SQL Interview questions(Medium to Hard level)

05 June 2025 18:35

1. How do you calculate the median salary in a department using SQL?
Calculating the median often involves window functions or subqueries, as there's no direct MEDIAN() aggregate function in all SQL databases (though some do have it).

- **Using NTILE or ROW_NUMBER (General Approach):**
  SQL
  ```
  WITH RankedSalaries AS (
      SELECT DepartmentID, Salary,
          ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary) AS RowNum,
          COUNT(*) OVER (PARTITION BY DepartmentID) AS TotalCount
      FROM Employees
  )
  SELECT DepartmentID, AVG(Salary) AS MedianSalary
  FROM RankedSalaries
  WHERE RowNum IN ((TotalCount + 1) / 2, (TotalCount + 2) / 2)
  GROUP BY DepartmentID;
  ```

- ```
  SELECT department_id, AVG(salary) AS median_salary
  FROM (
      SELECT
          department_id,
          salary,
          ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary) as rn_asc,
          ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) as rn_desc
      FROM Employees
      WHERE salary IS NOT NULL -- Exclude NULL salaries if applicable
  ) AS ranked_salaries
  WHERE rn_asc BETWEEN rn_desc - 1 AND rn_desc + 1 -- This covers both even and odd counts
  GROUP BY department_id;
  ```

- ```
  -- A more common way for true median with odd/even handling:
  SELECT department_id,
      AVG(salary) AS median_salary
  FROM (
      SELECT
          department_id,
          salary,
          ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary) as rn,
          COUNT(*) OVER (PARTITION BY department_id) as cnt
      FROM Employees
  ) AS sub
  WHERE rn IN (CEIL(cnt / 2.0), FLOOR(cnt / 2.0) + 1) -- For odd, both will point to the same row. For even, to the two middle rows.
  GROUP BY department_id;
  ```

- **Using PERCENTILE_CONT (if available, e.g., SQL Server, Oracle, PostgreSQL):**
  SQL

  ```
  SELECT department_id,
      PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary) OVER (PARTITION BY
  ```

department_id) AS median_salary
FROM Employees;

(Note: PERCENTILE_CONT interpolates for even counts, PERCENTILE_DISC picks an existing value).

2. What's the difference between correlated and non-correlated subqueries?

- **Non-Correlated Subquery (Independent):**
  - Executes once and returns a result set to the outer query.
  - Does *not* depend on the outer query for its execution.
  - The outer query uses the result of the subquery.
  - Example: SELECT employee_name FROM Employees WHERE salary > (SELECT AVG(salary) FROM Employees);
- **Correlated Subquery (Dependent):**
  - Executes once for *each row* processed by the outer query.
  - References columns from the outer query in its WHERE clause.
  - Used for row-by-row processing, often checking for existence or aggregation based on the outer row's context.
  - Example: SELECT department_name FROM Departments d WHERE EXISTS (SELECT 1 FROM Employees e WHERE e.department_id = d.department_id AND e.salary > 100000);

3. How can you update data in one table based on values from another?

- **Using UPDATE...JOIN (SQL Server, MySQL, PostgreSQL):**
  SQL

  ```
  UPDATE T1
  SET T1.column_to_update = T2.value_from_other_table
  FROM Table1 T1
  JOIN Table2 T2 ON T1.join_column = T2.join_column
  WHERE T2.some_condition = 'value';
  ```
- **Using UPDATE...FROM (PostgreSQL):**
  SQL

  ```
  UPDATE Table1
  SET column_to_update = Table2.value_from_other_table
  FROM Table2
  WHERE Table1.join_column = Table2.join_column
    AND Table2.some_condition = 'value';
  ```
- **Using UPDATE...SET (SELECT...) (Oracle, also generally works in other SQL):**
  SQL

  ```
  UPDATE Table1
  SET column_to_update = (SELECT T2.value_from_other_table
                          FROM Table2 T2
                          WHERE Table1.join_column = T2.join_column)
  WHERE EXISTS (SELECT 1 FROM Table2 T2 WHERE Table1.join_column = T2.join_column);
  ```

4. What are window functions and how are they different from GROUP BY?

- **Window Functions:**
  - Perform calculations across a set of table rows that are related to the current row.
  - They *do not* collapse rows; they return a value for each row in the result set.
  - Syntax often includes OVER(), PARTITION BY, ORDER BY.
  - Examples: ROW_NUMBER(), RANK(), DENSE_RANK(), LAG(), LEAD(), NTILE(), SUM() OVER(), AVG() OVER().

    - Useful for: ranking, moving averages, calculating differences between consecutive rows, cumulative sums, etc.
- **GROUP BY:**
    - Aggregates rows based on one or more columns, collapsing them into a single summary row for each group.
    - Returns one row per group.
    - All non-aggregated columns in the SELECT list must be in the GROUP BY clause.
    - Examples: COUNT(), SUM(), AVG(), MIN(), MAX().
    - Useful for: total sales per region, average salary per department, count of employees per job title.
- **Key Difference:** Window functions retain individual row details while performing calculations, whereas GROUP BY summarizes and reduces the number of rows. You can use both in the same query.

5. How do you find gaps in a sequence (like missing invoice numbers)?

- **Using LEFT JOIN and IS NULL:**
  SQL

```
SELECT s.invoice_number + 1 AS missing_start,
     MIN(t.invoice_number) - 1 AS missing_end
FROM (
  SELECT invoice_number
  FROM Invoices
) AS s
LEFT JOIN Invoices AS t
  ON s.invoice_number + 1 = t.invoice_number
WHERE t.invoice_number IS NULL;
```

- **Using LAG() or LEAD() (more robust for ranges):**
  SQL

```
SELECT
  invoice_number AS start_of_gap,
  next_invoice_number AS end_of_gap
FROM (
  SELECT
    invoice_number,
    LEAD(invoice_number, 1) OVER (ORDER BY invoice_number) AS next_invoice_number
  FROM Invoices
) AS sub
WHERE next_invoice_number IS NOT NULL AND next_invoice_number - invoice_number > 1;
```

  This shows the last invoice number *before* a gap and the first invoice number *after* a gap.
- **Using a Numbers/Series Table (very flexible):** If you have a table Numbers with integers from 1 to N:
  SQL

```
SELECT n.number AS missing_invoice
FROM Numbers n
LEFT JOIN Invoices i ON n.number = i.invoice_number
WHERE i.invoice_number IS NULL
 AND n.number BETWEEN (SELECT MIN(invoice_number) FROM Invoices) AND (SELECT
MAX(invoice_number) FROM Invoices);
```

6. How would you delete every nth row from a table?

This typically requires identifying rows using a numbering scheme and then deleting them. This is often done by adding a temporary row number or using a window function in a CTE.

- **Using ROW_NUMBER() with a CTE (SQL Server, PostgreSQL, Oracle):**
  SQL

  ```
  WITH NumberedRows AS (
    SELECT
      *,
      ROW_NUMBER() OVER (ORDER BY some_unique_column) AS rn
    FROM YourTable
  )
  DELETE FROM NumberedRows
  WHERE rn % N = 0; -- Replace N with the desired number (e.g., 3 for every 3rd row)
  ```

  - **Important:** ORDER BY some_unique_column is crucial to define "every nth row" consistently. If your table doesn't have a natural ordering, the result might be non-deterministic.
  - Some databases might not allow DELETE FROM CTE. In such cases, you might need to use a subquery or a temporary table.
- **Alternative (if DELETE FROM CTE is not supported directly):**
  SQL

  ```
  DELETE FROM YourTable
  WHERE some_unique_column IN (
    SELECT some_unique_column
    FROM (
      SELECT
        some_unique_column,
        ROW_NUMBER() OVER (ORDER BY some_unique_column) AS rn
      FROM YourTable
    ) AS numbered_rows
    WHERE rn % N = 0
  );
  ```

7. How can you transpose columns into rows (unpivot) in SQL?

- **Using UNION ALL (most common and portable):**
  SQL

  ```
  SELECT ID, 'ColumnA' AS Category, ColumnA AS Value FROM YourTable
  UNION ALL
  SELECT ID, 'ColumnB' AS Category, ColumnB AS Value FROM YourTable
  UNION ALL
  SELECT ID, 'ColumnC' AS Category, ColumnC AS Value FROM YourTable;
  ```
- **Using UNPIVOT (SQL Server, Oracle):**
  SQL

  ```
  SELECT ID, Category, Value
  FROM YourTable
  UNPIVOT (Value FOR Category IN (ColumnA, ColumnB, ColumnC)) AS unpivoted_table;
  ```
- **Using LATERAL or CROSS APPLY (PostgreSQL LATERAL, SQL Server CROSS APPLY):**
  SQL

  ```
  -- PostgreSQL (LATERAL)
  ```

```
SELECT t.ID, u.Category, u.Value
FROM YourTable t, LATERAL (VALUES
  ('ColumnA', t.ColumnA),
  ('ColumnB', t.ColumnB),
  ('ColumnC', t.ColumnC)
) AS u(Category, Value);

-- SQL Server (CROSS APPLY)
SELECT t.ID, u.Category, u.Value
FROM YourTable t
CROSS APPLY (VALUES ('ColumnA', t.ColumnA), ('ColumnB', t.ColumnB), ('ColumnC', t.ColumnC) ) AS u(Category, Value); ```
```

8. What's the purpose of the COALESCE function and how is it different from ISNULL or NVL?

- **COALESCE Purpose:**
  - Returns the first non-NULL expression in a list of expressions.
  - It's useful for providing default values or combining multiple nullable columns where you want the first available non-NULL value.
  - Example: COALESCE(column1, column2, 'Default Value')
- **Differences:**
  - **COALESCE:**
    - Standard SQL function (ANSI SQL).
    - Can take multiple arguments (e.g., COALESCE(expr1, expr2, expr3, ..., exprN)).
    - Evaluates arguments in order and stops at the first non-NULL.
  - **ISNULL (SQL Server):**
    - Microsoft SQL Server specific function.
    - Takes exactly two arguments: ISNULL(check_expression, replacement_value).
    - Replaces NULL check_expression with replacement_value.
    - Return type of replacement_value determines the return type of the function.
  - **NVL (Oracle):**
    - Oracle specific function.
    - Takes exactly two arguments: NVL(expr1, expr2).
    - If expr1 is NULL, returns expr2; otherwise, returns expr1.
    - The data types of expr1 and expr2 must be compatible.
- **Key takeaway:** COALESCE is the most versatile and portable, while ISNULL and NVL are database-specific and typically limited to two arguments.

9. How do you calculate the difference in days, months, or years between two dates?

The exact function varies by database system.

- **SQL Server:**
  - Days: DATEDIFF(day, date1, date2)
  - Months: DATEDIFF(month, date1, date2)
  - Years: DATEDIFF(year, date1, date2)
  - Example: SELECT DATEDIFF(day, '2023-01-01', '2023-01-10'); (returns 9)
- **MySQL:**
  - Days: DATEDIFF(date2, date1)
  - Months: PERIOD_DIFF(DATE_FORMAT(date2, '%Y%m'), DATE_FORMAT(date1, '%Y%m')) (approximates full months) or more complex calculations for exact months.
  - Years: TIMESTAMPDIFF(YEAR, date1, date2) or YEAR(date2) - YEAR(date1)
  - Example: SELECT DATEDIFF('2023-01-10', '2023-01-01'); (returns 9)
- **PostgreSQL:**
  - Difference in days: (date2 - date1) (returns interval type, cast to int for days) or EXTRACT(DAY FROM (date2 - date1))

    - Difference in months/years: Requires more complex interval calculations or AGE() function.
    - SELECT AGE('2023-01-10', '2023-01-01'); (returns an interval)
    - SELECT EXTRACT(DAY FROM '2023-01-10'::date - '2023-01-01'::date); (returns 9 as double precision)
- **Oracle:**
    - Days: date2 - date1 (returns number of days)
    - Months: MONTHS_BETWEEN(date2, date1)
    - Years: MONTHS_BETWEEN(date2, date1) / 12
    - Example: SELECT TRUNC(SYSDATE - (SYSDATE - 10)) FROM DUAL; (returns 10 days)

10. How do you write a SQL query to show cumulative percentage of sales per region?
This involves window functions.
SQL

```
SELECT Region, Sales,
    SUM(Sales) OVER (ORDER BY Region) / SUM(Sales) OVER () * 100 AS CumulativePercentage
FROM SalesData;
```

11. How do you rank items within partitions based on a custom sorting logic?
Use window functions with PARTITION BY and ORDER BY.
SQL

```
SELECT
  category,
  item_name,
  sales,
  ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS rn_sales, -- Rank by sales descending
  RANK() OVER (PARTITION BY category ORDER BY item_name ASC) AS rank_item_name, -- Rank by item name ascending
  DENSE_RANK() OVER (PARTITION BY category ORDER BY some_other_criteria DESC, item_name ASC) AS dense_rank_custom
FROM Products;
```

- PARTITION BY category: Divides the data into groups (partitions) based on the category.
- ORDER BY sales DESC: Defines the sorting logic within each partition for rn_sales.
- ROW_NUMBER(), RANK(), DENSE_RANK(): Different ranking functions (choose based on tie-breaking requirements).

12. What's the difference between EXISTS and IN?
Both are used in subqueries but have different performance characteristics and use cases.

- **IN:**
    - Evaluates the subquery first, which returns a list of values.
    - The outer query then checks if a value exists *within* that list.
    - Best when the subquery returns a relatively small list of distinct values.
    - Can perform poorly with large subquery result sets as it may involve a large lookup.
    - Handles NULLs differently: If the subquery result contains NULL, NOT IN can behave unexpectedly.
    - Example: SELECT employee_name FROM Employees WHERE department_id IN (SELECT department_id FROM Departments WHERE location = 'New York');
- **EXISTS:**
    - Acts as a boolean operator, returning TRUE if the subquery returns *any* rows, and FALSE otherwise.
    - Does not actually retrieve any values from the subquery; it just checks for existence.

    - Correlated subqueries often use EXISTS.
    - Generally performs better than IN when the subquery deals with large datasets or when you only need to check for the presence of a matching row.
    - Handles NULLs transparently (doesn't get tripped up by them).
    - Example: SELECT department_name FROM Departments d WHERE EXISTS (SELECT 1 FROM Employees e WHERE e.department_id = d.department_id);
- **When to use which:**
    - Use EXISTS when you only need to check for the presence of related rows (typically for correlated subqueries).
    - Use IN when you need to match against a specific list of values.
    - For performance, EXISTS is often preferred over IN when the subquery might return a large number of rows, as EXISTS can stop evaluating as soon as it finds the first match.

13. How can you detect and prevent deadlocks in SQL queries?

- **Detecting Deadlocks:**
    - **Database Logs/Monitoring Tools:** Most database systems (SQL Server, Oracle, MySQL, PostgreSQL) have built-in mechanisms to detect deadlocks. They typically log deadlock events, often providing information about the queries involved, the resources being locked, and the victim chosen (the transaction that was rolled back).
        - SQL Server: SQL Server Profiler, Extended Events, sys.dm_tran_locks, sys.dm_os_wait_stats.
        - Oracle: Alert log, V$SESSION_WAIT, V$LOCK.
        - MySQL: SHOW ENGINE INNODB STATUS; (for InnoDB).
        - PostgreSQL: Log files, pg_stat_activity.
    - **Application-level logging:** Your application can log errors that indicate a deadlock occurred (e.g., specific error codes).
- **Preventing Deadlocks:**
    - **Access objects in the same order:** This is the most crucial strategy. If all transactions acquire locks on resources (tables, rows) in a consistent, predefined order, deadlocks are significantly reduced.
    - **Keep transactions short and concise:** The less time a transaction holds locks, the lower the chance of conflict.
    - **Reduce lock granularity (if possible):** Use row-level locks instead of table-level locks when appropriate.
    - **Use appropriate isolation levels:**
        - READ COMMITTED (default for many DBs) often balances concurrency and consistency.
        - SERIALIZABLE or REPEATABLE READ increase consistency but reduce concurrency, making deadlocks more likely.
    - **Avoid user interaction within transactions:** Don't hold locks while waiting for user input.
    - **Index appropriately:** Well-indexed tables allow queries to find data quickly, reducing the time locks are held.
    - **Use NOLOCK (SQL Server) or READ UNCOMMITTED isolation level (with caution):** This allows dirty reads and can avoid deadlocks, but sacrifices data consistency. Use *only* for reporting or non-critical reads.
    - **Implement retry logic:** In your application, if a transaction fails due to a deadlock, retry the transaction. The database will typically pick one transaction as a "deadlock victim" and roll it back. Retrying the victim transaction is a common strategy.
    - **Consider database-specific locking hints:** Some databases offer hints (e.g., ROWLOCK, PAGLOCK in SQL Server) to control locking behavior, but use them carefully and only when you fully understand their implications.

14. How would you handle slowly changing dimensions in SQL?

Slowly Changing Dimensions (SCDs) are a common data warehousing concept for managing how attribute values of a dimension change over time.

- **SCD Type 0: Retain Original**
  - Simply keep the original value. No history is preserved.
  - Example: Customer's birth date (doesn't change).
- **SCD Type 1: Overwrite**
  - The new value overwrites the old value. History is lost.
  - Simple to implement.
  - Example: Correcting a misspelled name.
  - Implementation: UPDATE DimensionTable SET attribute = newValue WHERE key = ...;
- **SCD Type 2: Add New Row (History Preservation)**
  - The most common type. A new row is added to the dimension table to capture the change, and the old row is marked as inactive or given an end date.
  - Preserves full history.
  - Requires additional columns:
    - EffectiveStartDate
    - EffectiveEndDate (or CurrentFlag/IsCurrent)
    - A surrogate key (auto-incrementing ID) for each version of the dimension record.
  - **Implementation Steps:**
    1. **Check for changes:** Compare incoming data with the current active record in the dimension table.
    2. **If no change:** No action or update LastUpdatedDate.
    3. **If change:**
       - Update the EffectiveEndDate (and set CurrentFlag to FALSE) of the *old* record.
       - Insert a *new* record with the new attribute values, EffectiveStartDate set to the current date, EffectiveEndDate set to a far-future date (or NULL), and CurrentFlag set to TRUE.
  - Example: Customer's address changes.
- **SCD Type 3: Add New Column (Limited History)**
  - Adds a new column to store the "previous" value of a specific attribute.
  - Only stores one previous value.
  - Less common, but useful for very specific scenarios where only the immediate previous state is needed.
  - Example: Customer.CurrentAddress, Customer.PreviousAddress.
- **SCD Type 4: Add History Table**
  - Maintains the current data in the dimension table and moves historical data to a separate history table.
  - Useful when the dimension table is frequently queried for current data, and historical queries are less common or performance-critical.
- **SCD Type 6: Hybrid (combining 1, 2, and 3)**
  - Often a combination of Type 1 (for non-tracked attributes), Type 2 (for core historical attributes), and Type 3 (for specific attributes that only need one previous state).

15. How do you detect circular references in a hierarchical table structure?

Circular references occur when a parent-child relationship eventually leads back to a node already visited in the same path (e.g., A -> B -> C -> A).

```
WITH RecursiveCTE AS (
    SELECT ID, ParentID
    FROM Hierarchy
    UNION ALL
    SELECT h.ID, h.ParentID
    FROM Hierarchy h
```

```
    JOIN RecursiveCTE r ON h.ParentID = r.ID
)
SELECT ID, ParentID
FROM RecursiveCTE
WHERE ID = ParentID;
```

- **Graph Databases:** For extremely complex or frequently changing hierarchical structures, a graph database (like Neo4j) is often a much better solution as they are optimized for graph traversal and cycle detection.

16. What's the difference between a clustered and non-clustered index?

Both are used to improve query performance by providing quick data lookup, but they differ in how they store and organize the data.

- **Clustered Index:**
  - **Physical Storage:** Determines the physical order in which the data rows are stored on disk.
  - **One per table:** A table can have only one clustered index because the data rows can only be sorted in one physical order.
  - **Data is the leaf level:** The leaf level of a clustered index *is* the actual data rows of the table.
  - **Faster for range scans:** Excellent for queries that retrieve data within a range or those that require sorted data.
  - **Impact on Inserts/Updates:** Inserts and updates can be slower if they require reordering of physical data pages.
  - **Primary Key:** By default, the Primary Key constraint creates a clustered index if one doesn't already exist.
  - Example: A phone book sorted by last name. The book *is* sorted by last name.
- **Non-Clustered Index:**
  - **Logical Order:** Creates a separate structure (like a b-tree) that contains the indexed columns and a pointer (row locator or clustered index key) to the actual data rows in the table.
  - **Multiple per table:** A table can have many non-clustered indexes.
  - **Pointers to data:** The leaf level of a non-clustered index contains the indexed column values and pointers to the actual data rows.
  - **Good for specific lookups:** Efficient for finding specific rows quickly.
  - **Impact on Inserts/Updates:** Inserts and updates might require updates to multiple non-clustered indexes, which adds overhead.
  - **Covering Index:** A non-clustered index that includes all the columns required by a query (either in the index key or as INCLUDE columns) is called a covering index. This allows the query to be satisfied entirely from the index without accessing the base table, leading to very fast performance.
  - Example: An index in a book. It's a separate list of keywords with page numbers, not the actual book content.

Key Differences Summary:

Feature	Clustered Index	Non-Clustered Index
Physical Order	Yes (determines data storage)	No (separate structure, pointers)
Number per Table	One	Multiple
Leaf Level	Actual data rows	Indexed columns + row locators
Best For	Range scans, sorted data retrieval	Point lookups, specific searches
Overhead	Less storage, higher insert/update cost on sorted column if data needs rearranging	More storage, higher insert/update cost if many indexes

Export to Sheets

17. How do you audit changes (INSERT/UPDATE/DELETE) in a table using SQL?

Auditing typically involves capturing who made the change, when, and what the old/new values were.

- **1. Triggers (Most Common SQL-based method):**
  - Create AFTER INSERT, AFTER UPDATE, and AFTER DELETE triggers on the table you want to audit.
  - Inside the trigger, insert the relevant information into an AuditLog table.
  - INSERTED and DELETED virtual tables are available within triggers to access new and old values.

    SQL

```
CREATE TABLE YourTable_Audit (
   AuditID INT IDENTITY(1,1) PRIMARY KEY,
   ActionType VARCHAR(10) NOT NULL, -- 'INSERT', 'UPDATE', 'DELETE'
   TableName VARCHAR(128) NOT NULL,
   RecordID INT, -- The primary key of the audited table
   ColumnName VARCHAR(128), -- For updates, which column changed
   OldValue VARCHAR(MAX),
   NewValue VARCHAR(MAX),
   ChangedBy VARCHAR(128) DEFAULT SUSER_SNAME(), -- SQL Server system function for
current user
   ChangedDate DATETIME DEFAULT GETDATE()
);

-- Example for UPDATE trigger (SQL Server syntax)
CREATE TRIGGER trg_YourTable_Audit_Update
ON YourTable
AFTER UPDATE
AS
BEGIN
   SET NOCOUNT ON;

INSERT INTO YourTable_Audit (ActionType, TableName, RecordID, ColumnName, OldValue,
NewValue, ChangedBy, ChangedDate)
   SELECT
      'UPDATE',
      'YourTable',
      d.ID, -- Assuming ID is the primary key
      'ColumnA', -- Specific column being audited
      d.ColumnA,
      i.ColumnA,
      SUSER_SNAME(),
      GETDATE()
   FROM DELETED d
   JOIN INSERTED i ON d.ID = i.ID
   WHERE d.ColumnA <> i.ColumnA OR (d.ColumnA IS NULL AND i.ColumnA IS NOT NULL) OR
(d.ColumnA IS NOT NULL AND i.ColumnA IS NULL); -- Check for actual change

-- You'd repeat the above SELECT...INSERT for each column you want to audit
END;

-- Example for INSERT trigger (SQL Server syntax)
CREATE TRIGGER trg_YourTable_Audit_Insert
```

```
ON YourTable
AFTER INSERT
AS
BEGIN
  SET NOCOUNT ON;

INSERT INTO YourTable_Audit (ActionType, TableName, RecordID, ColumnName, OldValue,
NewValue, ChangedBy, ChangedDate)
  SELECT
    'INSERT',
    'YourTable',
    i.ID,
    NULL, -- No specific column change for insert
    NULL,
    NULL, -- Or you can list all new values if desired
    SUSER_SNAME(),
    GETDATE()
  FROM INSERTED i;
END;

-- Example for DELETE trigger (SQL Server syntax)
CREATE TRIGGER trg_YourTable_Audit_Delete
ON YourTable
AFTER DELETE
AS
BEGIN
  SET NOCOUNT ON;

INSERT INTO YourTable_Audit (ActionType, TableName, RecordID, ColumnName, OldValue,
NewValue, ChangedBy, ChangedDate)
  SELECT
    'DELETE',
    'YourTable',
    d.ID,
    NULL, -- No specific column change for delete
    NULL, -- Or you can list all old values if desired
    NULL,
    SUSER_SNAME(),
    GETDATE()
  FROM DELETED d;
END;
```

- **2. Change Data Capture (CDC) / Change Tracking (Database Features):**
  - Many modern databases offer built-in features for auditing:
    - **SQL Server:** Change Data Capture (CDC) and Change Tracking. CDC is more robust, capturing full old/new values. Change Tracking just indicates that a row changed.
    - **Oracle:** Flashback Data Archive (FDA), Audit Vault and Database Firewall.
    - **PostgreSQL:** Logical Replication, or extensions like audit_table.
    - **MySQL:** Binary logs (Binlog) can be parsed, or third-party tools.
  - These are generally more performant and less intrusive than triggers, as they leverage the database's internal logging mechanisms.
- **3. Application-Level Auditing:**
  - The application explicitly logs changes before or after committing transactions.
  - Gives the most flexibility but puts the burden on developers.

- Less reliable if not implemented carefully (e.g., if an application bypasses the audit logic).

18. What is a recursive CTE, and where would you use it?

- **Recursive CTE (Common Table Expression):**
  - A powerful SQL feature that allows a query to refer to itself.
  - It's used to process hierarchical or tree-structured data iteratively.
  - A recursive CTE consists of two main parts:
    1. **Anchor Member:** The initial query that establishes the base result set (the "root" of the recursion).
    2. **Recursive Member:** A query that refers to the CTE itself. It performs operations on the results of the anchor member and subsequent recursive steps, stopping when no more rows are returned.
  - The two members are typically combined with UNION ALL.
  - **Syntax:**
    SQL

```
WITH RecursiveCTE AS (
  SELECT ID, ParentID
  FROM Hierarchy
  UNION ALL
  SELECT h.ID, h.ParentID
  FROM Hierarchy h
  JOIN RecursiveCTE r ON h.ParentID = r.ID
)
SELECT ID, ParentID
FROM RecursiveCTE;
```

  - Where would you use it?
  - **Hierarchical Data Traversal:** The most common use case.
    - **Organizational Charts:** Finding all employees under a manager, or finding the entire management chain above an employee.
    - **Bill of Materials (BOM):** Listing all sub-components of a product, or all parent assemblies of a component.
    - **Folder/File Structures:** Navigating directories and subdirectories.
    - **Comments/Replies:** Displaying threaded comments on a forum.
  - **Pathfinding in Graphs:** Although dedicated graph databases are better for complex graphs, simple pathfinding (e.g., finding all connections between two points) can be done.
  - **Generating Series/Sequences:** Creating a sequence of dates, numbers, or other values programmatically.
  - **Cycle Detection:** As discussed in question 15, identifying loops in hierarchical data.

19. How do you filter the top 3 products by revenue within each category?

This is a classic use case for window functions, specifically ranking functions.
SQL

```
SELECT Category, Product, Revenue,
    ROW_NUMBER() OVER (PARTITION BY Category ORDER BY Revenue DESC) AS Rank
FROM Products
WHERE Rank <= 3;
```

- ROW_NUMBER(): Assigns a unique rank to each product within a category. If there are ties in revenue, ROW_NUMBER() gives arbitrary but consistent ranks.
- RANK(): Gives the same rank to products with the same revenue (ties), and then skips the next

rank(s).
- DENSE_RANK(): Gives the same rank to products with the same revenue (ties), but does *not* skip ranks.

Choose ROW_NUMBER(), RANK(), or DENSE_RANK() based on how you want to handle ties.

20. What strategies can you use to reduce the execution time of complex joins?

Optimizing complex joins is critical for query performance.

- **1. Indexing (Most Important):**
  - **Join Columns:** Create indexes on the columns used in JOIN conditions. This allows the database to quickly find matching rows.
  - **WHERE Clause Columns:** Index columns used in WHERE clauses, especially those with high selectivity (filter out many rows).
  - **ORDER BY and GROUP BY Columns:** Indexing these can help avoid costly sorts.
  - **Covering Indexes:** If a non-clustered index includes all the columns needed by the query (both in the SELECT list and WHERE/JOIN clauses), the database might not need to access the base table at all, which is extremely fast.
- **2. Optimize WHERE Clauses (Filtering Early):**
  - Apply filters as early as possible (before joins if possible). This reduces the number of rows that need to be processed by the join operation.
  - Avoid using functions on indexed columns in WHERE clauses, as this can prevent index usage.
- **3. Choose Appropriate Join Types:**
  - **INNER JOIN:** If you only need rows where there's a match in both tables. Most efficient.
  - **LEFT JOIN/RIGHT JOIN:** Necessary when you need all rows from one side, even if there's no match on the other. Can be slower than INNER JOIN if not needed.
  - **FULL OUTER JOIN:** Least common, often very expensive, use only when absolutely necessary.
- **4. Denormalization (Strategic):**
  - In data warehousing or read-heavy systems, strategically duplicating data into fewer tables can reduce the number of joins needed. This comes at the cost of increased data redundancy and more complex data integrity management during writes.
- **5. Subqueries vs. Joins:**
  - Sometimes, rewriting a JOIN as a WHERE EXISTS or WHERE IN subquery (or vice-versa) can improve performance, depending on the query plan and data distribution. Use EXISTS over IN for large subquery result sets.
- **6. CTEs (Common Table Expressions) and Temporary Tables:**
  - **CTEs:** Can improve readability and allow the optimizer to potentially reuse intermediate results, but don't always guarantee performance improvements (they are often just syntactic sugar for subqueries).
  - **Temporary Tables (#TempTable or ##GlobalTempTable):** For very complex multi-step queries, breaking down the query into stages and populating temporary tables with filtered or aggregated data can be beneficial. Indexing these temporary tables can further optimize subsequent joins.
- **7. Materialized Views (Pre-aggregated Data):**
  - If you have a complex join that is frequently queried for the same summary data, a materialized view (or indexed view in SQL Server) can store the pre-computed results. The database automatically maintains this view, and queries against it are often much faster.
- **8. Analyze Query Plans:**
  - Use EXPLAIN (PostgreSQL, MySQL), SHOW PLAN (SQL Server), or EXPLAIN PLAN (Oracle) to understand how the database executes your query. This will show you which operations are most expensive (e.g., full table scans, sorts, hash joins, nested loop joins) and guide your optimization efforts.
- **9. Database Statistics:**

    - Ensure that database statistics are up-to-date. The optimizer relies on accurate statistics to choose the best execution plan. Outdated statistics can lead to inefficient plans.
- **10. Hardware and Configuration:**
    - Sufficient RAM, fast I/O (SSDs), and proper CPU allocation can significantly impact query execution time.
    - Database configuration parameters (e.g., memory allocation, query optimizer settings) can also be tuned.
- **11. Avoid SELECT *:**
    - Only retrieve the columns you need. Retrieving unnecessary columns increases data transfer and processing overhead.