

SQL and PL/SQL Interview Questions

1. Basic SQL Interview Questions

1. What is SQL?

- **Answer:** SQL (Structured Query Language) is a standardized language used to manage and manipulate relational databases. It allows querying, updating, and managing data stored in tables.

2. What are the different types of SQL statements?

- **Answer:** SQL statements are divided into five categories:
 - **DDL (Data Definition Language):** CREATE, ALTER, DROP, TRUNCATE, RENAME.
 - **DML (Data Manipulation Language):** SELECT, INSERT, UPDATE, DELETE, MERGE.
 - **DQL (Data Query Language):** SELECT.
 - **DCL (Data Control Language):** GRANT, REVOKE.
 - **TCL (Transaction Control Language):** COMMIT, ROLLBACK, SAVEPOINT.

3. What is the difference between DELETE and TRUNCATE?

- **Answer:**
 - **DELETE:** Removes rows from a table. It can be rolled back if wrapped in a transaction and can have a WHERE clause to delete specific rows.
 - **TRUNCATE:** Removes all rows from a table without logging individual row deletions. It cannot be rolled back and is faster but doesn't trigger DELETE triggers.

4. What is a JOIN in SQL? Explain different types of joins.

- **Answer:** A JOIN is used to retrieve data from two or more tables based on a related column.
 - **Inner Join:** Returns records that have matching values in both tables.
 - **Left (Outer) Join:** Returns all records from the left table and matched records from the right table. Unmatched rows will contain NULL.
 - **Right (Outer) Join:** Returns all records from the right table and matched records from the left table. Unmatched rows from the left table will be NULL.
 - **Full (Outer) Join:** Returns all records when there is a match in either left or right table. If there is no match, NULL values are returned.

5. What is a primary key?

- **Answer:** A primary key is a column (or set of columns) in a table that uniquely identifies each row. Primary keys must contain unique values and cannot contain NULL.

6. What is a foreign key?

- **Answer:** A foreign key is a column (or set of columns) in one table that refers to the primary key in another table. It establishes a relationship between the two tables.

7. What are constraints in SQL?

- **Answer:** Constraints enforce rules on the data in a table. Common constraints include:
 - **PRIMARY KEY:** Uniquely identifies each row.
 - **FOREIGN KEY:** Links two tables.
 - **UNIQUE:** Ensures all values in a column are unique.
 - **NOT NULL:** Ensures that a column cannot have **NULL** values.
 - **CHECK:** Ensures that all values in a column satisfy a specific condition.
 - **DEFAULT:** Sets a default value for a column when no value is provided.

8. What is a **GROUP BY** clause?

Answer: **GROUP BY** groups rows that have the same values in specified columns into summary rows. It is often used with aggregate functions like **COUNT()**, **SUM()**, **AVG()**, etc.

sql

Copy code

```
SELECT department, COUNT(*)  
FROM employees  
GROUP BY department;
```

○

9. What is the difference between **HAVING** and **WHERE**?

- **Answer:**
 - **WHERE:** Filters rows before aggregation.
 - **HAVING:** Filters groups after aggregation.
 - Example:

sql

Copy code

```
SELECT department, COUNT(*)  
FROM employees  
WHERE salary > 50000  
GROUP BY department  
HAVING COUNT(*) > 5;
```

○

10. What is normalization?

- **Answer:** Normalization is the process of organizing data in a database to avoid redundancy and improve data integrity. It involves breaking a large table into smaller tables and defining relationships between them. The normal forms (NF) are:
 - **1NF:** Eliminate duplicate columns, create separate tables for related data.
 - **2NF:** Remove partial dependencies.
 - **3NF:** Remove transitive dependencies.

2. Advanced SQL Interview Questions

1. What are subqueries?

- **Answer:** A subquery is a query nested inside another query. It can be in **SELECT**, **FROM**, **WHERE**, or **HAVING** clauses.

- **Example:**

sql

Copy code

```
SELECT employee_id, first_name
FROM employees
WHERE department_id = (SELECT department_id FROM departments WHERE
department_name = 'Sales');
```

○

2. What is a correlated subquery?

Answer: A correlated subquery is a subquery that uses values from the outer query. It is evaluated once for each row in the outer query.

sql

Copy code

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

○

3. What is a view in SQL?

Answer: A view is a virtual table based on the result set of a query. It does not store the data itself but provides a way to simplify complex queries.

sql

Copy code

```
CREATE VIEW employee_view AS
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary > 50000;
```

○

4. What is an index in SQL?

- **Answer:** An index improves query performance by allowing the database to find rows faster. It is like a lookup table for finding rows more quickly based on indexed columns.

- **Types:** Clustered Index, Non-clustered Index, Composite Index, Unique Index.

5. What is a **UNION** and how is it different from **UNION ALL**?

○ **Answer:**

- **UNION**: Combines the results of two queries but removes duplicate rows.
- **UNION ALL**: Combines the results of two queries but includes duplicates.

sql

Copy code

```
SELECT employee_id FROM employees
UNION
SELECT employee_id FROM managers;
```

○

6. What is a **CASE** statement in SQL?

Answer: A **CASE** statement is used to implement conditional logic in SQL.

sql

Copy code

```
SELECT employee_id,
       salary,
       CASE
         WHEN salary > 50000 THEN 'High'
         WHEN salary BETWEEN 30000 AND 50000 THEN 'Medium'
         ELSE 'Low'
       END AS salary_grade
FROM employees;
```

○

7. What are transactions in SQL?

- **Answer:** A transaction is a sequence of one or more SQL operations executed as a single unit. It ensures **ACID** properties (Atomicity, Consistency, Isolation, Durability).
 - **COMMIT**: Saves the transaction's changes.
 - **ROLLBACK**: Reverts changes since the last **COMMIT**.

8. What is the difference between **RANK()**, **DENSE_RANK()**, and **ROW_NUMBER()**?

○ **Answer:**

- **ROW_NUMBER()**: Assigns a unique row number to each row in the result set.
- **RANK()**: Assigns ranks to rows, with gaps in ranks when there are ties.
- **DENSE_RANK()**: Similar to **RANK()**, but without gaps in the ranking when there are ties.

sql

Copy code

```
SELECT employee_id, salary,  
       RANK() OVER (ORDER BY salary DESC) AS rank,  
       DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank,  
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num  
FROM employees;
```

○

9. What is a stored procedure?

Answer: A stored procedure is a set of SQL statements that can be stored in the database and executed as needed. Stored procedures allow reusability and can accept parameters.

sql

Copy code

```
CREATE PROCEDURE getEmployee(IN emp_id INT)  
BEGIN  
    SELECT * FROM employees WHERE employee_id = emp_id;  
END;
```

○

3. PL/SQL Interview Questions

1. What is PL/SQL?

- **Answer:** PL/SQL (Procedural Language for SQL) is Oracle's procedural extension of SQL. It combines SQL with procedural features like loops, conditions, and exception handling, allowing more advanced programming with SQL.

2. What are the key features of PL/SQL?

- **Answer:** Key features include:
 - Support for loops, conditions, and exception handling.
 - Use of **packages**, **procedures**, and **functions** for modular programming.
 - **Cursors** for handling query results.
 - **Triggers** to automatically perform actions in response to specific database events.

3. What are cursors in PL/SQL?

- **Answer:** A cursor is a pointer to a query result set. There are two types:
 - **Implicit cursor:** Automatically created for **SELECT** statements.
 - **Explicit cursor:** Declared explicitly to handle queries that return more than one row.

4. What are exceptions in PL/SQL?

Answer: Exceptions handle errors that occur during execution. There are predefined exceptions (like `NO_DATA_FOUND`, `TOO_MANY_ROWS`) and user-defined exceptions.

plsql

Copy code

```
BEGIN
    SELECT salary INTO emp_salary FROM employees WHERE employee_id =
100;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee not found.');
```

END;

○

5. What are triggers in PL/SQL?

Answer: Triggers are PL/SQL blocks that execute automatically in response to certain events on a table (e.g., `INSERT`, `UPDATE`, `DELETE`).

plsql

Copy code

```
CREATE OR REPLACE TRIGGER emp_insert_trigger
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    :NEW.salary := :NEW.salary * 1.10;
END;
```

○

6. What is a package in PL/SQL?

Answer: A package is a collection of related PL/SQL types, variables, procedures, and functions grouped together for modularity.

plsql

Copy code

```
CREATE PACKAGE employee_pkg AS
    PROCEDURE updateSalary(emp_id NUMBER, new_salary NUMBER);
END employee_pkg;
```

○

7. What is the difference between a function and a procedure in PL/SQL?

○ **Answer:**

- **Function:** Returns a value and can be used in SQL statements.
 - **Procedure:** Does not return a value but can return values via `OUT` parameters. It is primarily used to perform actions.
-

These questions will help you prepare for SQL and PL/SQL interviews, covering both basic and advanced concepts.

40

You said:

Difference between triggers?

ChatGPT said:

ChatGPT

Triggers are a type of stored procedure in SQL and PL/SQL that are automatically executed or "fired" when specific events occur in the database. There are several types of triggers, and they differ based on when they are executed and the kind of events they are associated with. Below is a breakdown of the **different types of triggers** and their key differences:

1. Row-level vs. Statement-level Triggers

- **Row-level Trigger:**

- Executed once **for each row** affected by the triggering event (e.g., **INSERT**, **UPDATE**, or **DELETE**).
- If 10 rows are updated, the trigger will execute 10 times (once per row).
- Syntax usually includes **FOR EACH ROW**.
- Useful when you need to track or log changes for each row separately.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER after_update_employee
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    -- Code to log each row update
END;
```

-

- **Statement-level Trigger:**

- Executed **once** for the entire SQL statement, regardless of how many rows are affected.
- If 10 rows are updated, the trigger will still execute only once.
- Does not include **FOR EACH ROW**.
- Useful for actions that only need to occur once, like logging or auditing an entire operation.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER after_update_employee_stmt
AFTER UPDATE ON employees
BEGIN
    -- Code to log once per UPDATE statement
END;
```

•

2. Before vs. After Triggers

- **Before Trigger:**
 - Fired **before** the triggering event (e.g., **INSERT**, **UPDATE**, or **DELETE**) is executed.
 - Used when you need to check or modify data **before** it is written to the database.
 - Can modify the values of the rows being inserted or updated.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER before_insert_employee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    :NEW.salary := :NEW.salary + 1000; -- Modify inserted salary
END;
```

•

- **After Trigger:**
 - Fired **after** the triggering event (e.g., **INSERT**, **UPDATE**, or **DELETE**) is executed.
 - Used when the action is dependent on the event already having occurred, like logging the final state of the data or updating related records.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER after_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    -- Code to log after row has been inserted
END;
```

•

3. INSTEAD OF Triggers

- **INSTEAD OF Trigger:**
 - Used **only on views**, and it tells the database what to do **instead** of the triggering SQL operation (e.g., **INSERT**, **UPDATE**, **DELETE**).
 - Since views are often read-only, **INSTEAD OF** triggers are used to define custom behavior for modifying the data in a view by redirecting the operation to the underlying base tables.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER instead_of_update_view
INSTEAD OF UPDATE ON employee_view
BEGIN
    -- Custom logic to update base table instead of view
END;
```

●

4. Compound Triggers

- **Compound Trigger:**
 - Introduced in Oracle 11g, it allows the declaration of multiple timing points in a single trigger. It combines multiple triggers into one, with sections for **BEFORE**, **AFTER**, and **FOR EACH ROW**.
 - Useful when managing the complexity of firing multiple triggers at different timing points.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER compound_employee_trigger
FOR INSERT OR UPDATE OR DELETE ON employees
COMPOUND TRIGGER
    -- Before statement section
    BEFORE STATEMENT IS
    BEGIN
        -- Executed before the triggering statement
    END BEFORE STATEMENT;

    -- Before each row section
    BEFORE EACH ROW IS
    BEGIN
        -- Executed before each row
```

```

END BEFORE EACH ROW;

-- After each row section
AFTER EACH ROW IS
BEGIN
    -- Executed after each row
END AFTER EACH ROW;

-- After statement section
AFTER STATEMENT IS
BEGIN
    -- Executed after the triggering statement
END AFTER STATEMENT;
END;

```

•

5. DML vs. DDL Triggers

- **DML (Data Manipulation Language) Trigger:**
 - Fired when a **DML** event like **INSERT**, **UPDATE**, or **DELETE** occurs on a table or view.
 - This is the most commonly used type of trigger for handling row or table changes.

Example:

plsql

Copy code

```

CREATE OR REPLACE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    -- Actions after data manipulation
END;

```

•

- **DDL (Data Definition Language) Trigger:**
 - Fired when a **DDL** event like **CREATE**, **ALTER**, **DROP**, **RENAME**, or **TRUNCATE** occurs in the database.
 - Often used for database-level auditing, such as tracking changes to the schema.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER ddl_audit_trigger
AFTER CREATE OR DROP OR ALTER
ON DATABASE
BEGIN
    -- Code to log schema changes
END;
```

-

6. Logon/Logoff and System-level Triggers

- **Logon Trigger:**

- Fired whenever a user logs into the database. Useful for enforcing security, initializing session-specific values, or auditing logins.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER logon_trigger
AFTER LOGON ON DATABASE
BEGIN
    -- Code to execute after user logon
END;
```

-

- **Logoff Trigger:**

- Fired when a user logs off from the database. Often used to clean up session-related data or perform session-specific auditing.

Example:

plsql

Copy code

```
CREATE OR REPLACE TRIGGER logoff_trigger
BEFORE LOGOFF ON DATABASE
BEGIN
    -- Code to execute before user logoff
END;
```

-

- **System-level Trigger:**

- Fired at the system or schema level, often for managing administrative tasks like auditing database-wide activities.

Key Differences Summary:

Type of Trigger	Purpose	When Fired	Execution Frequency
Row-level	Executes per row affected by DML statement	Before or after each row-level event	Once per affected row
Statement-level	Executes for the entire statement	Before or after a DML statement	Once per SQL statement
Before Trigger	Executes before a DML event occurs	Before INSERT , UPDATE , or DELETE	Before event execution
After Trigger	Executes after a DML event occurs	After INSERT , UPDATE , or DELETE	After event execution
INSTEAD OF Trigger	Used on views to define custom behavior	Instead of executing INSERT , UPDATE , or DELETE	Once per affected row (on views)
Compound Trigger	Allows multiple timing points in a single trigger	Allows BEFORE , AFTER , FOR EACH ROW , etc.	Varies (multiple points in one trigger)
DML Trigger	Fires after DML operations like INSERT , UPDATE , DELETE	After INSERT , UPDATE , or DELETE	Row-level or statement-level
DDL Trigger	Fires after DDL operations like CREATE , ALTER , DROP	After a DDL event	Statement-level
Logon/Logoff Trigger	Fires on user login or logout	After LOGON or before LOGOFF	Once per logon/logoff event

These types of triggers serve different purposes depending on when and why they are executed, ranging from handling individual row changes to system-wide events like user logins or schema modifications.

40

You said:

Types of cursors?

ChatGPT said:

ChatGPT

In SQL and PL/SQL, **cursors** are used to retrieve and manipulate query results one row at a time. Cursors are essentially pointers to the result set of a query, and they allow developers to fetch data row by row, process it, and perform specific actions. There are two main types of cursors: **implicit cursors** and **explicit cursors**. Within explicit cursors, there are also different types of cursor behaviors.

1. Implicit Cursors

- **Implicit Cursors** are automatically created by the database whenever a DML statement (e.g., `INSERT`, `UPDATE`, `DELETE`, `SELECT INTO`) is executed. They are managed by the database itself, and the developer does not have to declare or control them explicitly.
- These cursors are used for queries that return a single row or when you are performing operations like `INSERT`, `UPDATE`, or `DELETE`.

Key Points:

- Automatically managed by the database.
- Used when a SQL statement returns a single row.
- You don't have to declare or control them.
- Errors can be detected using predefined exceptions like `NO_DATA_FOUND` or `TOO_MANY_ROWS`.

Example:

plsql

Copy code

```
BEGIN
    -- Implicit cursor automatically handles this query
    SELECT salary INTO v_salary FROM employees WHERE employee_id =
101;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No record found.');
```

END;

2. Explicit Cursors

- **Explicit Cursors** are defined and controlled by the programmer. They are used when a query returns **more than one row**, and you need to process each row individually.
- With explicit cursors, you can fetch, loop through, and manipulate multiple rows returned by a query.

Key Points:

- Must be explicitly declared, opened, fetched, and closed by the developer.

- Used for queries that return multiple rows.
- Provide more control over the fetching process and allow row-by-row processing.

Explicit Cursor Operations:

1. **Declare** the cursor to define the SQL query.
2. **Open** the cursor to execute the query.
3. **Fetch** rows one at a time into variables.
4. **Close** the cursor once all rows have been processed.

Example:

plsql

Copy code

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, first_name, salary FROM employees;
    v_emp_id employees.employee_id%TYPE;
    v_first_name employees.first_name%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cursor; -- Open the cursor
    LOOP
        FETCH emp_cursor INTO v_emp_id, v_first_name, v_salary; --
Fetch a row
        EXIT WHEN emp_cursor%NOTFOUND; -- Exit the loop when no more
rows
        DBMS_OUTPUT.PUT_LINE(v_emp_id || ' ' || v_first_name || ' ' ||
v_salary);
    END LOOP;
    CLOSE emp_cursor; -- Close the cursor
END;
```

Subtypes of Explicit Cursors

2.1. Parameterized Cursors

- A **parameterized cursor** allows parameters to be passed into the cursor when it is opened, which helps make the cursor more flexible and reusable.

Example:

plsql

Copy code

```
DECLARE
    CURSOR emp_cursor (dept_id NUMBER) IS
```

```

        SELECT employee_id, first_name, salary FROM employees WHERE
department_id = dept_id;
        v_emp_id employees.employee_id%TYPE;
        v_first_name employees.first_name%TYPE;
        v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cursor(10); -- Pass department_id as a parameter
    LOOP
        FETCH emp_cursor INTO v_emp_id, v_first_name, v_salary;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_id || ' ' || v_first_name || ' ' ||
v_salary);
    END LOOP;
    CLOSE emp_cursor;
END;

```

2.2. Cursor FOR Loop

- The **FOR loop with cursors** automatically handles the opening, fetching, and closing of the cursor. It simplifies the code by eliminating the need to explicitly handle these operations.

Example:

plsql

Copy code

```

BEGIN
    FOR emp_record IN (SELECT employee_id, first_name, salary FROM
employees) LOOP
        DBMS_OUTPUT.PUT_LINE(emp_record.employee_id || ' ' ||
emp_record.first_name || ' ' || emp_record.salary);
    END LOOP;
END;

```

In this case, you do not need to explicitly declare, open, fetch, or close the cursor.

2.3. Ref Cursors (Cursor Variables)

- A **ref cursor** (or cursor variable) is a pointer to a cursor that allows more flexibility, such as passing a cursor as a parameter to a procedure or returning it from a function. This type of cursor can be dynamically associated with different queries at runtime.
- There are two types of ref cursors:

- **Strongly Typed Ref Cursor:** The structure (return type) of the cursor is predefined.
- **Weakly Typed Ref Cursor:** The cursor is more flexible and can be associated with different types of result sets.

Example (Weakly Typed Ref Cursor):

plsql

Copy code

```
DECLARE
    TYPE ref_cursor_type IS REF CURSOR;
    emp_ref_cursor ref_cursor_type;
    v_emp_id employees.employee_id%TYPE;
    v_first_name employees.first_name%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_ref_cursor FOR SELECT employee_id, first_name, salary
FROM employees;
    LOOP
        FETCH emp_ref_cursor INTO v_emp_id, v_first_name, v_salary;
        EXIT WHEN emp_ref_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_emp_id || ' ' || v_first_name || ' ' ||
v_salary);
    END LOOP;
    CLOSE emp_ref_cursor;
END;
```

Comparison of Cursor Types

Cursor Type	Definition	Use Case	Advantages
Implicit Cursor	Automatically managed by the database for INSERT , UPDATE , DELETE , or single-row queries.	Simple queries where only one row is affected or fetched.	Simplifies coding; you don't need to explicitly declare or manage it.
Explicit Cursor	Developer-defined cursors for queries that return more than one row.	When you need to process multiple rows or have more control.	Provides control over fetching rows and offers flexibility with parameterized and cursor FOR loops.

Parameterized Cursor	An explicit cursor that accepts parameters.	Reusable cursors where different parameters drive query results.	Enhances cursor reusability by allowing input parameters.
Cursor FOR Loop	A loop structure that automates cursor operations.	Simplifying explicit cursor handling in looping scenarios.	Reduces the need to explicitly open, fetch, and close the cursor.
Ref Cursor (Weak/Strong)	Dynamically associated cursors that can point to different result sets.	Passing cursors between subprograms or functions.	Offers dynamic query handling and the ability to return a cursor from a procedure or function for flexible logic.

Summary

- **Implicit Cursors** are automatically created and managed by the database when simple SQL statements (like **INSERT**, **UPDATE**, **DELETE**, or single-row **SELECT INTO**) are executed.
- **Explicit Cursors** are manually defined and controlled by the programmer, providing the flexibility to process multi-row query results, with options like parameterized cursors for added functionality.
- **Cursor FOR Loops** simplify cursor usage by automating the process of opening, fetching, and closing.
- **Ref Cursors** add flexibility by allowing dynamic cursor associations and enabling cursor passing between subprograms, making them highly useful for dynamic SQL queries.

Each type of cursor has specific use cases, depending on whether you need simplicity, control over multiple rows, reusability, or dynamic query handling.