What is a Query?

A query is a request for data or information from a database table or combination of tables. A database query can be either a select query or an action query.

```
SELECT fname, lname     /* select query */
FROM myDb.students
WHERE student_id = 1;

UPDATE myDB.students     /* action query */
SET fname = 'Captain', lname = 'America'
WHERE student_id = 1;
```

11. What is Data Integrity?

Data Integrity is the assurance of accuracy and consistency of data over its entire life-cycle and is a critical aspect of the design, implementation, and usage of any system which stores, processes, or retrieves data. It also defines integrity constraints to enforce business rules on the data when it is entered into an application or a database.

12. What is the difference between Clustered and Non-clustered index?

As explained above, the differences can be broken down into three small factors -

- Clustered index modifies the way records are stored in a database based on the indexed column. A non-clustered index creates a separate entity within the table which references the original table.
- Clustered index is used for easy and speedy retrieval of data from the database, whereas, fetching records from the non-clustered index is relatively slower.
- In SQL, a table can have a single clustered index whereas it can have multiple non-clustered indexes.

13. What is an Index? Explain its different types.

A database index is a data structure that provides a quick lookup of data in a column or columns of a table. It enhances the speed of operations accessing data from a database table at the cost of additional writes and memory to maintain the index data structure.

```
CREATE INDEX index_name    /* Create Index */
ON table_name (column_1, column_2);
DROP INDEX index_name;    /* Drop Index */
```

There are different types of indexes that can be created for different purposes:

- **Unique and Non-Unique Index:**

Unique indexes are indexes that help maintain data integrity by ensuring that no two rows of data in a table have identical key values. Once a unique index has been defined for a table, uniqueness is enforced whenever keys are added or changed within the index.

```
CREATE UNIQUE INDEX myIndex
ON students (enroll_no);
```

Non-unique indexes, on the other hand, are not used to enforce constraints on the tables with which they are associated. Instead, non-unique indexes are used solely to improve query performance by maintaining a sorted order of data values that are used frequently.

- **Clustered and Non-Clustered Index:**

Clustered indexes are indexes whose order of the rows in the database corresponds to the order of the rows in the index. This is why only one clustered index can exist in a given table, whereas, multiple non-clustered indexes can exist in the table.

The only difference between clustered and non-clustered indexes is that the database manager attempts to keep the data in the database in the same order as the corresponding keys appear in the clustered index.

Clustering indexes can improve the performance of most query operations because they provide a linear-access path to data stored in the database.

14. What is a Cross-Join?

Cross join can be defined as a cartesian product of the two tables included in the join. The table after join contains the same number of rows as in the cross-product of the number of rows in the two tables. If a WHERE clause is used in cross join then the query will work like an INNER JOIN.

```
SELECT stu.name, sub.subject
FROM students AS stu
CROSS JOIN subjects AS sub;
```

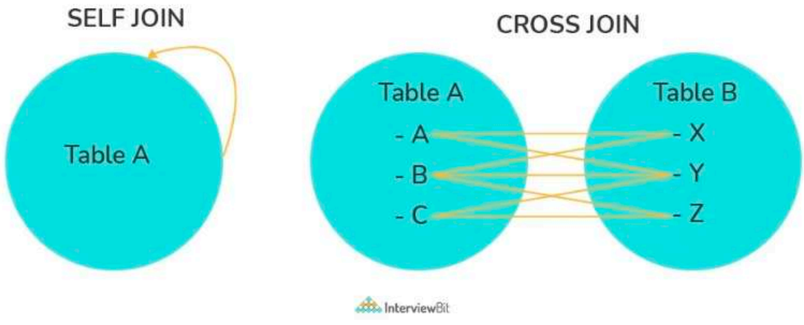
database in the same order as the corresponding keys appear in the clustered index.

Clustering indexes can improve the performance of most query operations because they provide a linear-access path to data stored in the database.

14. What is a Cross-Join?

Cross join can be defined as a cartesian product of the two tables included in the join. The table after join contains the same number of rows as in the cross-product of the number of rows in the two tables. If a WHERE clause is used in cross join then the query will work like an INNER JOIN.

```
SELECT stu.name, sub.subject
FROM students AS stu
CROSS JOIN subjects AS sub;
```



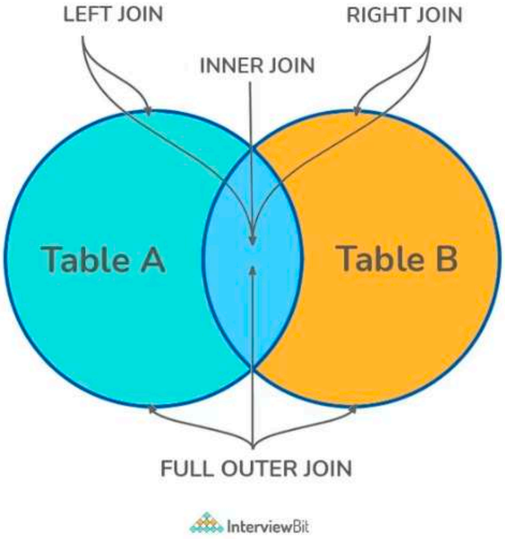
15. What is a Self-Join?

A **self JOIN** is a case of regular join where a table is joined to itself based on some relation between its own column(s). Self-join uses the INNER JOIN or LEFT JOIN clause and a table alias is used to assign different names to the table within the query.

```
SELECT A.emp_id AS "Emp_ID",A.emp_name AS "Employee",
B.emp_id AS "Sup_ID",B.emp_name AS "Supervisor"
FROM employee A, employee B
WHERE A.emp_sup = B.emp_id;
```

16. What is a Join? List its different types.

The SQL Join clause is used to combine records (rows) from two or more tables in a SQL database based on a related column between the two.



There are four different types of JOINs in SQL:

- **(INNER) JOIN:** Retrieves records that have matching values in both tables involved in the join. This is the widely used join for queries.

```
SELECT *
FROM Table_A
JOIN Table_B;
SELECT *
FROM Table_A
INNER JOIN Table_B;
```

- **LEFT (OUTER) JOIN:** Retrieves all the records/rows from the left and the matched records/rows from the right table.

```
SELECT *
FROM Table_A A
LEFT JOIN Table_B B
```

InterviewBit

There are four different types of JOINs in SQL:

- **(INNER) JOIN:** Retrieves records that have matching values in both tables involved in the join. This is the widely used join for queries.

```
SELECT *
FROM Table_A
JOIN Table_B;
SELECT *
FROM Table_A
INNER JOIN Table_B;
```

- **LEFT (OUTER) JOIN:** Retrieves all the records/rows from the left and the matched records/rows from the right table.

```
SELECT *
FROM Table_A A
LEFT JOIN Table_B B
ON A.col = B.col;
```

- **RIGHT (OUTER) JOIN:** Retrieves all the records/rows from the right and the matched records/rows from the left table.

```
SELECT *
FROM Table_A A
RIGHT JOIN Table_B B
ON A.col = B.col;
```

- **FULL (OUTER) JOIN:** Retrieves all the records where there is a match in either the left or right table.

```
SELECT *
FROM Table_A A
FULL JOIN Table_B B
ON A.col = B.col;
```

17. What is a Foreign Key?

A FOREIGN KEY comprises of single or collection of fields in a table that essentially refers to the PRIMARY KEY in another table. Foreign key constraint ensures referential integrity in the relation between two tables.
The table with the foreign key constraint is labeled as the child table, and the table containing the candidate key is labeled as the referenced or parent table.

```
CREATE TABLE Students (    /* Create table with foreign key - Way 1 */
   ID INT NOT NULL
   Name VARCHAR(255)
   LibraryID INT
   PRIMARY KEY (ID)
   FOREIGN KEY (Library_ID) REFERENCES Library(LibraryID)
);

CREATE TABLE Students (    /* Create table with foreign key - Way 2 */
   ID INT NOT NULL PRIMARY KEY
   Name VARCHAR(255)
   LibraryID INT FOREIGN KEY (Library_ID) REFERENCES Library(LibraryID)
);

ALTER TABLE Students    /* Add a new foreign key */
ADD FOREIGN KEY (LibraryID)
REFERENCES Library (LibraryID);
```

18. What is a Subquery? What are its types?

A subquery is a query within another query, also known as a **nested query** or **inner query**. It is used to restrict or enhance the data to be queried by the main query, thus restricting or enhancing the output of the main query respectively. For example, here we fetch the contact information for students who have enrolled for the maths subject:

```
SELECT name, email, mob, address
FROM myDb.contacts
WHERE roll_no IN (
 SELECT roll_no
 FROM myDb.students
 WHERE subject = 'Maths');
```

There are two types of subquery - **Correlated** and **Non-Correlated**.

- A **correlated** subquery cannot be considered as an independent query, but it can refer to the column in a table listed in the FROM of the main query.
- A **non-correlated** subquery can be considered as an independent query and the output of the subquery is substituted in the main query.

A subquery is a query within another query, also known as a **nested query** or **inner query**. It is used to restrict or enhance the data to be queried by the main query, thus restricting or enhancing the output of the main query respectively. For example, here we fetch the contact information for students who have enrolled for the maths subject:

```
SELECT name, email, mob, address
FROM myDb.contacts
WHERE roll_no IN (
 SELECT roll_no
 FROM myDb.students
 WHERE subject = 'Maths');
```

There are two types of subquery - **Correlated** and **Non-Correlated**.

- A **correlated** subquery cannot be considered as an independent query, but it can refer to the column in a table listed in the FROM of the main query.
- A **non-correlated** subquery can be considered as an independent query and the output of the subquery is substituted in the main query.

19. What is a Primary Key?

The PRIMARY KEY constraint uniquely identifies each row in a table. It must contain UNIQUE values and has an implicit NOT NULL constraint.
A table in SQL is strictly restricted to have one and only one primary key, which is comprised of single or multiple fields (columns).

```
CREATE TABLE Students (    /* Create table with a single field as primary key */
   ID INT NOT NULL
   Name VARCHAR(255)
   PRIMARY KEY (ID)
);

CREATE TABLE Students (    /* Create table with multiple fields as primary key */
   ID INT NOT NULL
   LastName VARCHAR(255)
   FirstName VARCHAR(255) NOT NULL,
   CONSTRAINT PK_Student
   PRIMARY KEY (ID, FirstName)
);

ALTER TABLE Students    /* Set a column as primary key */
ADD PRIMARY KEY (ID);
ALTER TABLE Students    /* Set multiple columns as primary key */
ADD CONSTRAINT PK_Student    /*Naming a Primary Key*/
PRIMARY KEY (ID, FirstName);
```


20. What are Constraints in SQL?

Constraints are used to specify the rules concerning data in the table. It can be applied for single or multiple fields in an SQL table during the creation of the table or after creating using the ALTER TABLE command. The constraints are:

- **NOT NULL** - Restricts NULL value from being inserted into a column.
- **CHECK** - Verifies that all values in a field satisfy a condition.
- **DEFAULT** - Automatically assigns a default value if no value has been specified for the field.
- **UNIQUE** - Ensures unique values to be inserted into the field.
- **INDEX** - Indexes a field providing faster retrieval of records.
- **PRIMARY KEY** - Uniquely identifies each record in a table.
- **FOREIGN KEY** - Ensures referential integrity for a record in another table.

21. What are Tables and Fields?

A table is an organized collection of data stored in the form of rows and columns. Columns can be categorized as vertical and rows as horizontal. The columns in a table are called fields while the rows can be referred to as records.

22. What is the difference between SQL and MySQL?

- **INDEX** - Indexes a field providing faster retrieval of records.
- **PRIMARY KEY** - Uniquely identifies each record in a table.
- **FOREIGN KEY** - Ensures referential integrity for a record in another table.

21. What are Tables and Fields?

A table is an organized collection of data stored in the form of rows and columns. Columns can be categorized as vertical and rows as horizontal. The columns in a table are called fields while the rows can be referred to as records.

22. What is the difference between SQL and MySQL?

SQL is a standard language for retrieving and manipulating structured databases. On the contrary, MySQL is a relational database management system, like SQL Server, Oracle or IBM DB2, that is used to manage SQL databases.

23. What is SQL?

SQL stands for Structured Query Language. It is the standard language for relational database management systems. It is especially useful in handling organized data comprised of entities (variables) and relations between different entities of the data.

24. What is RDBMS? How is it different from DBMS?

RDBMS stands for Relational Database Management System. The key difference here, compared to DBMS, is that RDBMS stores data in the form of a collection of tables, and relations can be defined between the common fields of these tables. Most modern database management systems like MySQL, Microsoft SQL Server, Oracle, IBM DB2, and Amazon Redshift are based on RDBMS.

25. What is DBMS?

DBMS stands for Database Management System. DBMS is a system software responsible for the creation, retrieval, updation, and management of the database. It ensures that our data is consistent, organized, and is easily accessible by serving as an interface between the database and its end-users or application software.

26. What is Database?

A database is an organized collection of data, stored and retrieved digitally from a remote or local computer system. Databases can be vast and complex, and such databases are developed using fixed design and modeling approaches.

27. What is the SELECT statement?

SELECT operator in SQL is used to select data from a database. The data returned is stored in a result table, called the result-set.

```
SELECT * FROM myDB.students;
```

28. What are some common clauses used with SELECT query in SQL?

Some common SQL clauses used in conjuction with a SELECT query are as follo

be vast and complex, and such databases are developed using fixed design and modeling approaches.

27. What is the SELECT statement?

SELECT operator in SQL is used to select data from a database. The data returned is stored in a result table, called the result-set.

```
SELECT * FROM myDB.students;
```

28. What are some common clauses used with SELECT query in SQL?

Some common SQL clauses used in conjuction with a SELECT query are as follows:

- **WHERE** clause in SQL is used to filter records that are necessary, based on specific conditions.
- **ORDER BY** clause in SQL is used to sort the records based on some field(s) in ascending (**ASC**) or descending order (**DESC**).

```
SELECT *
FROM myDB.students
WHERE graduation_year = 2019
ORDER BY studentID DESC;
```

- **GROUP BY** clause in SQL is used to group records with identical data and can be used in conjunction with some aggregation functions to produce summarized results from the database.
- **HAVING** clause in SQL is used to filter records in combination with the GROUP BY clause. It is different from WHERE, since the WHERE clause cannot filter aggregated records.

```
SELECT COUNT(studentId), country
FROM myDB.students
WHERE country != "INDIA"
GROUP BY country
HAVING COUNT(studentID) > 5;
```

29. What are UNION, MINUS and INTERSECT commands?

The **UNION** operator combines and returns the result-set retrieved by two or more SELECT statements.
The **MINUS** operator in SQL is used to remove duplicates from the result-set obtained by the second SELECT query from the result-set obtained by the first SELECT query and then return the filtered results from the first.
The **INTERSECT** clause in SQL combines the result-set fetched by the two SELECT statements where records from one match the other and then returns this intersection of result-sets.

Certain conditions need to be met before executing either of the above statements in SQL -

- Each SELECT statement within the clause must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement should necessarily have the same order

```
SELECT name FROM Students    /* Fetch the union of queries */
UNION
SELECT name FROM Contacts;
SELECT name FROM Students    /* Fetch the union of queries with duplicates*/
UNION ALL
SELECT name FROM Contacts;

SELECT name FROM Students    /* Fetch names from students */
MINUS      /* that aren't present in contacts */
SELECT name FROM Contacts;

SELECT name FROM Students    /* Fetch names from students */
INTERSECT     /* that are present in contacts as well */
SELECT name FROM Contacts;
```

30. What is Cursor? How to use a Cursor?

A database cursor is a control structure that allows for the traversal of records in a database. Cursors, in addition, facilitates processing after traversal, such as retrieval, addition, and deletion of database records. They can be viewed as a pointer to one row in a set of rows.

Working with SQL Cursor:

1. **DECLARE** a cursor after any variable declaration. The cursor declaration must always be associated with a SELECT Statement.
2. Open cursor to initialize the result set. The **OPEN** statement must be called before fetching rows from the result set.
3. **FETCH** statement to retrieve and move to the next row in the result set.
4. Call the **CLOSE** statement to deactivate the cursor.
5. Finally use the **DEALLOCATE** statement to delete the cursor definition and release the associated resources.

```
DECLARE @name VARCHAR(50)    /* Declare All Required Variables */
DECLARE db_cursor CURSOR FOR    /* Declare Cursor Name*/
SELECT name
FROM myDB.students
WHERE parent_name IN ('Sara', 'Ansh')
OPEN db_cursor    /* Open cursor and Fetch data into @name */
FETCH next
FROM db_cursor
INTO @name
CLOSE db_cursor    /* Close the cursor and deallocate the resources */
DEALLOCATE db_cursor
```

31. What are Entities and Relationships?

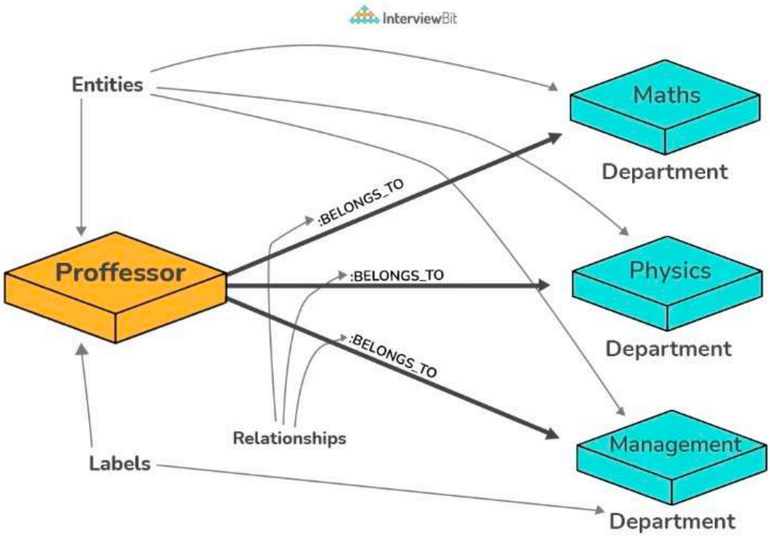
4. Call the **CLOSE** statement to deactivate the cursor.
5. Finally use the **DEALLOCATE** statement to delete the cursor definition and release the associated resources.

```
DECLARE @name VARCHAR(50)    /* Declare All Required Variables */
DECLARE db_cursor CURSOR FOR    /* Declare Cursor Name*/
SELECT name
FROM myDB.students
WHERE parent_name IN ('Sara', 'Ansh')
OPEN db_cursor    /* Open cursor and Fetch data into @name */
FETCH next
FROM db_cursor
INTO @name
CLOSE db_cursor    /* Close the cursor and deallocate the resources */
DEALLOCATE db_cursor
```

31. What are Entities and Relationships?

Entity: An entity can be a real-world object, either tangible or intangible, that can be easily identifiable. For example, in a college database, students, professors, workers, departments, and projects can be referred to as entities. Each entity has some associated properties that provide it an identity.

Relationships: Relations or links between entities that have something to do with each other. For example - The employee's table in a company's database can be associated with the salary table in the same database.



32. List the different types of relationships in SQL.

- **One-to-One** - This can be defined as the relationship between two tables where each record in one table is associated with the maximum of one record in the other table.
- **One-to-Many & Many-to-One** - This is the most commonly used relationship where a record in a table is associated with multiple records in the other table.
- **Many-to-Many** - This is used in cases when multiple instances on both sides are needed for defining a relationship.
- **Self-Referencing Relationships** - This is used when a table needs to define a relationship with itself.

33. What is an Alias in SQL?

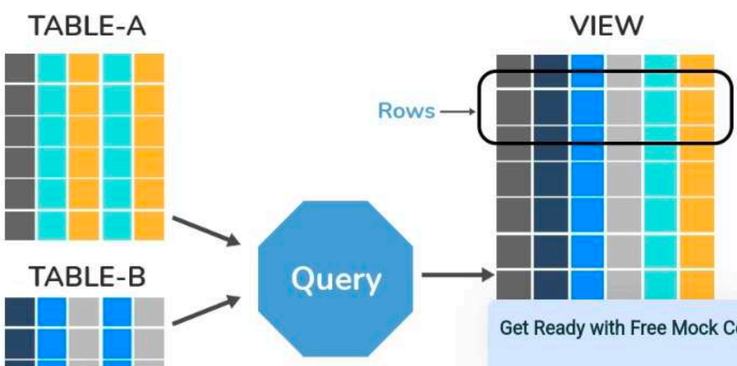
An alias is a feature of SQL that is supported by most, if not all, RDBMSs. It is a temporary name assigned to the table or table column for the purpose of a particular SQL query. In addition, aliasing can be employed as an obfuscation technique to secure the real names of database fields. A table alias is also called a correlation name.

An alias is represented explicitly by the AS keyword but in some cases, the same can be performed without it as well. Nevertheless, using the AS keyword is always a good practice.

```
SELECT A.emp_name AS "Employee"    /* Alias using AS keyword */
B.emp_name AS "Supervisor"
FROM employee A, employee B    /* Alias without AS keyword */
WHERE A.emp_sup = B.emp_id;
```

34. What is a View?

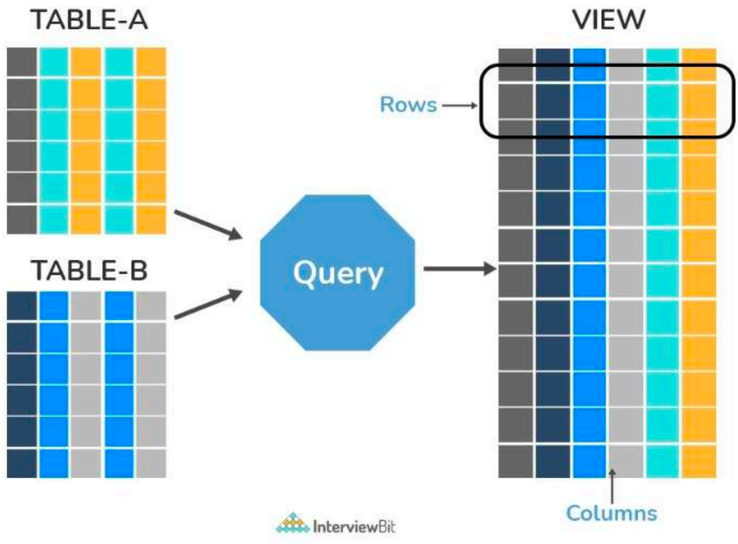
A view in SQL is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.



```
B.emp_name AS "Supervisor"
FROM employee A, employee B    /* Alias without AS keyword */
WHERE A.emp_sup = B.emp_id;
```

34. What is a View?

A view in SQL is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.



35. What is Normalization?

Normalization represents the way of organizing structured data in the database efficiently. It includes the creation of tables, establishing relationships between them, and defining rules for those relationships. Inconsistency and redundancy can be kept in check based on these rules, hence, adding flexibility to the database.

36. What is Denormalization?

Denormalization is the inverse process of normalization, where the normalized schema is converted into a schema that has redundant information. The performance is improved by using redundancy and keeping the redundant data consistent. The reason for performing denormalization is the overheads produced in the query processor by an over-normalized structure.

37. What are the various forms of Normalization?

Normal Forms are used to eliminate or reduce redundancy in database tables. The different forms are as follows:

- **First Normal Form:**
  A relation is in first normal form if every attribute in that relation is a **single-valued attribute**. If a relation contains a composite or multi-valued attribute, it violates the first normal form. Let's consider the following **students** table. Each student in the table, has a name, his/her address, and the books they issued from the public library -

Students Table

Student	Address	Books Issued	Salutation
Sara	Amanora Park Town 94	Until the Day I Die (Emily Carpenter), Inception (Christopher Nolan)	Ms.
Ansh	62nd Sector A-10	The Alchemist (Paulo Coelho), Inferno (Dan Brown)	Mr.
Sara	24th Street Park Avenue	Beautiful Bad (Annie Ward), Woman 99 (Greer Macallister)	Mrs.
Ansh	Windsor Street 777	Dracula (Bram Stoker)	Mr.

As we can observe, the Books Issued field has more than one value per record, and to convert it into 1NF, this has to be resolved into separate individual records for each book issued. Check the following table in 1NF form -

Students Table (1st Normal Form)

Student	Address	Books Issued	Salutation
Sara	Amanora Park Town 94	Until the Day I Die (Emily Carpenter)	Ms.
Sara	Amanora Park Town 94	Inception (Christopher Nolan)	Ms.
Ansh	62nd Sector A-10	The Alchemist (Paulo Coelho)	Mr.
Ansh	62nd Sector A-10	Inferno (Dan Brown)	Mr.
Sara	24th Street Park Avenue	Beautiful Bad (Annie Ward)	Mrs.
Sara	24th Street Park Avenue	Woman 99 (Greer Macallister)	Mrs.
Ansh	Windsor Street 777	Dracula (Bram Stoker)	Mr.

- **Second Normal Form:**

A relation is in second normal form if it satisfies the conditions for the first normal form and does not contain any partial dependency. A relation in 2NF has **no partial dependency**, i.e., it has no non-prime
candidate key of the table. Often, specifying a single column Primary Key is the s

Sara	24th Street Park Avenue	Beautiful Bad (Annie Ward)	Mrs.
Sara	24th Street Park Avenue	Woman 99 (Greer Macallister)	Mrs.
Ansh	Windsor Street 777	Dracula (Bram Stoker)	Mr.

- **Second Normal Form:**

A relation is in second normal form if it satisfies the conditions for the first normal form and does not contain any partial dependency. A relation in 2NF has **no partial dependency**, i.e., it has no non-prime attribute that depends on any proper subset of any candidate key of the table. Often, specifying a single column Primary Key is the solution to the problem. Examples -

Example 1 - Consider the above example. As we can observe, the Students Table in the 1NF form has a candidate key in the form of [Student, Address] that can uniquely identify all records in the table. The field Books Issued (non-prime attribute) depends partially on the Student field. Hence, the table is not in 2NF. To convert it into the 2nd Normal Form, we will partition the tables into two while specifying a new ***Primary Key*** attribute to identify the individual records in the Students table. The ***Foreign Key*** constraint will be set on the other table to ensure referential integrity.

Students Table (2nd Normal Form)

Student_ID	Student	Address	Salutation
1	Sara	Amanora Park Town 94	Ms.
2	Ansh	62nd Sector A-10	Mr.
3	Sara	24th Street Park Avenue	Mrs.
4	Ansh	Windsor Street 777	Mr.

Books Table (2nd Normal Form)

Student_ID	Book Issued
1	Until the Day I Die (Emily Carpenter)
1	Inception (Christopher Nolan)
2	The Alchemist (Paulo Coelho)
2	Inferno (Dan Brown)
3	Beautiful Bad (Annie Ward)
3	Woman 99 (Greer Macallister)
4	Dracula (Bram Stoker)

Example 2 - Consider the following dependencies in relation to R(W,X,Y,Z)

```
WX -> Y    [W and X together determine Y]
XY -> Z    [X and Y together determine Z]
```

Here, WX is the only candidate key and there is no partial dependency, i.e., any proper subset of WX doesn't determine any non-prime attribute in the relation.

- **Third Normal Form**

A relation is said to be in the third normal form, if it satisfies the conditions for the second normal form and there is **no transitive dependency** between the non-prime attributes, i.e., all non-prime attributes are determined only by the candidate keys of the relation and not by any other non-prime attribute.

Example 1 - Consider the Students Table in the above example. As we can observe, the Students Table in the 2NF form has a single candidate key Student_ID (primary key) that can uniquely identify all records in the table. The field Salutation (non-prime attribute), however, depends on the Student Field rather than the candidate key. Hence, the table is not in 3NF. To convert it into the 3rd Normal Form, we will once again partition the tables into two while specifying a new ***Foreign Key*** constraint to identify the salutations for individual records in the Students table. The ***Primary Key*** constraint for the same will be set on the Salutations table to identify each record uniquely.

Students Table (3rd Normal Form)

Student_ID	Student	Address	Salutation_ID
1	Sara	Amanora Park Town 94	1
2	Ansh	62nd Sector A-10	2
3	Sara	24th Street Park Avenue	3
4	Ansh	Windsor Street 777	1

Books Table (3rd Normal Form)

Student_ID	Book Issued
1	Until the Day I Die (Emily Carpenter)
1	Inception (Christopher Nolan)
2	The Alchemist (Paulo Coelho)
2	Inferno (Dan Brown)
3	Beautiful Bad (Annie Ward)
3	Woman 99 (Greer Maca

Form, we will once again partition the tables into two while specifying a new ***Foreign Key*** constraint to identify the salutations for individual records in the Students table. The ***Primary Key*** constraint for the same will be set on the Salutations table to identify each record uniquely.

Students Table (3rd Normal Form)

Student_ID	Student	Address	Salutation_ID
1	Sara	Amanora Park Town 94	1
2	Ansh	62nd Sector A-10	2
3	Sara	24th Street Park Avenue	3
4	Ansh	Windsor Street 777	1

Books Table (3rd Normal Form)

Student_ID	Book Issued
1	Until the Day I Die (Emily Carpenter)
1	Inception (Christopher Nolan)
2	The Alchemist (Paulo Coelho)
2	Inferno (Dan Brown)
3	Beautiful Bad (Annie Ward)
3	Woman 99 (Greer Macallister)
4	Dracula (Bram Stoker)

Salutations Table (3rd Normal Form)

Salutation_ID	Salutation
1	Ms.
2	Mr.
3	Mrs.

Example 2 - Consider the following dependencies in relation to R(P,Q,R,S,T)

```
P -> QR     [P together determine C]
RS -> T     [B and C together determine D]
Q -> S
T -> P
```

For the above relation to exist in 3NF, all possible candidate keys in the above relation should be {P, RS, QR, T}.

- **Boyce-Codd Normal Form**

A relation is in Boyce-Codd Normal Form if satisfies the conditions for third normal form and for every functional dependency, Left-Hand-Side is super key. In other words, a relation in BCNF has non-trivial functional dependencies in form X –> Y, such that X is always a super key. For example - In the above example, Student_ID serves as the sole unique identifier for the Students Table and Salutation_ID for the Salutations Table, thus these tables exist in BCNF. The same cannot be said for the Books Table and there can be several books with common Book Names and the same Student_ID.

38. What are the TRUNCATE, DELETE and DROP statements?

DELETE statement is used to delete rows from a table.

```
DELETE FROM Candidates
WHERE CandidateId > 1000;
```

TRUNCATE command is used to delete all the rows from the table and free the space containing the table.

```
TRUNCATE TABLE Candidates;
```

DROP command is used to remove an object from the database. If you drop a table, all the rows in the table are deleted and the table structure is removed from the database.

```
DROP TABLE Candidates;
```

39. What is the difference between DROP and TRUNCATE statements?

If a table is dropped, all things associated with the tables are dropped as well. This includes - the relationships defined on the table with other tables, the integrity checks and constraints, access privileges and other grants that the table has. To create and use the table again in its original form, all these relations, checks, constraints, privileges and relationships need to be redefined. However, if a table is truncated, none of the above problems exist and the table retains its original structure.

40. What is the difference between DELETE and TRUNCATE statements?

The **TRUNCATE** command is used to delete all the rows from the table and free the space containing the table.
The **DELETE** command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.

41. What are Aggregate and Scalar functions?

An aggregate function performs operations on a collection of values to return a s
used with the GROUP BY and HAVING clauses of the SELECT statement. Followin

The **TRUNCATE** command is used to delete all the rows from the table and free the space containing the table.
The **DELETE** command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.

41. What are Aggregate and Scalar functions?

An aggregate function performs operations on a collection of values to return a single scalar value. Aggregate functions are often used with the GROUP BY and HAVING clauses of the SELECT statement. Following are the widely used SQL aggregate functions:

- **AVG()** - Calculates the mean of a collection of values.
- **COUNT()** - Counts the total number of records in a specific table or view.
- **MIN()** - Calculates the minimum of a collection of values.
- **MAX()** - Calculates the maximum of a collection of values.
- **SUM()** - Calculates the sum of a collection of values.
- **FIRST()** - Fetches the first element in a collection of values.
- **LAST()** - Fetches the last element in a collection of values.

Note: All aggregate functions described above ignore NULL values except for the COUNT function.

A scalar function returns a single value based on the input value. Following are the widely used SQL scalar functions:

- **LEN()** - Calculates the total length of the given field (column).
- **UCASE()** - Converts a collection of string values to uppercase characters.
- **LCASE()** - Converts a collection of string values to lowercase characters.
- **MID()** - Extracts substrings from a collection of string values in a table.
- **CONCAT()** - Concatenates two or more strings.
- **RAND()** - Generates a random collection of numbers of a given length.
- **ROUND()** - Calculates the round-off integer value for a numeric field (or decimal point values).
- **NOW()** - Returns the current date & time.
- **FORMAT()** - Sets the format to display a collection of values.

PostgreSQL Interview Questions

42. What is PostgreSQL?

PostgreSQL was first called Postgres and was developed by a team led by Computer Science Professor Michael Stonebraker in 1986. It was developed to help developers build enterprise-level applications by upholding data integrity by making systems fault-tolerant. PostgreSQL is therefore an enterprise-level, flexible, robust, open-source, and object-relational DBMS that supports flexible workloads along with handling concurrent users. It has been consistently supported by the global developer community. Due to its fault-tolerant nature, PostgreSQL has gained widespread popularity among developers.

43. What is the capacity of a table in PostgreSQL?

The maximum size of PostgreSQL is 32TB.

44. What is the importance of the TRUNCATE statement?

`TRUNCATE TABLE name_of_table` statement removes the data efficiently and quickly from the table.
The truncate statement can also be used to reset values of the identity columns along with data cleanup as shown below:

```
TRUNCATE TABLE name_of_table
RESTART IDENTITY;
```

We can also use the statement for removing data from multiple tables all at once by mentioning the table names separated by comma as shown below:

```
TRUNCATE TABLE
   table_1,
   table_2,
   table_3;
```

45. Define tokens in PostgreSQL?

A token in PostgreSQL is either a keyword, identifier, literal, constant, quotes identifier, or any symbol that has a distinctive personality. They may or may not be separated using a space, newline or a tab. If the tokens are keywords, they are usually commands with useful meanings. Tokens are known as building blocks of any PostgreSQL code.

46. What are partitioned tables called in PostgreSQL?

Partitioned tables are logical structures that are used for dividing large tables into smaller structures that are called partitions. This approach is used for effectively increasing the query performance while dealing with large database tables. To create a partition, a key called partition key which is usually a table column or an expression, and a partitioning method needs to be defined. There are three types of inbuilt partitioning methods provided by Postgres:

- **Range Partitioning**: This method is done by partitioning based on a range of values. This method is most commonly used upon date fields to get monthly, weekly or yearly data. In the case of corner cases like value belonging to the end of the range, for example: if the range of partition 1 is 10-20 and the range of partition 2 is 20-30, and the given value is 10, then 10 belongs to the second partition and not the first.
- **List Partitioning:** This method is used to partition based on a list of known values. Most commonly used when we have a key with a categorical value. For example, getting sales data based on regions divided as countries, cities, or states.
- **Hash Partitioning:** This method utilizes a hash function upon the partition key. This is done when there are no specific requirements for data division and is used to access data individually. For example, you want to access data based on a specific product, then using hash partition would result in the dataset that we require

A token in PostgreSQL is either a keyword, identifier, literal, constant, quotes identifier, or any symbol that has a distinctive personality. They may or may not be separated using a space, newline or a tab. If the tokens are keywords, they are usually commands with useful meanings. Tokens are known as building blocks of any PostgreSQL code.

46. What are partitioned tables called in PostgreSQL?

Partitioned tables are logical structures that are used for dividing large tables into smaller structures that are called partitions. This approach is used for effectively increasing the query performance while dealing with large database tables. To create a partition, a key called partition key which is usually a table column or an expression, and a partitioning method needs to be defined. There are three types of inbuilt partitioning methods provided by Postgres:

- **Range Partitioning**: This method is done by partitioning based on a range of values. This method is most commonly used upon date fields to get monthly, weekly or yearly data. In the case of corner cases like value belonging to the end of the range, for example: if the range of partition 1 is 10-20 and the range of partition 2 is 20-30, and the given value is 10, then 10 belongs to the second partition and not the first.
- **List Partitioning:** This method is used to partition based on a list of known values. Most commonly used when we have a key with a categorical value. For example, getting sales data based on regions divided as countries, cities, or states.
- **Hash Partitioning:** This method utilizes a hash function upon the partition key. This is done when there are no specific requirements for data division and is used to access data individually. For example, you want to access data based on a specific product, then using hash partition would result in the dataset that we require.

The type of partition key and the type of method used for partitioning determines how positive the performance and the level of manageability of the partitioned table are.

47. How can we start, restart and stop the PostgreSQL server?

- To **start** the PostgreSQL server, we run:

```
service postgresql start
```

- Once the **server** is successfully started, we get the below message:

```
Starting PostgreSQL: ok
```

- To **restart** the PostgreSQL server, we run:

```
service postgresql restart
```

Once the server is successfully restarted, we get the message:

```
Restarting PostgreSQL: server stopped
ok
```

- To **stop** the server, we run the command:

```
service postgresql stop
```

Once stopped successfully, we get the message:

```
Stopping PostgreSQL: server stopped
ok
```

48. What is the command used for creating a database in PostgreSQL?

The first step of using PostgreSQL is to create a database. This is done by using the createdb command as shown below: `createdb db_name`
After running the above command, if the database creation was successful, then the below message is shown:

```
CREATE DATABASE
```

49. How will you change the datatype of a column?

This can be done by using the ALTER TABLE statement as shown below:

Syntax:

```
ALTER TABLE tname
ALTER COLUMN col_name [SET DATA] TYPE new_data_type;
```

50. How do you define Indexes in PostgreSQL?

Indexes are the inbuilt functions in PostgreSQL which are used by the queries to perform search more efficiently on a table in the database. Consider that you have a table with thousands of records and you have the below query that only a few records can satisfy the condition, then it will take a lot of time to search and return those rows that abide by this condition as the engine has to perform the search operation on every single to check this condition. This is undoubtedly inefficient for a system dealing with huge data. Now if this system had an index on the column where we are applying search, it can use an efficient method for identifying matching rows by walking through only a few levels. This is called indexing.

```
Select * from some_table where table_col=120
```

51. Define sequence.

A sequence is a schema-bound, user-defined object which aids to generate a sequence of integers. This is most commonly used to generate values to identity columns in a table. We can create a sequence by using the `CREATE SEQUENCE` statement as shown below:

```
CREATE SEQUENCE serial_num START 100;
```

To get the next number 101 from the sequence, we use the nextval() method as s

```
SELECT nextval('serial_num');
```

```
Select * from some_table where table_col=120
```

51. Define sequence.

A sequence is a schema-bound, user-defined object which aids to generate a sequence of integers. This is most commonly used to generate values to identity columns in a table. We can create a sequence by using the `CREATE SEQUENCE` statement as shown below:

```
CREATE SEQUENCE serial_num START 100;
```

To get the next number 101 from the sequence, we use the nextval() method as shown below:

```
SELECT nextval('serial_num');
```

We can also use this sequence while inserting new records using the INSERT command:

```
INSERT INTO ib_table_name VALUES (nextval('serial_num'), 'interviewbit');
```

52. What are string constants in PostgreSQL?

They are character sequences bound within single quotes. These are using during data insertion or updation to characters in the database.
There are special string constants that are quoted in dollars. Syntax: `$tag$<string_constant>$tag$` The tag in the constant is optional and when we are not specifying the tag, the constant is called a double-dollar string literal.

53. How can you get a list of all databases in PostgreSQL?

This can be done by using the command `\l` -> backslash followed by the lower-case letter L.

54. How can you delete a database in PostgreSQL?

This can be done by using the DROP DATABASE command as shown in the syntax below:

```
DROP DATABASE database_name;
```

If the database has been deleted successfully, then the following message would be shown:

```
DROP DATABASE
```

55. What are ACID properties? Is PostgreSQL compliant with ACID?

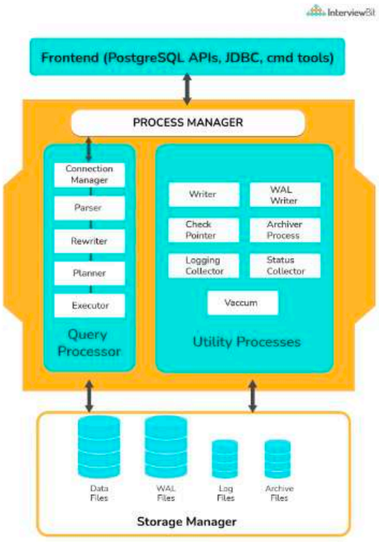
ACID stands for Atomicity, Consistency, Isolation, Durability. They are database transaction properties which are used for guaranteeing data validity in case of errors and failures.

- **Atomicity**: This property ensures that the transaction is completed in all-or-nothing way.
- **Consistency**: This ensures that updates made to the database is valid and follows rules and restrictions.
- **Isolation**: This property ensures integrity of transaction that are visible to all other transactions.
- **Durability**: This property ensures that the committed transactions are stored permanently in the database.

PostgreSQL is compliant with ACID properties.

56. Can you explain the architecture of PostgreSQL?

- The architecture of PostgreSQL follows the client-server model.
- The server side comprises of background process manager, query processer, utilities and shared memory space which work together to build PostgreSQL's instance that has access to the data. The client application does the task of connecting to this instance and requests data processing to the services. The client can either be GUI (Graphical User Interface) or a web application. The most commonly used client for PostgreSQL is pgAdmin.



57. What do you understand by multi-version concurrency control?

MVCC or Multi-version concurrency control is used for avoiding unnecessary database locks when 2 or more requests tries to access or modify the data at the same time. This ensures that the time lag for a user to log in to the database is avoided. The transactions are recorded when anyone tries to access the content.

For more information regarding this, you can refer here.

57. What do you understand by multi-version concurrency control?

MVCC or Multi-version concurrency control is used for avoiding unnecessary database locks when 2 or more requests tries to access or modify the data at the same time. This ensures that the time lag for a user to log in to the database is avoided. The transactions are recorded when anyone tries to access the content.

For more information regarding this, you can refer here.

58. What do you understand by command enable-debug?

The command enable-debug is used for enabling the compilation of all libraries and applications. When this is enabled, the system processes get hindered and generally also increases the size of the binary file. Hence, it is not recommended to switch this on in the production environment. This is most commonly used by developers to debug the bugs in their scripts and help them spot the issues. For more information regarding how to debug, you can refer here.

59. How do you check the rows affected as part of previous transactions?

SQL standards state that the following three phenomena should be prevented whilst concurrent transactions. SQL standards define 4 levels of transaction isolations to deal with these phenomena.

- **Dirty reads**: If a transaction reads data that is written due to concurrent uncommitted transaction, these reads are called dirty reads.
- **Phantom reads**: This occurs when two same queries when executed separately return different rows. For example, if transaction A retrieves some set of rows matching search criteria. Assume another transaction B retrieves new rows in addition to the rows obtained earlier for the same search criteria. The results are different.
- **Non-repeatable reads**: This occurs when a transaction tries to read the same row multiple times and gets different values each time due to concurrency. This happens when another transaction updates that data and our current transaction fetches that updated data, resulting in different values.

To tackle these, there are 4 standard isolation levels defined by SQL standards. They are as follows:

- **Read Uncommitted** – The lowest level of the isolations. Here, the transactions are not isolated and can read data that are not committed by other transactions resulting in dirty reads.
- **Read Committed** – This level ensures that the data read is committed at any instant of read time. Hence, dirty reads are avoided here. This level makes use of read/write lock on the current rows which prevents read/write/update/delete of that row when the current transaction is being operated on.
- **Repeatable Read** – The most restrictive level of isolation. This holds read and write locks for all rows it operates on. Due to this, non-repeatable reads are avoided as other transactions cannot read, write, update or delete the rows.
- **Serializable** – The highest of all isolation levels. This guarantees that the execution is serializable where execution of any concurrent operations are guaranteed to be appeared as executing serially.

The following table clearly explains which type of unwanted reads the levels avoid:

Isolation levels	Dirty Reads	Phantom Reads	Non-repeatable reads
Read Uncommitted	Might occur	Might occur	Might occur
Read Committed	Won't occur	Might occur	Might occur
Repeatable Read	Won't occur	Might occur	Won't occur
Serializable	Won't occur	Won't occur	Won't occur

60. What can you tell about WAL (Write Ahead Logging)?

Write Ahead Logging is a feature that increases the database reliability by logging changes **before** any changes are done to the database. This ensures that we have enough information when a database crash occurs by helping to pinpoint to what point the work has been complete and gives a starting point from the point where it was discontinued.

For more information, you can refer here.

61. What is the main disadvantage of deleting data from an existing table using the DROP TABLE command?

`DROP TABLE` command deletes complete data from the table along with removing the complete table structure too. In case our requirement entails just remove the data, then we would need to recreate the table to store data in it. In such cases, it is advised to use the TRUNCATE command.

62. How do you perform case-insensitive searches using regular expressions in PostgreSQL?

To perform case insensitive matches using a regular expression, we can use POSIX `(~*)` expression from pattern matching operators. For example:

```
'interviewbit' ~* '.*INTervIewBit.*'
```

63. How will you take backup of the database in PostgreSQL?

We can achieve this by using the pg_dump tool for dumping all object contents in the database into a single file. The steps are as follows:

Step 1: Navigate to the bin folder of the PostgreSQL installation path.

```
C:\>cd C:\Program Files\PostgreSQL\10.0\bin
```

requirement entails just remove the data, then we would need to recreate the table to store data in it. In such cases, it is advised to use the TRUNCATE command.

62. How do you perform case-insensitive searches using regular expressions in PostgreSQL?

To perform case insensitive matches using a regular expression, we can use POSIX `(~*)` expression from pattern matching operators. For example:

```
'interviewbit' ~* '.*INTervIewBit.*'
```

63. How will you take backup of the database in PostgreSQL?

We can achieve this by using the pg_dump tool for dumping all object contents in the database into a single file. The steps are as follows:

Step 1: Navigate to the bin folder of the PostgreSQL installation path.

```
C:\>cd C:\Program Files\PostgreSQL\10.0\bin
```

Step 2: Execute pg_dump program to take the dump of data to a .tar folder as shown below:

```
pg_dump -U postgres -W -F t sample_data > C:\Users\admin\pgbackup\sample_data.tar
```

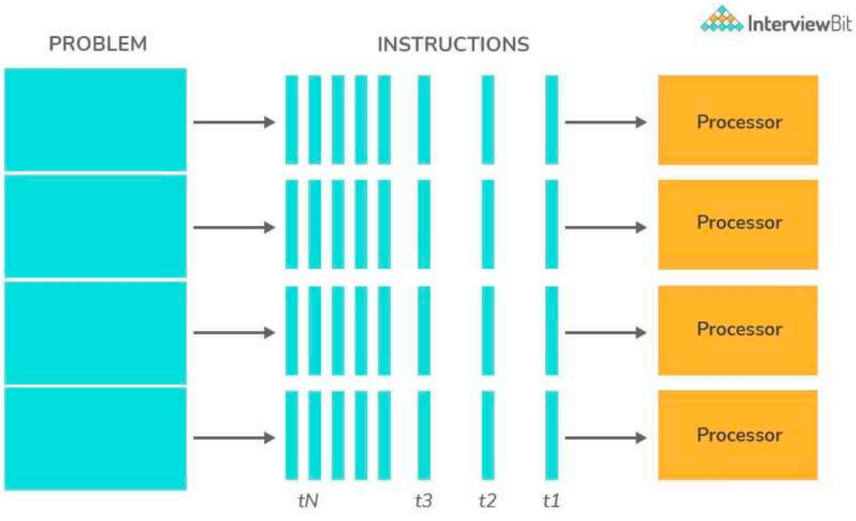
The database dump will be stored in the sample_data.tar file on the location specified.

64. Does PostgreSQL support full text search?

Full-Text Search is the method of searching single or collection of documents stored on a computer in a full-text based database. This is mostly supported in advanced database systems like SOLR or ElasticSearch. However, the feature is present but is pretty basic in PostgreSQL.

65. What are parallel queries in PostgreSQL?

Parallel Queries support is a feature provided in PostgreSQL for devising query plans capable of exploiting multiple CPU processors to execute the queries faster.



66. Differentiate between commit and checkpoint.

The commit action ensures that the data consistency of the transaction is maintained and it ends the current transaction in the section. Commit adds a new record in the log that describes the COMMIT to the memory. Whereas, a checkpoint is used for writing all changes that were committed to disk up to SCN which would be kept in datafile headers and control files.

Conclusion:

SQL is a language for the database. It has a vast scope and robust capability of creating and manipulating a variety of database objects using commands like CREATE, ALTER, DROP, etc, and also in loading the database objects using commands like INSERT. It also provides options for Data Manipulation using commands like DELETE, TRUNCATE and also does effective retrieval of data using cursor commands like FETCH, SELECT, etc. There are many such commands which provide a large amount of control to the programmer to interact with the database in an efficient way without wasting many resources. The popularity of SQL has grown so much that almost every programmer relies on this to implement their application's storage functionalities thereby making it an exciting language to learn. Learning this provides the developer a benefit of understanding the data structures used for storing the organization's data and giving an additional level of control and in-depth understanding of the application.

PostgreSQL being an open-source database system having extremely robust and sophisticated ACID, Indexing, and Transaction supports has found widespread popularity among the developer community.

References and Resources:

- PostgreSQL Download
- PostgreSQL Tutorial
- SQL Guide