Amazon SQL and DSA Interview Questions (0-3 Years) 10-20 LPA

SQL Questions

1. Write a query to find duplicate rows in a table.

To detect duplicates, identify columns that should be unique and group by them.

Example:

```
SELECT column1, column2, COUNT(*) AS count
FROM your_table
GROUP BY column1, column2
HAVING COUNT(*) > 1;
```

Explanation:

- GROUP BY combines rows with the same values in the specified columns.
- HAVING COUNT(*) > 1 filters those combinations that occur more than once, indicating duplicates.

Tip: Add ROW_NUMBER() or RANK() with CTE to highlight or delete duplicates if needed.

2. Explain the difference between INNER JOIN and OUTER JOIN with examples.

INNER JOIN:

Returns only **matching** records from both tables.

```
SELECT e.name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

- Output: Only employees who belong to a department.

LEFT OUTER JOIN:

Returns **all records from the left** table, and matching records from the right table. If no match, NULL is returned.

```
SELECT e.name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
```

- Output: All employees, with department info where available.

RIGHT OUTER JOIN:

Returns **all records from the right** table, and matching records from the left.

FULL OUTER JOIN:

Returns **all records from both tables**, matching where possible.
Key Difference:

- INNER JOIN = intersection (matched data only)
- OUTER JOIN = union + NULLs (matched + unmatched data)

3. Write a query to fetch the second-highest salary from an employee table.

Option 1: Using DISTINCT, ORDER BY, and LIMIT (MySQL/PostgreSQL)

```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

Option 2: Using subquery (Generic SQL)

```
SELECT MAX(salary)
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

Explanation:

- The subquery fetches the highest salary.
- The outer query finds the maximum salary **less than** the highest — giving the second-highest.

4. How do you use GROUP BY and HAVING together? Provide an example.

Use GROUP BY to group data and HAVING to filter **aggregated results** (unlike WHERE, which filters raw rows).

```
SELECT department_id, COUNT(*) AS emp_count
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;
```

Explanation:

- Groups employees by department.
- Filters groups where the count of employees is **more than 5**.

5. Write a query to find employees earning more than their managers.

Assume the table employees has:
emp_id, name, salary, manager_id

```
SELECT e.name AS employee_name, e.salary, m.name AS manager_name, m.salary AS
manager_salary
FROM employees e
JOIN employees m ON e.manager_id = m.emp_id
WHERE e.salary > m.salary;
```

Explanation:

- Self-join: matches employees (e) with their managers (m).
- Filters those where employee's salary > manager's salary.

6. What is a window function in SQL? Provide examples of ROW_NUMBER and RANK.

Definition:

A **window function** performs calculations **across a set of table rows** related to the current row — without collapsing rows like GROUP BY.

Syntax:

```
FUNCTION_NAME() OVER (PARTITION BY column ORDER BY column)
```

Example: ROW_NUMBER()

Assigns a unique sequential number to each row **within a partition**.

```
SELECT name, department, salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_num
FROM employees;
```

- Each employee within the same department gets a row number based on salary rank (highest first).

Example: RANK()

Assigns **the same rank** to rows with **equal values**, but skips the next rank(s).

```
SELECT name, department, salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank_num
FROM employees;
```

- If 2 employees have the same salary, both get rank 1, and the next gets rank 3.

7. Write a query to fetch the top 3 performing products based on sales.

Assume table sales_data has:
product_id, product_name, total_sales

```
SELECT product_id, product_name, total_sales
FROM sales_data
ORDER BY total_sales DESC
LIMIT 3;
```

Alternate using RANK() (if ties matter):

```
SELECT product_id, product_name, total_sales
FROM (
  SELECT *, RANK() OVER (ORDER BY total_sales DESC) AS rank_num
  FROM sales_data
) ranked_sales
WHERE rank_num <= 3;
```

8. Explain the difference between UNION and UNION ALL.

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps all rows, including duplicates
Performance	Slower (because of sorting)	Faster (no de-duplication)
Use case	When you want distinct rows	When duplicates are meaningful

Example:

```
SELECT city FROM customers
UNION
SELECT city FROM vendors;
```

→ Returns a unique list of cities.

```
SELECT city FROM customers
UNION ALL
SELECT city FROM vendors;
```

→ Returns **all cities**, including duplicates.

9. How do you use a CASE statement in SQL? Provide an example.

CASE lets you write conditional logic in SQL (similar to IF/ELSE).

```
SELECT name, salary,
      CASE
        WHEN salary >= 100000 THEN 'High'
        WHEN salary >= 50000 THEN 'Medium'
        ELSE 'Low'
      END AS salary_category
FROM employees;
```

Explanation:

- Assigns a category based on salary value.
- Works inside SELECT, WHERE, ORDER BY, etc.

10. Write a query to calculate the cumulative sum of sales.

Assume table sales has:
order_date, product_id, sales_amount

```
SELECT order_date, product_id, sales_amount,
      SUM(sales_amount) OVER (PARTITION BY product_id ORDER BY order_date) AS
cumulative_sales
FROM sales;
```

Explanation:

- SUM(...) OVER (...) calculates a **running total** per product based on order date.
- PARTITION BY groups by product, and ORDER BY ensures the accumulation follows chronological order.

11. What is a CTE (Common Table Expression), and how is it used?

Definition:

A **CTE (Common Table Expression)** is a temporary, named result set that you can reference within a SQL query.
It improves readability and simplifies complex subqueries or recursive logic.

Syntax:

```
WITH cte_name AS (
  SELECT ...
)
SELECT * FROM cte_name;
```

Example – Filter top-paid employees using CTE:

```
WITH HighEarners AS (
  SELECT emp_id, name, salary
  FROM employees
  WHERE salary > 100000
)
SELECT * FROM HighEarners;
```

Benefits:

- Reusable and readable
- Allows recursion (e.g., hierarchical data)
- Avoids repeating subqueries

12. Write a query to identify customers who have made transactions above $5,000 multiple times.

Assume transactions table has:
customer_id, transaction_amount

```
SELECT customer_id, COUNT(*) AS high_value_txns
FROM transactions
WHERE transaction_amount > 5000
GROUP BY customer_id
HAVING COUNT(*) > 1;
```

Explanation:

- Filters high-value transactions (> $5000).
- Groups them by customer.
- Returns customers who've done this **more than once**.

13. Explain the difference between DELETE and TRUNCATE commands.

Feature	DELETE	TRUNCATE
Removes rows	Yes (can use WHERE condition)	Yes (removes all rows)
WHERE supported?	Yes	No
Logging	Logs each deleted row (slower)	Minimal logging (faster)
Rollback	Can be rolled back (if within transaction)	Can be rolled back (in some RDBMS)
Identity reset	Retains identity	Resets identity (in most DBs)
Use case	Partial deletion or audit trail needed	Full data wipe without audit needed

14. How do you optimize SQL queries for better performance?

Here are **key SQL optimization techniques**:

1. Use SELECT only required columns

```
-- Bad
SELECT * FROM orders;
-- Good
SELECT order_id, customer_id FROM orders;
```

2. Create proper indexes

- Index frequently used columns in JOIN, WHERE, ORDER BY.

3. Avoid functions on indexed columns

```
-- Slower (cannot use index)
WHERE YEAR(order_date) = 2024
-- Better
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31'
```

4. Use EXISTS instead of IN (for subqueries)

```
-- Prefer EXISTS (better for large datasets)
SELECT name FROM customers c
WHERE EXISTS (
  SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id
);
```

5. Avoid unnecessary joins or nested subqueries

6. Use appropriate data types and avoid implicit conversions

7. Analyze execution plans (EXPLAIN or EXPLAIN ANALYZE)

15. Write a query to find all customers who have not made

any purchases in the last 6 months.

Assume:

- customers(customer_id, name)
- transactions(customer_id, transaction_date)

```
SELECT c.customer_id, c.name
FROM customers c
LEFT JOIN transactions t
  ON c.customer_id = t.customer_id
     AND t.transaction_date >= CURRENT_DATE - INTERVAL '6 months'
WHERE t.customer_id IS NULL;
```

Explanation:

- LEFT JOIN includes all customers.
- WHERE t.customer_id IS NULL ensures the customer had **no purchase in the last 6 months**.

```
WHERE customer_id IN (
    SELECT customer_id FROM customers WHERE tier = 'Premium'
)
GROUP BY customer_id, txn_month, category
ORDER BY customer_id, txn_month, total_spend DESC;
```

DSA Questions

1. Two Sum (Array / Hash Map)

Problem:

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Solution:

1. Use a hash map (dict) to store elements as you iterate through the array.
2. For each element x, check if target - x already exists in the map:
   If yes → Found the pair.
   If no → Store x in the map with its index.
3. Return the pair of indices.

```
def twoSum(nums, target):
    seen = {}
    for i, num in
enumerate(nums):
        diff = target - num
        if diff in seen:
```

```
            return [seen[diff], i]
        seen[num] = i
    return []
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(n)

2. Best Time to Buy and Sell Stock I (Array / One Pass)

Problem:

You are given an array prices where prices[i] is the price of a stock on day i.
You want to maximize your profit by choosing **one day to buy** and **a later day to sell**.
Return the **maximum profit**. If you cannot achieve any profit, return 0.

Solution Outline:

Track the minimum price so far as you scan left → right.
At each day i, compute the profit if sold today: prices[i] - min_price_so_far.
Update the max profit with this value.
Update min_price_so_far if the current price is smaller.
Answer is the max profit found (or 0 if negative).

Python Code Example (Binary Search approach):

```
def maxProfit(prices):
    min_price = float('inf')
    max_profit = 0
    for p in prices:
        if p < min_price:
            min_price = p
        else:
            max_profit =
max(max_profit, p - min_price)
    return max_profit
```

Time Complexity: O(n)

Space Complexity: O(1)

3. Valid Parentheses (Stack)

Problem:

Given a string s containing just the characters '(' , ')' , '{' , '}' , '[' , ']', determine if the input string is **valid**.

A string is valid if:

1. Open brackets are closed by the same type of brackets, and
2. Open brackets are closed in the correct order.

Solution Outline:

1. Use a **stack** to keep track of opening brackets.
2. Use a map pairs = {')':'(', ']':'[', '}':'{'} to match closing → opening.
3. Scan the string:
   If the char is an opening bracket, **push** to stack.
   If it's a closing bracket, check if the stack is non-empty and the top matches pairs[char].
   If not, return False.
   Else, **pop**.
4. After processing all chars, the string is valid iff the **stack is empty**.

```
def isValid(s):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}
    for ch in s:
        if ch in "([{":
            stack.append(ch)
        else:
            if not stack or stack[-1] != pairs.get(ch, '#'):
                return False
            stack.pop()
    return len(stack) == 0
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(n) (in worst case, all openings)

4. Maximum Subarray (Kadane's Algorithm)

Problem:
Given an integer array nums, find the **contiguous subarray** (containing at least one number) which has the **largest sum**, and return that sum.

Solution Outline:

Maintain two variables while scanning left → right:

- cur_sum: best sum ending at current index.
- best: best sum seen so far.

Transition:

- cur_sum = max(nums[i], cur_sum + nums[i]) (start new subarray vs extend)
- best = max(best, cur_sum)

Initialize both to the first element to handle all-negative arrays.
Return best.

Result

- The desired length is dp[n][m], where n = len(text1), m = len(text2)
- This fills the table in O(n·m) time and uses O(n·m) space.

```
def maxSubArray(nums):

    cur_sum = best =

nums[0]

    for x in nums[1:]:

        cur_sum = max(x,

cur_sum + x)

        best = max(best,

cur_sum)

    return best
```

Time Complexity: O(n)
Space Complexity: O(1)

5. Merge Intervals (Sorting + Greedy)

Problem:

Given an array of intervals intervals where each interval is represented as [start, end], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Solution Outline:

1. Sort the intervals by their **start time**.
2. Initialize a result list with the first interval.
3. For each next interval:
   If it **overlaps** with the last interval in the result (current_start <= last_end), merge them by updating the end time.
   Otherwise, append it as a new interval.
4. Return the result list.

```
def merge(intervals):

    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]



    for start, end in intervals[1:]:

        last_start, last_end = merged[-1]

        if start <= last_end:  # overlap

            merged[-1][1] = max(last_end, end)

        else:

            merged.append([start, end])

    return merged
```

Problem: Detect a Cycle in a Directed Graph (Graph / DFS + Recursion Stack

Solution Outline:

Use Depth-First Search (DFS) with a recursion stack (path_visited) to detect back edges:

1. Build adjacency list from edge list.
2. Initialize two boolean arrays of size V:
    - visited[] — tracks globally visited nodes.
    - path_visited[] — tracks nodes in the current DFS path.
3. For each vertex i from 0 to V-1:
    - If not visited, call dfs(i).
4. dfs(node):
    - Mark visited[node] = True and path_visited[node] = True.
    - For each neighbor nbr:
        - If nbr is unvisited, recursively call dfs(nbr); if True, return True.
        - Else if path_visited[nbr] is True, we've found a back edge, so return True.
    - On exit, set path_visited[node] = False and return False.
5. If any DFS returns True, a cycle exists; otherwise, no cycle.

This runs in O(V + E) time and O(V + E) space (for adjacency list and recursion tracking)

```
def hasCycle(V, edges):
    adj = [[] for _ in range(V)]
    for u, v in edges:
        adj[u].append(v)

    visited = [False] * V
    path_vis = [False] * V

    def dfs(u):
        visited[u] = True
        path_vis[u] = True
        for v in adj[u]:
            if not visited[v]:
                if dfs(v):
                    return True
            elif path_vis[v]:
                return True
        path_vis[u] = False
        return False

    for i in range(V):
        if not visited[i]:
            if dfs(i):
                return True
    return False
```

- **Time Complexity: O(V + E)**

Space Complexity: O(V + E) for adjacency list + O(V) recursion stack

Longest Palindromic Substring (Strings / Expand Around Center or DP)

Problem:

Given a string s, find the longest substring that reads the same forwards and backwards.

Solution Outline (Expand Around Center – $O(n^2)$ time, $O(1)$ space):

1. For each index i in the string, expand around:
    - One center: both left and right pointers start at i (odd-length palindromes).
    - Two centers: left = i, right = i+1 (even-length palindromes).
2. Define a helper to expand and return the longest palindrome around the given left/right.
3. Update the global maximum substring when a longer palindrome is found

```
def longestPalindrome(s):
    if not s: return ""
    start, end = 0, 0

    def expand(l, r):
        while l >= 0 and r < len(s) and s[l] == s[r]:
            l -= 1
            r += 1
        return l + 1, r - 1

    for i in range(len(s)):
        l1, r1 = expand(i, i)
        l2, r2 = expand(i, i + 1)
        if r1 - l1 > end - start:
            start, end = l1, r1
        if r2 - l2 > end - start:
            start, end = l2, r2

    return s[start:end + 1]
```

Time Complexity: $O(n^2)$ (due to nested expansion)

Space Complexity: $O(1)$

This method is often recommended in interview prep guides as the easiest performant solution

Implement Trie (Prefix Tree) (Trie / N-ary Tree)

Problem:

Design a Trie with the following operations:

- **insert(word)**: Inserts the string word.
- **search(word)**: Returns true if word is in the Trie.

- **startsWith(prefix)**: Returns true if any word in the Trie starts with prefix.

Solution Outline:

Trie Node Structure:

- Each node has:
  - children: A map (or array of size 26) pointing to next chars.
  - isWord: A boolean flag indicating end of a valid word.

Insert Operation (O(L)):

- Start at the root.
- For each character c in word:
  - If c not in current node's children → create a new node.
  - Move to the child node.
- After the last char, mark isWord = True.

Search Word (O(L)):

- Traverse nodes following each character.
- If a character isn't found → return false.
- At end, check isWord flag.

Search Prefix (O(L)):

- Similar traversal without checking isWord.
- If traversal completes, return true.

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.isWord = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        or c in word:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
        node.isWord = True

    def search(self, word):
```

```
        node = self.root
        for c in word:
            if c not in node.children:
                return False
            node = node.children[c]
        return node.isWord

    def startsWith(self, prefix):
        node = self.root
        for c in prefix:
            if c not in node.children:
                return False
            node = node.children[c]
        return True
```

- **Time Complexity:** O(L) per operation, where L = length of input string
- **Space Complexity:** O(N × L) in worst case, N = number of inserted words