Amr Abdelkarem

01

SQL INTERVIEW PREPARATION HANDBOOK – FROM BASICS TO ADVANCED

Programmingvalley.com

Q: What is the difference between CHAR and VARCHAR2?

A:

- CHAR stores fixed-length data and pads unused space with blanks.
- VARCHAR2 stores variable-length data and uses only the required space.

Example:
CHAR(10) with 'SQL' → 10 bytes
VARCHAR2(10) with 'SQL' → 3 bytes

Q: What is a view in SQL?

A:
A view is a virtual table created from a SELECT query.
It doesn’t store data but displays data from one or more tables.
Views make complex queries simpler, improve readability, and enhance security by limiting access to certain rows or columns.

Example:

```sql
CREATE VIEW employee_view AS
SELECT name, department
FROM employees
WHERE status = 'Active';
```

Q: What is the purpose of the UNIQUE constraint?

A:

The UNIQUE constraint ensures that all values in a column or a group of columns are distinct.

It prevents duplicate entries and maintains data integrity.

Example:

```sql
CREATE TABLE users (
    email VARCHAR(100) UNIQUE
);
```

Q: What is a composite primary key?

A:

A composite primary key combines two or more columns to uniquely identify each record when a single column cannot do so.

Example:

```sql
CREATE TABLE enrollment (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Q: What is the difference between the WHERE and HAVING clauses?

A:

- WHERE filters individual rows before grouping or aggregation. It cannot use aggregate functions like SUM or COUNT.
- HAVING filters grouped results after GROUP BY. It is used for conditions involving aggregate functions.

Example:

```sql
SELECT customer_id, COUNT(*) AS
orders_2025
FROM orders
WHERE order_date >= '2025-01-01' AND
order_date < '2026-01-01'
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

Q: What are SQL joins, and what are the differences between INNER, LEFT, RIGHT, and FULL joins?

A:

SQL joins combine rows from two tables based on a related column, usually a key.

- INNER JOIN → Returns rows present in both tables.
- LEFT JOIN → Returns all rows from the left table and matching rows from the right; unmatched rows on the right are NULL.
- RIGHT JOIN → Returns all rows from the right table and matching rows from the left; unmatched rows on the left are NULL.
- FULL JOIN → Returns all rows from both tables; unmatched rows are filled with NULL.

Example:

```sql
SELECT a.id, a.name, b.salary
FROM employees a
FULL JOIN salaries b
ON a.id = b.emp_id;
```

Q: What is a PRIMARY KEY, and how does it differ from a UNIQUE key?

A:

- PRIMARY KEY uniquely identifies each row in a table. It enforces both UNIQUE and NOT NULL. Only one primary key is allowed per table, but it can span multiple columns. It's commonly referenced by foreign keys.
- UNIQUE Key also enforces uniqueness but allows NULL values. Multiple unique constraints can exist in the same table.

Example:

```sql
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
);
```

Q: What is a CTE (Common Table Expression), and when would you use it?

A:
A CTE is a temporary named result set created using the WITH clause. It exists only during the execution of one SQL statement.
You use CTEs to:

- Simplify complex queries by breaking them into steps
- Avoid repeating subqueries
- Improve readability and maintainability
- Support recursive queries like organization charts or folder hierarchies

Example:

```sql
WITH dept_sales AS (
    SELECT department_id, SUM(sales) AS
total_sales
    FROM orders
    GROUP BY department_id
)
SELECT *
FROM dept_sales
WHERE total_sales > 100000;
```

Q: What is normalization, and what are the different normal forms?

A:

Normalization structures data to reduce redundancy and avoid update, insert, or delete anomalies. It organizes tables based on data dependencies while keeping relationships consistent.

Normal Forms:

- 1NF: Each column holds atomic values; no repeating groups.
- 2NF: In 1NF, and every non-key depends on the whole composite key.
- 3NF: In 2NF, and no transitive dependencies; non-keys depend only on the key.
- BCNF: Stronger than 3NF; for every dependency X→Y, X must be a candidate key.
- 4NF: Removes multi-valued dependencies.
- 5NF (PJNF): Removes join dependencies to allow lossless recombination.

Q: What is the difference between UNION and UNION ALL?

A:

- UNION combines results from multiple SELECT queries and removes duplicates. It performs a DISTINCT across all columns, which can make it slower.
- UNION ALL combines results but keeps duplicates. It’s faster since it doesn’t perform duplicate checks.

Example (UNION):

```
SELECT name FROM customers
UNION
SELECT name FROM employees;
```

Example (UNION ALL):

```
SELECT name FROM customers
UNION ALL
SELECT name FROM employees;
```

Q: What is the difference between clustered and non-clustered indexes, and when should each be used?

A:

- Clustered Index
  - Stores rows in the physical order of the index key (the table data itself is the index).
  - Only one per table.
  - Best for range queries and primary key lookups.
  - Use when the key is narrow, stable, and often sorted (e.g., order_id, order_date).
- Non-Clustered Index
  - Separate structure that maps key values to row locations.
  - Multiple per table allowed.
  - Best for frequent filters, joins, or sorts on specific columns.
  - Use when you need fast point lookups (e.g., WHERE email = ?) or to optimize queries with common JOIN, GROUP BY, or ORDER BY clauses.

Example:

```
CREATE CLUSTERED INDEX idx_orders_id ON
orders(order_id);
CREATE NONCLUSTERED INDEX
idx_orders_email ON orders(email);
```

Q: How do you perform pattern matching in SQL?

A:
Pattern matching is done using the LIKE or NOT LIKE operators with wildcards:

- % matches any sequence of characters
- _ matches a single character

Example:

```sql
SELECT * FROM employees
WHERE name LIKE 'A%';
```

(matches names starting with 'A')
Other Options:

- PostgreSQL: ILIKE (case-insensitive), SIMILAR TO, regex operators ~ and ~*
- MySQL / SQLite: REGEXP or REGEXP_LIKE
- All databases allow an optional ESCAPE clause to treat % or _ as normal characters.

Q: How would you calculate the running total of sales for each product?

A:

Use a window function to compute a cumulative SUM for each product ordered by date (and optionally by sale ID). It keeps individual rows while adding a running total column.

Example:

```sql
SELECT
  product_id,
  sale_date,
  amount,
  SUM(amount) OVER (
    PARTITION BY product_id
    ORDER BY sale_date, sale_id
    ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW
  ) AS running_total
FROM sales;
```

Q: What is a correlated subquery, and when would you use it?

A:

A correlated subquery depends on the current row of the outer query. It references columns from the outer query and runs once for each outer row.

Use Case:

Find employees whose salary is above their department's average.

Example:

```
SELECT e.employee_id, e.name, e.salary,
e.department_id
FROM employees e
WHERE e.salary >
      (SELECT AVG(e2.salary)
       FROM employees e2
       WHERE e2.department_id =
e.department_id);
```

Q: What are EXISTS and NOT EXISTS, and how do they differ from IN?

A:

- EXISTS checks if a subquery returns at least one row.
- NOT EXISTS checks that the subquery returns no rows.
- They return a boolean result and stop after finding the first match.

IN compares a value against a list or the output of a subquery.

Key Differences:

- EXISTS and NOT EXISTS use correlated subqueries and handle NULL safely.
- NOT IN is sensitive to NULL values and may produce no results if any NULL exists in the list.
- EXISTS generally performs better on large subqueries with indexes, while IN is better for small lists.

Example:

```sql
-- EXISTS
SELECT name FROM customers c
WHERE EXISTS (
  SELECT 1 FROM orders o
  WHERE o.customer_id = c.id
);

-- NOT IN
SELECT name FROM customers
WHERE id NOT IN (SELECT customer_id FROM
orders);
```

Q: What is an anti-join?

A:
An anti-join returns rows from one table that have no corresponding matches in another table. It identifies records that exist in one dataset but not in the other.

Example (LEFT JOIN):

```sql
SELECT a.id, a.name
FROM customers a
LEFT JOIN orders b ON a.id =
b.customer_id
WHERE b.customer_id IS NULL;
```

Example (NOT EXISTS):

```sql
SELECT a.id, a.name
FROM customers a
WHERE NOT EXISTS (
  SELECT 1 FROM orders b
  WHERE b.customer_id = a.id
);
```

Q: What is the difference between RANK(), DENSE_RANK(), and ROW_NUMBER()?

A:

All three are window functions that assign ordering numbers to rows.

- ROW_NUMBER() → Assigns a unique number to each row, even for ties.
- RANK() → Assigns the same rank to ties but skips numbers after ties (1,1,3...).
- DENSE_RANK() → Assigns the same rank to ties but doesn't skip numbers (1,1,2...).

Example:

```sql
SELECT name, salary,
  ROW_NUMBER() OVER (ORDER BY salary
DESC) AS row_number,
  RANK() OVER (ORDER BY salary DESC) AS
rank,
  DENSE_RANK() OVER (ORDER BY salary
DESC) AS dense_rank
FROM employees;
```

Q: What is the purpose of LAG and LEAD functions?

A:
LAG and LEAD are window functions used to access values from previous or next rows within the same result set. They help compare values across rows without using self-joins.

Use Cases:

- Track changes between consecutive rows (e.g., day-to-day sales difference)
- Detect trends or anomalies over time
- Fill missing values using data from adjacent rows

Example:

```sql
SELECT
  sale_date,
  amount,
  LAG(amount) OVER (ORDER BY sale_date)
AS prev_day_amount,
  LEAD(amount) OVER (ORDER BY sale_date)
AS next_day_amount,
  amount - LAG(amount) OVER (ORDER BY
sale_date) AS daily_change
FROM sales;
```

Q: What is a CROSS JOIN, and how does it differ from an INNER JOIN?

A:

- CROSS JOIN produces the Cartesian product of two tables. Every row from the first table is paired with every row from the second. It does not require a join condition.
  - Result size = rows(A) × rows(B)
  - Used to generate all possible combinations (e.g., dates × products, sizes × colors).
- INNER JOIN returns only rows that satisfy a join condition between the two tables (e.g., matching IDs). It produces a filtered result rather than all combinations.

Example:

```sql
-- CROSS JOIN
SELECT a.product, b.color
FROM products a
CROSS JOIN colors b;

-- INNER JOIN
SELECT a.id, a.name, b.order_id
FROM customers a
INNER JOIN orders b ON a.id =
b.customer_id;
```

Q: What is a foreign key, and how does it enforce referential integrity?

A:

A foreign key is a column or group of columns in one table that references the primary or unique key of another table. It ensures that every value in the child table exists in the parent table.

This enforces referential integrity by preventing orphaned records — you can't insert a child row with a non-existent parent, and you can't delete a parent row that still has related children (unless cascading rules are defined).

Example:

```sql
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES
departments(dept_id)
);
```

Q: What are set operations like UNION, INTERSECT, and EXCEPT, and when are they useful?

A:

Set operations combine results from two queries that have the same number of columns and compatible data types.

- UNION → Returns distinct rows from both queries (removes duplicates).
- Use when you need to merge data from multiple sources.
- INTERSECT → Returns only rows that appear in both queries.
- Use when you need to find common records.
- EXCEPT (or MINUS in Oracle) → Returns rows from the first query that don't exist in the second.
- Use when you need to find differences between datasets.

Example:

```
SELECT name FROM customers
UNION
SELECT name FROM suppliers;

SELECT name FROM customers
INTERSECT
SELECT name FROM suppliers;

SELECT name FROM customers
EXCEPT
SELECT name FROM suppliers;
```

Q: How would you optimize a slow SQL query?

A:
Query optimization focuses on identifying bottlenecks, improving execution plans, and reducing unnecessary work.
Steps:

1. Measure first
   - Reproduce the issue and record timings.
   - Use EXPLAIN or EXPLAIN ANALYZE to review execution plans and row estimates.
2. Fix fundamentals
   - Update table statistics.
   - Add or adjust indexes based on filter and join columns.
   - Ensure predicates are sargable (avoid functions or leading % in indexed columns).
3. Reduce data early
   - Use selective WHERE filters.
   - Fetch only needed columns instead of SELECT *.
   - Prefer EXISTS over IN for semi-joins.
4. Control joins and aggregations
   - Check join selectivity to prevent row explosion.
   - Remove duplicates before joins.
   - Pre-aggregate data when possible.
5. Rewrite inefficient patterns
   - Replace wide OR conditions with UNION ALL.
   - Convert correlated subqueries into joins.
   - Use window functions efficiently.
6. Handle large datasets wisely
   - Use keyset (seek) pagination instead of OFFSET.
   - Implement materialized views, caching, or partitioning for recurring heavy queries.

Q: What is a query in SQL?

A:
A query is a SQL command used to retrieve, insert, update, or delete data in a database.
The most common type is the SELECT query, which fetches data from one or more tables based on defined conditions.

Example:

```sql
SELECT name, salary
FROM employees
WHERE department = 'HR';
```

Q: What is a subquery?

A:
A subquery is a query embedded inside another SQL query. It returns a result that the outer query uses for filtering, comparison, or computation.
Subqueries are commonly used in WHERE, FROM, or SELECT clauses to handle complex filtering or calculations.

Example:

```sql
SELECT name, salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
);
```

Q: What is database partitioning?

A:
Database partitioning divides a large table and its indexes into smaller, more manageable parts called partitions while keeping it logically as one table.
Benefits:

- Improves query performance by scanning only relevant partitions (partition pruning).
- Simplifies maintenance tasks such as backup, reindexing, and archiving.
- Increases availability by isolating failures or heavy operations to specific partitions.

Example (range partitioning):

```
CREATE TABLE sales (
    id INT,
    sale_date DATE,
    amount DECIMAL
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p2023 VALUES LESS THAN
(2024),
    PARTITION p2024 VALUES LESS THAN
(2025)
);
```

Q: What strategies protect a web application from SQL injection?

A:

SQL injection is prevented by ensuring user input never alters query structure.

Key Strategies:

- Parameterized queries (prepared statements): Always bind parameters instead of concatenating strings.
- Input validation: Allow only expected patterns (e.g., digits for IDs, fixed enums for status).
- Least privilege: Use database accounts with only necessary permissions (no DROP, limited schema access).
- Safe stored procedures: Avoid building dynamic SQL inside procedures.
- Escaping and ORM frameworks: Use database libraries or ORMs that automatically escape input and enforce safe query patterns.

Example (Python):

```
cursor.execute("SELECT * FROM users
WHERE id = %s", (user_id,))
```

Q: What are the main types of SQL commands?

A:
SQL commands are grouped by their purpose in managing data and structure.

- DDL (Data Definition Language): Defines and modifies database objects.
- CREATE, ALTER, DROP, TRUNCATE
- DML (Data Manipulation Language): Manages data inside tables.
- SELECT, INSERT, UPDATE, DELETE
- DCL (Data Control Language): Controls access and permissions.
- GRANT, REVOKE
- TCL (Transaction Control Language): Manages transactions and ensures consistency.
- COMMIT, ROLLBACK, SAVEPOINT

Q: What is the purpose of the DEFAULT constraint?

A:
The DEFAULT constraint assigns a predefined value to a column when no value is specified during an INSERT. It ensures consistency and simplifies data entry.

Example:

```sql
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    status VARCHAR(20) DEFAULT 'Active'
);
```

If no status is provided, 'Active' is automatically inserted.

Q: What is denormalization, and when is it used?

A:
Denormalization combines normalized tables into fewer, larger tables to improve query performance. It trades some redundancy for faster reads.

When to Use:

- When frequent joins cause slow performance.
- In reporting or analytics systems focused on read-heavy workloads.
- When data is mostly static and consistency risks are manageable.

Example:
Merging orders and customers into a single table to avoid joining during reporting.

Q: What are the different operators available in SQL?

A:
SQL provides several categories of operators for calculations, comparisons, and logical expressions.

- Arithmetic Operators: +, -, *, /, %
- Comparison Operators: =, !=, <>, >, <, >=, <=
- Logical Operators: AND, OR, NOT
- Set Operators: UNION, INTERSECT, EXCEPT
- Special Operators: BETWEEN, IN, LIKE, IS NULL
- Concatenation Operators:
  - || (Oracle, PostgreSQL)
  - + (SQL Server)

Example:

```sql
SELECT name
FROM employees
WHERE salary > 5000 AND department IN
('HR', 'IT');
```

Q: What are the different types of joins in SQL?

A:
Joins combine data from multiple tables based on related columns.

- INNER JOIN: Returns rows that have matching values in both tables.
- LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right; unmatched rows on the right are NULL.
- RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left; unmatched rows on the left are NULL.
- FULL JOIN (FULL OUTER JOIN): Returns all rows when there is a match in either table; unmatched rows are filled with NULL.
- CROSS JOIN: Produces the Cartesian product of both tables (every row from A paired with every row from B).
- SELF JOIN: Joins a table with itself, often for hierarchical or relational comparisons.

Example:

```sql
SELECT e.name, m.name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id =
m.id;
```

Q: What is the purpose of the GROUP BY clause?

A:
The GROUP BY clause groups rows that have the same values in specified columns. It’s used with aggregate functions to perform calculations per group instead of on the full table.

Example:

```sql
SELECT department_id, COUNT(*) AS
employee_count, AVG(salary) AS
avg_salary
FROM employees
GROUP BY department_id;
```

This query returns one row per department with the total number of employees and their average salary.

Q: What are aggregate functions in SQL?

A:
Aggregate functions compute a single result from a group of rows. They are often used with GROUP BY to summarize data.

Common Aggregate Functions:

- COUNT() → Counts the number of rows.
- SUM() → Returns the total of numeric values.
- AVG() → Calculates the average value.
- MIN() → Finds the smallest value.
- MAX() → Finds the largest value.

Example:

```sql
SELECT department_id, COUNT(*) AS
total_employees, AVG(salary) AS
avg_salary
FROM employees
GROUP BY department_id;
```

Q: What are indexes, and why are they used?

A:

Indexes are database structures that speed up data retrieval by allowing quick lookups instead of scanning the entire table. They function like a book index that helps locate data efficiently.

Why Use Indexes:

- Improve performance for SELECT, JOIN, and WHERE queries.
- Reduce I/O by accessing only relevant rows.
- Enforce uniqueness for specific columns (e.g., user emails).

Trade-offs:

- Consume extra storage.
- Slow down INSERT, UPDATE, and DELETE operations due to index maintenance.

Types of Indexes:

- Clustered Index: Physically sorts data by the key (one per table).
- Non-Clustered Index: Separate structure pointing to data rows (many allowed).
- Unique Index: Ensures all values are distinct.
- Composite Index: Built on multiple columns for combined lookups.

Q: What is the difference between DELETE and TRUNCATE commands?

A:
Both remove data from a table but differ in type, behavior, and performance.

- DELETE
  - Type: DML (Data Manipulation Language)
  - Removes rows one by one.
  - Records each deletion in the transaction log → supports rollback.
  - Can use a WHERE clause to remove specific rows.
- TRUNCATE
  - Type: DDL (Data Definition Language)
  - Removes all rows instantly without logging each one.
  - Cannot include a WHERE clause.
  - Faster for large tables but cannot selectively delete rows.

Example:

```
DELETE FROM employees WHERE
department_id = 10;
TRUNCATE TABLE employees;
```

Q: What are the differences between SQL and NoSQL databases?

A:
SQL and NoSQL differ in structure, scalability, and consistency model.

SQL Databases:

- Store data in structured tables with rows and columns.
- Use a fixed schema.
- Support ACID (Atomicity, Consistency, Isolation, Durability) transactions.
- Best suited for complex queries and relationships.
- Examples: MySQL, PostgreSQL, Oracle, SQL Server.

NoSQL Databases:

- Store data in flexible, schema-less formats (key-value, document, column, graph).
- Scale horizontally across many servers.
- Often prioritize availability and scalability over strict consistency (BASE model).
- Ideal for large volumes of unstructured or rapidly changing data.
- Examples: MongoDB, Cassandra, Redis, DynamoDB.

Q: What are the types of constraints in SQL?

A:
Constraints define rules to maintain accuracy and consistency of data in a table.

Main Types:

- NOT NULL: Prevents a column from storing NULL values.
- UNIQUE: Ensures all values in a column are distinct.
- PRIMARY KEY: Uniquely identifies each row (implies UNIQUE + NOT NULL).
- FOREIGN KEY: Maintains referential integrity by referencing another table's key.
- CHECK: Ensures values meet a specific condition.
- DEFAULT: Assigns a predefined value when no value is provided.

Example:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT CHECK (age >= 18),
    department_id INT,
    salary DECIMAL DEFAULT 3000,
    FOREIGN KEY (department_id)
REFERENCES departments(id)
);
```

Q: What is a cursor in SQL?

A:
A cursor is a database object that allows row-by-row processing of a query result set. It's used when operations require sequential handling rather than set-based processing.
Common Use Case:
When each row needs custom logic or procedural handling inside stored procedures.

Types of Cursors (SQL Server):

- STATIC: Takes a snapshot of the result set; changes made after opening are not reflected.
- DYNAMIC: Reflects all changes made to the underlying data while open.
- FORWARD_ONLY: Moves only forward through the result set.
- KEYSET: Uses keys to identify rows; updates to non-key columns are visible.

Example:

```
DECLARE employee_cursor CURSOR FOR
SELECT name, salary FROM employees;

OPEN employee_cursor;
FETCH NEXT FROM employee_cursor;
-- process each row
CLOSE employee_cursor;
DEALLOCATE employee_cursor;
```

Q: What is a trigger in SQL?

A:
A trigger is a stored set of SQL statements that automatically run when a specific database event occurs—such as INSERT, UPDATE, or DELETE. Triggers help enforce rules, maintain data integrity, and automate actions.

Types:

- BEFORE Trigger: Executes before the triggering event.
- AFTER Trigger: Executes after the event completes.

Common Uses:

- Enforcing business rules (e.g., preventing invalid updates).
- Maintaining audit logs.
- Validating or transforming data automatically.

Example:

```
CREATE TRIGGER update_audit
AFTER UPDATE ON employees
FOR EACH ROW
INSERT INTO audit_log (emp_id,
action_time)
VALUES (NEW.id, NOW());
```

Q: What is the purpose of the SQL SELECT statement?

A:
The SELECT statement retrieves data from one or more tables in a database. It allows filtering, sorting, grouping, and joining to extract meaningful information.

Example:

```sql
SELECT name, department, salary
FROM employees
WHERE department = 'IT'
ORDER BY salary DESC;
```

Q: What is the purpose of the ORDER BY clause?

A:
The ORDER BY clause sorts query results in ascending (ASC, default) or descending (DESC) order based on one or more columns.

Example:

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

This example sorts employees first by highest salary, then alphabetically by name.

Q: What is a table in SQL?

A:
A table is a structured collection of data stored in rows and columns.

- Columns define the data type and attributes (e.g., name, age, salary).
- Rows represent individual records.

Example:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10,2)
);
```

Q: What are NULL values in SQL?

A:
NULL represents a missing, unknown, or inapplicable value in a database.
It is not the same as zero or an empty string — it means “no value.”

Key Points:

- Comparisons using = or != with NULL return unknown; use IS NULL or IS NOT NULL instead.
- Aggregate functions ignore NULL values unless specified otherwise.

Example:

```sql
SELECT name
FROM employees
WHERE manager_id IS NULL;
```

Q: What is a stored procedure?

A:
A stored procedure is a precompiled group of SQL statements stored in the database. It can accept input parameters, execute logic or queries, and return output values or result sets.

Benefits:

- Improves performance by avoiding repeated parsing and compilation.
- Centralizes business logic for easier maintenance.
- Enhances security by controlling direct access to tables.

Example:

```sql
CREATE PROCEDURE
GetEmployeeByDept(@dept_id INT)
AS
BEGIN
    SELECT name, salary
    FROM employees
    WHERE department_id = @dept_id;
END;
```

Q: What is the difference between DDL and DML commands?

A:

1. DDL (Data Definition Language):
Defines and modifies the structure of database objects like tables, indexes, and views.
Affects the schema, not the data.
Changes are auto-committed.
Common Commands:
CREATE, ALTER, DROP, TRUNCATE
Example:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

2. DML (Data Manipulation Language):
Used to manage and manipulate the data inside tables.
Supports transactions and can be rolled back.
Common Commands:
SELECT, INSERT, UPDATE, DELETE
Example:

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

Q: What is the purpose of the ALTER command in SQL?

A:
The ALTER command modifies the structure of an existing database object, such as a table or index. It’s used to adjust the schema as requirements change.

Common Uses:

- Add or drop columns.
- Change a column’s data type or size.
- Add or remove constraints.
- Rename columns or tables.
- Modify indexing or storage settings.

Example:

```
ALTER TABLE employees ADD hire_date
DATE;
ALTER TABLE employees MODIFY name
VARCHAR(100);
ALTER TABLE employees DROP COLUMN
department;
```

Q: How is data integrity maintained in SQL databases?

A:
Data integrity ensures data remains accurate, consistent, and reliable. SQL databases maintain it through several key mechanisms:

- Constraints:
- Enforce rules at the column or table level.
  - NOT NULL → prevents missing values.
  - UNIQUE → prevents duplicates.
  - FOREIGN KEY → ensures valid references between tables.
  - CHECK → enforces valid ranges or conditions.
- Transactions:
- Guarantee all operations in a unit succeed or fail together using ACID properties (COMMIT / ROLLBACK).
- Triggers:
- Automatically enforce business rules before or after data changes.
- Normalization:
- Structures data into related tables to eliminate redundancy and avoid anomalies.
- Cascading Actions:
- Foreign keys can use ON DELETE CASCADE or ON UPDATE CASCADE to automatically maintain referential integrity.

These mechanisms together preserve data consistency and trustworthiness across all operations.

Q: How does the CASE statement work in SQL?

A:

The CASE statement adds conditional logic to SQL queries. It checks each condition in order and returns the value for the first condition that evaluates to true. If no condition matches, it returns the value from the ELSE clause.

Example:

```sql
SELECT id,
       CASE
           WHEN salary > 100000 THEN
'High'
           WHEN salary BETWEEN 50000 AND
100000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_level
FROM employees;
```

This query labels each employee's salary as High, Medium, or Low based on defined conditions.

Q: What is the purpose of the COALESCE function?

A:
The COALESCE function returns the first non-NULL value from a list of expressions. It’s used to replace NULL values with defaults or fallback values.

Example:

```sql
SELECT COALESCE(phone, mobile, 'No
Contact') AS contact_number
FROM customers;
```

If phone is NULL, it checks mobile; if both are NULL, it returns 'No Contact'.

Q: What is the purpose of the COALESCE function?

A:
The COALESCE function returns the first non-NULL value from a list of expressions. It’s used to replace NULL values with defaults or fallback values.

Example:

```
SELECT COALESCE(phone, mobile, 'No
Contact') AS contact_number
FROM customers;
```

If phone is NULL, it checks mobile; if both are NULL, it returns 'No Contact'.

Q: What are the differences between SQL's COUNT() and SUM() functions?

A:

Both are aggregate functions but serve different purposes.

COUNT():

Counts the number of rows or non-NULL values in a column.

```sql
SELECT COUNT(*) FROM orders;
```

Counts all rows in the orders table.

SUM():

Calculates the total of numeric values in a column.

```sql
SELECT SUM(total_amount) FROM orders;
```

Returns the total value of the total_amount column

Key Difference:

COUNT() measures quantity (rows), while SUM() measures total value (numeric data).

Q: What is the difference between the NVL and NVL2 functions?

A:

Both handle NULL values but behave differently.

NVL(expr1, expr2)

Replaces NULL in expr1 with expr2.

```sql
SELECT NVL(salary, 0) AS adjusted_salary
FROM employees;
```

If salary is NULL, it returns 0.

NVL2(expr1, expr2, expr3)

Returns expr2 if expr1 is not NULL, otherwise returns expr3.

```sql
SELECT NVL2(salary, salary, 0) AS
adjusted_salary
FROM employees;
```

If salary is not NULL, it returns salary; if it's NULL, it returns 0.

Key Difference:

- NVL handles a single replacement for NULL.
- NVL2 lets you define separate values for both NULL and non-NULL cases.

Q: What are scalar functions in SQL?

A:
Scalar functions operate on a single input value and return one output value. They're often used to format, transform, or compute data at the column or row level.

Common Scalar Functions:

- LEN() → Returns the length of a string.
- ROUND() → Rounds a numeric value to a specified precision.
- CONVERT() → Converts a value from one data type to another.
- UPPER() / LOWER() → Changes text case.
- GETDATE() → Returns the current system date and time.

Example:

```sql
SELECT name,
       LEN(name) AS name_length,
       ROUND(salary, 2) AS
rounded_salary
FROM employees;
```

Q: What happens if you use COUNT() on NULLs?

A:

- COUNT(column) ignores NULL values — it only counts rows where the column has a non-NULL value.
- COUNT(*) counts all rows, including those with NULL values in any column.

Example:

```sql
SELECT
    COUNT(salary) AS non_null_salaries,
    COUNT(*) AS total_rows
FROM employees;
```

If a table has 10 rows and 2 NULL salaries, COUNT(salary) returns 8, while COUNT(*) returns 10.

Q: What are window functions, and how are they used?

A:

Window functions perform calculations across a set of rows related to the current row, without reducing the result set. They're useful for analytics such as rankings, running totals, and moving averages.

Common Uses:

- Ranking rows (RANK(), DENSE_RANK(), ROW_NUMBER())
- Aggregates over partitions (SUM(), AVG(), COUNT())
- Accessing neighboring rows (LAG(), LEAD())

Example — Running Total:

```sql
SELECT name, salary,
       SUM(salary) OVER (ORDER BY
salary) AS running_total
FROM employees;
```

Each row shows its own data plus the cumulative total of all preceding salaries.

Q: What is the difference between an index and a key in SQL?

A:

1. Index

An index is a physical database object that improves query performance by allowing faster data retrieval. It creates an ordered structure (like a lookup map) for efficient searches.

- Can be unique or non-unique.
- Used mainly for performance optimization.

Example:

```
CREATE INDEX idx_lastname ON
employees(last_name);
```

2. Key

A key is a logical constraint that enforces data integrity and relationships within tables.

- PRIMARY KEY: Ensures each row is unique and not null.
- FOREIGN KEY: Maintains referential integrity between tables.
- UNIQUE KEY: Prevents duplicate values in a column.

Example:

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    department_id INT,
    FOREIGN KEY (department_id)
REFERENCES departments(id)
);
```

Key Difference:

- Index → improves performance.
- Key → enforces data integrity and relationships.

Q: How does indexing improve query performance?

A:

Indexes make data retrieval faster by allowing the database to locate rows directly instead of scanning the entire table. They work like a lookup table that maps key values to physical row locations.

How It Helps:

- Reduces full table scans and disk I/O.
- Speeds up WHERE, JOIN, ORDER BY, and GROUP BY operations.
- Enhances query response time on large datasets.

Example:

```sql
CREATE INDEX idx_lastname ON
employees(last_name);

SELECT *
FROM employees
WHERE last_name = 'Smith';
```

Here, the database uses the idx_lastname index to find all employees named “Smith” without checking every row in the table.

Q: What are the trade-offs of using indexes in SQL databases?

A:

Advantages:

- Speeds up SELECT queries, especially those using WHERE, JOIN, or ORDER BY.
- Improves sorting and filtering performance.
- Reduces overall query execution time for large datasets.

Disadvantages:

- Consumes additional storage for index structures.
- Slows down INSERT, UPDATE, and DELETE operations since indexes must be updated with every data change.
- Makes bulk inserts and batch loads slower due to index maintenance.

Summary:
Indexes improve read performance but increase storage needs and write overhead. The right balance depends on whether your workload is read-heavy or write-heavy.

Q: What are temporary tables, and how are they used?

A:
Temporary tables store intermediate results and exist only for the duration of a session or transaction. They're useful for breaking down complex queries, caching intermediate data, or performing transformations without affecting main tables.

Types of Temporary Tables:

- Local Temporary Tables (#TempTable):
  - Visible only to the session that created them.
  - Automatically deleted when the session ends.
- Global Temporary Tables (##GlobalTempTable):
  - Accessible by all sessions.
  - Dropped when all sessions referencing them are closed.

Example:

```
CREATE TABLE #TempResults (id INT, value
VARCHAR(50));

INSERT INTO #TempResults VALUES (1,
'Test');
SELECT * FROM #TempResults;
```

The table #TempResults is created for temporary use and automatically removed when the session ends.

Q: What is a materialized view, and how does it differ from a standard view?

A:

Standard View:

- Acts as a virtual table defined by a query.
- Does not store data; it executes the underlying query each time it's accessed.
- Always displays real-time data from base tables.

Materialized View:

- Physically stores the query result as a table.
- Provides faster reads for complex or aggregate queries.
- Must be refreshed periodically to stay up to date.
- Can use refresh modes like ON DEMAND or ON COMMIT, depending on the database system.

Example (PostgreSQL / Oracle):

```
CREATE MATERIALIZED VIEW sales_summary
AS
SELECT region, SUM(amount) AS
total_sales
FROM sales
GROUP BY region;

REFRESH MATERIALIZED VIEW sales_summary;
```

Key Difference:

- Standard View: Real-time, always current, slower for large queries.
- Materialized View: Cached data, faster access, needs manual or scheduled refresh.

Q: What is a sequence in SQL?

A:
A sequence is a database object that generates a sequential series of unique numeric values, typically used for primary keys or ID columns. Each call to the sequence produces the next number in the series.

Key Features:

- Auto-increments independently of any table.
- Can define start value, increment step, and limits.
- Ensures uniqueness without locking the table.

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
```

Used in inserts to auto-generate IDs:

```
INSERT INTO employees (emp_id, name)
VALUES (NEXT VALUE FOR seq_emp_id,
'John');
```

Q: What are the advantages of using sequences over identity columns?

A:

1. Greater Flexibility

- Can define start value, increment, minimum, and maximum limits.
- Work as independent objects that can be shared across multiple tables.

2. Dynamic Adjustment

- Can alter sequence properties (e.g., restart value or increment) without changing the table schema.

3. Cross-Table Consistency

- **O**ne sequence can generate unique IDs for several related tables, ensuring no duplication across them.

Summary:

Sequences provide more control, configurability, and reusability than identity columns, which are tied to a single table.

Q: How do constraints improve database integrity?

A:

Constraints ensure data accuracy, consistency, and reliability by enforcing predefined rules on the database. They prevent invalid or conflicting data from being inserted or updated.

Main Types of Constraints:

- NOT NULL: Prevents missing values.
- UNIQUE: Ensures all values in a column are distinct.
- PRIMARY KEY: Combines NOT NULL and UNIQUE to uniquely identify each row.
- FOREIGN KEY: Enforces referential integrity by linking related tables.
- CHECK: Ensures values meet specific conditions, such as CHECK (salary > 0).

Example:

```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    salary DECIMAL CHECK (salary > 0),
    department_id INT,
    FOREIGN KEY (department_id)
REFERENCES departments(id)
);
```

By automatically applying these rules, constraints keep the database valid and consistent.

Q: What is the difference between a local and a global temporary table?

A:

Local Temporary Table:

- Name starts with # (e.g., #TempTable).
- Exists only for the session that created it.
- Automatically dropped when the session ends.
- Not accessible by other connections.

Global Temporary Table:

- Name starts with ## (e.g., ##GlobalTempTable).
- Visible to all sessions and users.
- Dropped only when all sessions using it are closed.

Example:

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

Use local temporary tables for session-specific operations and global ones for shared temporary data across sessions.

Q: What is the purpose of the SQL MERGE statement?

A:

The MERGE statement performs INSERT, UPDATE, and DELETE operations in a single command. It’s used to synchronize data between a source table and a target table efficiently.

Functions:

- INSERT: Adds new rows that don’t exist in the target.
- UPDATE: Modifies existing rows that match between source and target.
- DELETE: Removes rows from the target that meet specific conditions.

Example:

```
MERGE INTO target_table AS t
USING source_table AS 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);
```

Note:

While powerful, MERGE can introduce concurrency issues such as race conditions in SQL Server if not handled carefully.

Q: How can you handle duplicates in a query without using DISTINCT?

A:

You can remove duplicates using grouping or window functions instead of DISTINCT.

1. Using GROUP BY

Groups rows by one or more columns and returns a single row per group.

```sql
SELECT column1, MAX(column2) AS column2
FROM table_name
GROUP BY column1;
```

2. Using ROW_NUMBER()

Assigns a sequence number to each row within a partition, then filters out duplicates.

```sql
WITH cte AS (
    SELECT column1, column2,
           ROW_NUMBER() OVER (PARTITION
BY column1 ORDER BY column2) AS row_num
    FROM table_name
)
SELECT column1, column2
FROM cte
WHERE row_num = 1;
```

Summary:

- GROUP BY is simple and efficient for aggregate-based deduplication.
- ROW_NUMBER() provides more control over which record to keep.

Q: What are the ACID properties of a transaction?

A:
ACID represents the four essential properties that guarantee reliable and consistent database transactions: Atomicity, Consistency, Isolation, and Durability.

1. Atomicity

- A transaction is an all-or-nothing operation.
- If any step fails, the entire transaction is rolled back.
- Example: In a money transfer, both debit and credit must complete — or neither happens.

2. Consistency

- Ensures the database moves from one valid state to another.
- All integrity constraints, rules, and relationships remain intact.
- Example: A balance cannot become negative if rules prevent it.

3. Isolation

- Each transaction runs independently of others.
- Intermediate results are hidden until the transaction commits.
- Example: Two users booking the same seat won't see inconsistent states.

4. Durability

- Once committed, changes are permanent, even after a crash or restart.
- Example: A successfully placed order remains saved after a system reboot.

Together, these properties ensure data integrity and reliability in all transactional systems.

Q: What are the differences between isolation levels in SQL?

A:

Isolation levels control how transactions interact with each other to maintain consistency when multiple transactions run concurrently. Each level balances data accuracy and performance differently.

1. Read Uncommitted

- Transactions can read uncommitted (in-progress) changes from others.
- Fastest but least safe.
- Problems: Dirty reads, non-repeatable reads, phantom reads.

2. Read Committed

- Only reads data that has been committed.
- Prevents dirty reads.
- Problems: Non-repeatable reads and phantom reads may still occur.

3. Repeatable Read

- Ensures rows read cannot change during the transaction.
- Prevents dirty and non-repeatable reads.
- Problem: Phantom reads can still occur when new rows are inserted by others.

4. Serializable

- Highest isolation; transactions execute as if sequentially.
- Prevents dirty reads, non-repeatable reads, and phantom reads.
- Trade-off: Strong consistency but slower performance due to locking and reduced concurrency.

Q: What is the purpose of the WITH (NOLOCK) hint in SQL Server?

A:
The WITH (NOLOCK) hint allows reading data without taking shared locks or waiting for other transactions to release their locks. It reads uncommitted data, improving concurrency and performance in high-traffic systems.
Behavior:

- Reduces blocking between read and write operations.
- Allows faster queries on large or busy tables.
- Reads uncommitted (dirty) data — meaning results may be inconsistent or include rolled-back changes.

Example:

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

This query retrieves data without locking rows, avoiding waits caused by other active transactions.

Use Case:
WITH (NOLOCK) can be helpful for read-only analytics or reporting queries where absolute accuracy isn't critical, but it should be avoided in transactional or financial operations where data integrity matters.

Q: How do you handle deadlocks in SQL databases?

A:
Deadlocks happen when two or more transactions block each other by holding resources the others need. To handle and prevent them, use a combination of detection, retry logic, and good design practices.
1. Deadlock Detection and Retry
- Most databases automatically detect deadlocks and abort one transaction (the “victim”) to resolve the conflict.
- The aborted transaction should be retried after a short delay.

```
BEGIN TRY
    BEGIN TRANSACTION;
    -- transactional work
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 1205  -- SQL
Server deadlock error
        WAITFOR DELAY '00:00:02'; --
short wait before retry
END CATCH;
```

2. Keep Transactions Short and Ordered
- Acquire locks in a consistent order across all transactions.
- Avoid holding locks longer than necessary.

3. Use Lower Isolation Levels (When Safe)
- Switching from SERIALIZABLE to READ COMMITTED or READ UNCOMMITTED can reduce locking contention.

4. Index Properly
- Proper indexes reduce the number of locked rows and improve query efficiency.

5. Avoid User Interaction in Transactions
- Never pause for user input mid-transaction; it increases lock duration and deadlock risk.

Summary:
Detect, retry, and design transactions carefully — short, consistent, and well-indexed transactions minimize deadlocks.

Q: What is a database snapshot, and how is it used?

A:
A database snapshot is a read-only, static copy of a database at a specific point in time. It captures the database's state without duplicating all the data, using a copy-on-write mechanism to track changes made after the snapshot is created.

Common Uses:

- Reporting: Enables querying consistent, point-in-time data without impacting the live database.
- Backup and Recovery: Allows quick rollback to a previous state after accidental changes or corruption.
- Testing and Auditing: Provides a stable, unchanging dataset for verification or test environments.

Example (SQL Server):

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

To revert to the snapshot:

```
RESTORE DATABASE MyDatabase FROM
DATABASE_SNAPSHOT = 'MySnapshot';
```

Key Point:
Snapshots are lightweight and fast for reads but cannot be updated. They depend on the original database and grow as data changes.

Q: What are the differences between OLTP and OLAP systems?

A:

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.

Q: What is a live lock, and how does it differ from a deadlock?

A:

1. Live Lock

- Occurs when two or more transactions continuously react to each other's actions, preventing completion.
- Transactions remain active but make no progress — they keep retrying or yielding resources instead of completing.
- Example: Two transactions repeatedly releasing and reacquiring locks to avoid blocking, yet never finishing.

2. Deadlock

- Occurs when two or more transactions hold resources the others need, each waiting indefinitely for the other to release them.
- All involved transactions are blocked until one is terminated.

Key Difference:

- Deadlock: Transactions are stuck waiting.
- Live Lock: Transactions are active but not progressing.

In both cases, no useful work is completed, but live locks involve continuous activity, while deadlocks involve complete waiting.

Q: What is the purpose of the SQL EXCEPT operator?

A:
The EXCEPT operator returns rows from the first query that do not appear in the second query. It performs a set difference between two result sets.

Behavior:

- Removes duplicates by default (like UNION).
- Both queries must have the same number of columns with compatible data types.

Example:

```
SELECT product_id FROM products_sold
EXCEPT
SELECT product_id FROM
products_returned;
```

Returns products that were sold but not returned.

Use Cases:

- Finding discrepancies between datasets.
- Checking for records that exist in one table but not another.

Performance Note:
Efficient when both datasets are indexed; large unindexed tables can slow performance due to full comparisons.

Q: How do you implement dynamic SQL, and what are its advantages and risks?

A:
Dynamic SQL is SQL code built and executed at runtime instead of being predefined in the application or script. It's useful when object names, filters, or conditions are not known in advance.

Implementation (SQL Server Example):

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

In other databases, dynamic SQL is typically executed through query concatenation functions or runtime execution commands.

Advantages:

- Flexibility: Can adapt to variable table names, columns, or conditions at runtime.
- Simplified Logic: Reduces the need for multiple hardcoded query variations.
- Reusable Logic: Enables building generalized procedures for different datasets.

Risks:

- SQL Injection: Unsanitized user input can allow execution of harmful SQL. Always validate and parameterize inputs.
- Performance Overhead: Execution plans aren't always cached, which can slow performance.
- Debugging Difficulty: Dynamically built queries are harder to trace and troubleshoot.

Best Practice:
Use sp_executesql with parameterization to mitigate SQL injection and improve execution plan reuse.

Q: What is the difference between horizontal and vertical partitioning?

A:
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.

Q: What are the considerations for indexing very large tables?

A:
Indexing large tables requires a balance between query performance and maintenance overhead.

1. Indexing Strategy
- Prioritize columns used in WHERE, JOIN, and ORDER BY clauses.
- Avoid over-indexing — every index consumes storage and slows write operations.

2. Index Types
- Clustered Index: Best for primary key and range-based queries.
- Non-Clustered Index: Improves filtering and sorting on non-key columns.
- Use covering indexes to include all columns a query needs.

3. Partitioned Indexes
- For partitioned tables, create local indexes per partition.
- Speeds up queries that target specific partitions.
- Simplifies maintenance and parallel operations.

4. Maintenance Overhead
- Regularly monitor fragmentation and rebuild or reorganize indexes during off-peak hours.
- Consider online index rebuilds to avoid blocking.
- Reassess indexing needs as data volume and query patterns evolve.

5. Monitoring and Tuning
- Review execution plans to confirm index usage.
- Drop unused or redundant indexes.
- Keep statistics updated for optimal query optimization.

Summary:
Effective indexing of large tables focuses on the most beneficial columns, minimizes write overhead, and uses partitioning and maintenance strategies to sustain long-term performance.

Q: What is the difference between database sharding and partitioning?

A:

Both techniques divide large datasets into smaller parts but differ in scope and purpose.

1. Sharding

- Splits a database into multiple independent databases (shards).
- Each shard holds a subset of the data and operates on its own server.
- Aimed at horizontal scaling across multiple machines.
- Used to handle very large datasets and high query loads.
- Example: A global users database divided by region — one shard for North America, another for Europe, another for Asia.
- Key Benefit: Distributes load across servers, improving scalability and fault isolation.

2. Partitioning

- Divides a single table or database into smaller, logical pieces.
- All partitions remain within the same database instance.
- Aimed at performance optimization and maintenance efficiency.
- Example: A sales table partitioned by year so that queries for recent sales skip older partitions.
- Key Benefit: Reduces query scan size, simplifies maintenance, and improves data management.

Key Difference:

- Sharding: Distributes data across multiple databases or servers (horizontal scaling).
- Partitioning: Organizes data within a single database (performance and management optimization).

Q: What are best practices for writing optimized SQL queries?

A:

- Keep queries simple and readable
- Filter early with selective WHERE predicates
- Avoid SELECT * and return only needed columns
- Index columns used in WHERE, JOIN, ORDER BY
- Review execution plans to spot scans, bad estimates, missing indexes
- Pick the correct join type for the relationship
- Break complex logic into CTEs or temporary tables
- Optimize aggregations; pre-aggregate when reused
- Monitor performance and update statistics regularly

Examples:

```sql
-- Only needed columns
SELECT id, name FROM customers WHERE
country = 'US';

-- Sargable predicate (no function on
indexed column)
WHERE order_date >= '2025-01-01'

-- Covering index idea
CREATE INDEX ix_orders_cust_date ON
orders(customer_id, order_date)
INCLUDE(total);
```

Q: How can you monitor query performance in a production database?

A:

- Review execution plans to see scans, seeks, and index use
- Track wait stats to find lock, I/O, or CPU bottlenecks
- Use built-in tools
  - SQL Server: Query Store, DMVs, performance dashboards
  - MySQL: EXPLAIN, SHOW PROFILE, Performance Schema
  - PostgreSQL: EXPLAIN ANALYZE, pg_stat_statements, logs
- Set baselines and alerts on duration, CPU, memory, IOPS
- Enable slow-query logging and sample the worst offenders
- Tune continuously as data and patterns change
- Rebuild or drop ineffective indexes; update statistics regularly

Q: What are the trade-offs of using indexing versus denormalization?

A:

1. Indexing

- Advantages:
  - Improves read performance and speeds up lookups.
  - Does not alter the logical data structure.
  - Can be added or removed easily without changing schema design.
- Disadvantages:
  - Slows down writes (INSERT, UPDATE, DELETE) because indexes must be maintained.
  - Consumes extra storage space.
  - Requires ongoing monitoring and maintenance to stay effective.

2. Denormalization

- Advantages:
  - Reduces the need for complex joins and repeated aggregations.
  - Boosts read performance for analytics and reporting workloads.
  - Simplifies query logic by combining related data into one table.
- Disadvantages:
  - Introduces redundancy, which can lead to data inconsistencies.
  - Increases storage requirements.
  - Complicates updates because redundant data must stay synchronized.

Summary:

- Indexing → boosts read performance without changing schema but increases write cost.
- Denormalization → simplifies queries and speeds up reads but sacrifices storage efficiency and data consistency.

Q: How does SQL handle recursive queries?

A:
1SQL 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:

```sql
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;
```

Q: What are the differences between transactional and analytical queries?

A:

Transactional Queries (OLTP)

- Perform short, real-time operations like inserts, updates, and deletes.
- Optimized for speed, concurrency, and maintaining data integrity.
- Work on small sets of current data.
- Common in banking, retail, and order-processing systems.

Analytical Queries (OLAP)

- Perform complex aggregations, joins, and calculations over large datasets.
- Optimized for read-heavy workloads and trend analysis.
- Work on historical or summarized data.
- Common in business intelligence, reporting, and forecasting.

Key Difference:

Transactional queries keep operational systems running efficiently. Analytical queries extract insights and patterns from large data volumes to support strategic decisions.

Q: How can you ensure data consistency across distributed databases?

A:

1. Distributed Transactions
Use a two-phase commit (2PC) protocol so that all databases either commit or roll back together. This guarantees atomicity but can reduce performance and scalability.

2. Eventual Consistency
Adopt eventual consistency when strict synchronization isn't required. Data updates propagate asynchronously and converge to a consistent state over time, improving availability.

3. Conflict Resolution
Apply versioning, timestamps, or last-write-wins rules to detect and resolve data conflicts when multiple nodes update the same record.

4. Data Replication and Synchronization
Implement reliable replication mechanisms to ensure updates in one database are propagated accurately to others. Use change-data-capture (CDC) or message queues for synchronization.

5. Regular Audits and Validation
Schedule data verification tasks to detect mismatches between replicas and correct them automatically or manually.

Summary:
Choose between strong consistency (2PC) and eventual consistency based on system needs. Combine replication, conflict handling, and regular validation to keep distributed data accurate and reliable.

Q: What is the purpose of the SQL PIVOT operator?

A:

The PIVOT operator converts row values into columns, allowing you to reshape and summarize data for easier analysis and reporting. It's often used to display categories or time periods as columns instead of rows.

Example — Transforming Yearly Sales Data:

```
SELECT product_id, [2021], [2022]
FROM (
    SELECT product_id, YEAR(sale_date)
AS sale_year, amount
    FROM sales
) AS src
PIVOT (
    SUM(amount)
    FOR sale_year IN ([2021], [2022])
) AS pvt;
```

How It Works:

1. The inner query selects the base data.
2. The PIVOT operator aggregates values (using SUM, AVG, etc.) based on the specified column.
3. Each unique value in the pivot column becomes a new column in the result set.

Use Case:

Ideal for turning transactional data into cross-tab reports, such as displaying months, years, or categories as columns.

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

A:

Bitmap Index

- Represents each distinct column value as a bit array (bitmap).
- Each bit corresponds to a row and indicates whether the row has that value.
- Performs fast logical operations (AND, OR, NOT) across columns.
- Best for low-cardinality columns such as gender, status, or yes/no fields.

B-tree Index

- Stores data in a balanced tree where keys are sorted and linked to their corresponding rows.
- Allows quick lookups, inserts, and range scans.
- Best for high-cardinality columns like IDs, timestamps, or prices.

Key Difference

- Bitmap indexes are optimized for few distinct values and boolean filtering.
- B-tree indexes are optimized for unique or wide-ranging data and range queries.

Example Use:

- Bitmap index on is_active for fast filtering of true/false values.
- B-tree index on order_date for efficient range queries over time.