**1. What is PL/SQL?**

PL/SQL is Oracle’s procedural extension to SQL. Unlike SQL, which is declarative and focuses on data retrieval and manipulation, PL/SQL allows developers to implement *procedural* logic, making it a powerful tool for writing complex business rules within the database. It supports variables, loops, conditionals, exception handling, and modular programming through procedures, functions, and packages.

**2. What is the basic structure of a PL/SQL block?**

A PL/SQL block is the fundamental unit of execution in PL/SQL, and it consists of four main sections:

* DECLARE (Optional): Used to define variables, constants, cursors, and user-defined types.
* BEGIN: The executable section where SQL queries and procedural statements are written.
* EXCEPTION (Optional): Handles runtime errors and exceptions to ensure graceful error recovery.
* END;: Marks the end of the block.

**3. What are the essential PL/SQL data types?**

PL/SQL supports various data types, categorized as follows:

* **Scalar Types**: Single-value types like NUMBER, VARCHAR2, DATE, BOOLEAN.
* **Composite Types:** Collections such as RECORD (custom structures) and TABLE/VARRAY (arrays).
* **Reference Types:** Pointers to database objects, like REF CURSOR for dynamic query processing.

**4. What are the basic control structures in PL/SQL?**

PL/SQL includes several control structures that help manage the flow of a program:

* **Loops:** These include LOOP, FOR LOOP, and WHILE LOOP, allowing repetitive execution of statements.
* **Conditional Statements:** These include IF and CASE statements, which execute different blocks of code based on conditions. The [**DECODE() function**](https://www.datacamp.com/tutorial/sql-decode) is another good example of a conditional that is worth studying.

**Intermediate PL/SQL Interview Questions**

Having covered the basic questions, now let's move on to some intermediate-level data structure interview questions. After testing your basic knowledge, interviewers are going to now test your technical proficiency in implementing and using PL/SQL concepts.

**5. What is the difference between stored procedures and functions?**

Stored procedures and functions are both reusable PL/SQL code blocks, but they serve rather different purposes.

[**Stored procedures**](https://www.datacamp.com/tutorial/sql-stored-procedure) are used to perform operations that do not return a value, such as inserting, updating, or deleting data. They are used for tasks that modify data or perform complex operations without returning a result.

For example, the procedure below updates the salary of the employee with the given employee\_id by adding the specified p\_increment such that the salary updates dynamically based on input parameters,

CREATE PROCEDURE update\_salary(p\_emp\_id NUMBER, p\_increment NUMBER) AS

BEGIN

-- Update the salary of the employee with the given ID

UPDATE employees

SET salary = salary + p\_increment

WHERE employee\_id = p\_emp\_id;

END;

[**Powered By**](https://www.datacamp.com/datalab)

Functions, on the other hand, return a value after performing operations. They are suitable for calculations or data retrieval that need to return a result.

The function below simplifies fetching an employee's salary, making it reusable in SQL queries or other procedures.

CREATE FUNCTION get\_employee\_salary(p\_emp\_id NUMBER) RETURN NUMBER AS

v\_salary NUMBER;

BEGIN

-- Retrieve the salary for the given employee ID

SELECT salary INTO v\_salary

FROM employees

WHERE employee\_id = p\_emp\_id;

-- Return the retrieved salary

RETURN v\_salary;

END;

[**Powered By**](https://www.datacamp.com/datalab)

**6. What are PL/SQL packages?**

PL/SQL packages are collections of related procedures, functions, and variables that encapsulate code for better organization and reusability. They consist of two parts:

* **Package Specification:** Declares public elements (procedures, functions, variables).
* **Package Body:** Contains the implementation details of the procedures and functions declared in the specification.

For example, the query below creates a package employee\_pkg that defines a procedure to raise an employee's salary and a function to retrieve the total number of employees, with their implementations to be provided in the package body.

-- Create a package named 'employee\_pkg'

CREATE PACKAGE employee\_pkg AS

-- Procedure to increase an employee's salary by a percentage

PROCEDURE raise\_salary(p\_emp\_id NUMBER, p\_percent NUMBER);

-- Function to return the total number of employees

FUNCTION get\_total\_employees RETURN NUMBER;

END employee\_pkg;

[**Powered By**](https://www.datacamp.com/datalab)

**7. What are PL/SQL triggers?**

Triggers are PL/SQL blocks that execute automatically in response to specific database events, such as insertions, updates, or deletions. Triggers are used to enforce business rules. They also are commonly used to perform auditing. They are categorized into:

* **Row-Level Triggers:** Execute once for each affected row.
* **Statement-Level Triggers:** Execute once per SQL statement, regardless of the number of rows affected.

The query below creates an AFTER UPDATE trigger trg\_salary\_audit on the employees table that logs salary changes into the salary\_audit table, capturing the employee ID, old and new salary, and the update timestamp.

-- Create or replace a trigger 'trg\_salary\_audit'

CREATE OR REPLACE TRIGGER trg\_salary\_audit

AFTER UPDATE OF salary ON employees -- Fires after salary updates in 'employees' table

FOR EACH ROW -- Executes for each updated row

BEGIN

-- Inserts old and new salary details into 'salary\_audit' table

INSERT INTO salary\_audit (employee\_id, old\_salary, new\_salary, change\_date)

VALUES (:OLD.employee\_id, :OLD.salary, :NEW.salary, SYSDATE);

END;

[**Powered By**](https://www.datacamp.com/datalab)

Study our tutorial on [**SQL Triggers**](https://www.datacamp.com/tutorial/sql-triggers) so that you will be ready in the event that triggers are brought up in the interview.

**8. What are the methods of exception handling in PL/SQL?**

PL/SQL provides error-handling mechanisms for stability and to prevent crashes. The types of exceptions include the following:

* **Predefined Exceptions:** Built-in exceptions like NO\_DATA\_FOUND, TOO\_MANY\_ROWS, and ZERO\_DIVIDE.
* **User-Defined Exceptions:** Custom exceptions are declared using EXCEPTION and raised using RAISE.

For example, the following PL/SQL block retrieves the salary of an employee with ID 100, raises a custom exception if the salary is below 1000, and handles possible errors, including missing employee records and unexpected exceptions.

DECLARE

v\_salary NUMBER; -- Variable to store the employee's salary

e\_low\_salary EXCEPTION; -- Custom exception for low salary

BEGIN

-- Retrieve salary of employee with ID 100

SELECT salary INTO v\_salary FROM employees WHERE employee\_id = 100;

-- Raise exception if salary is below 1000

IF v\_salary < 1000 THEN

RAISE e\_low\_salary;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN -- Handle case where employee ID is not found

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

WHEN e\_low\_salary THEN -- Handle custom low salary exception

DBMS\_OUTPUT.PUT\_LINE('Salary is below the allowed threshold.');

WHEN OTHERS THEN -- Handle any other unexpected errors

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END;

[**Powered By**](https://www.datacamp.com/datalab)

The best practices for exception handling include the following:

* Always anticipate potential exceptions.
* Use meaningful error messages.
* Log exceptions for auditing.
* Keep exception handling code concise and focused on recovery.

**9. How can you verify whether an UPDATE statement is executed or not?**

The SQL %NOTFOUND attribute can be used to determine whether or not the UPDATE statement successfully changed any records. If the last SQL statement run did not affect any rows, this variable returns TRUE.

For example, the query below updates the salary of employees in department 10 by increasing it by 10%, and then checks whether any rows were affected by the UPDATE statement using the %NOTFOUND attribute. If no rows were updated, it outputs a message saying, "No rows were updated." If rows were updated, it outputs the number of rows that were affected using the SQL%ROWCOUNT attribute.

DECLARE

-- Declare a variable to store the number of rows updated

rows\_updated INTEGER;

BEGIN

-- Perform an UPDATE statement on the 'employees' table

UPDATE employees

SET salary = salary \* 1.1

WHERE department\_id = 10;

-- Check if any rows were updated by using %NOTFOUND

IF SQL%NOTFOUND THEN

-- If no rows were updated, print a message

DBMS\_OUTPUT.PUT\_LINE('No rows were updated.');

ELSE

-- If rows were updated, print how many rows were affected

rows\_updated := SQL%ROWCOUNT; -- Store the number of rows updated

DBMS\_OUTPUT.PUT\_LINE(rows\_updated || ' rows were updated.');

END IF;

END;

/

[**Powered By**](https://www.datacamp.com/datalab)

**Advanced PL/SQL Interview Questions**

Let's now explore some advanced interview questions in case you are applying for a more senior role that requires more experience.

**10. What are the methods of optimizing performance in PL/SQL?**

Minimizing context switches between SQL and PL/SQL is crucial for optimizing performance. Each switch incurs overhead, which can slow down execution times, especially in situations involving frequent transitions between the two.

**11. How do you use bulk operations to minimize context switches?**

PL/SQL provides bulk processing techniques to optimize SQL-to-PL/SQL interaction by fetching or modifying multiple rows at once.

For example, the PL/SQL block below retrieves all employees from department 10 using BULK COLLECT into a collection and iterates through it to print each employee's name, improving performance by minimizing context switches between SQL and PL/SQL.

DECLARE

-- Define a table-type collection based on the 'employees' table structure

TYPE emp\_table IS TABLE OF employees%ROWTYPE;

v\_emps emp\_table; -- Declare a variable of this type

BEGIN

-- Bulk fetch employees from department 10 into the collection

SELECT \* BULK COLLECT INTO v\_emps FROM employees WHERE department\_id = 10;

-- Loop through the collection and print employee names

FOR i IN 1..v\_emps.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_emps(i).employee\_name);

END LOOP;

END;

[**Powered By**](https://www.datacamp.com/datalab)

Also, this PL/SQL block uses FORALL for bulk updating, increasing the salary by 10% for employees with IDs 101, 102, and 103.

DECLARE

-- Define a table-type collection for employee IDs

TYPE t\_emp\_ids IS TABLE OF employees.employee\_id%TYPE;

-- Initialize collection with specific employee IDs

v\_emp\_ids t\_emp\_ids := t\_emp\_ids(101, 102, 103);

BEGIN

-- Bulk update salaries by 10% for specified employee IDs

FORALL i IN 1..v\_emp\_ids.COUNT

UPDATE employees SET salary = salary \* 1.10 WHERE employee\_id = v\_emp\_ids(i);

END;

[**Powered By**](https://www.datacamp.com/datalab)

**12. What are dynamic SQL and Ref Cursors?**

Dynamic SQL allows executing SQL statements dynamically at runtime, which is useful when dealing with variable table names, columns, or query structures.

The following PL/SQL block uses dynamic SQL to count the number of rows in the employees table and prints the result. I like this kind of approach because it allows flexibility.

DECLARE

v\_table\_name VARCHAR2(50) := 'employees'; -- Store table name

v\_count NUMBER; -- Variable to hold row count

BEGIN

-- Dynamically count rows in the specified table

EXECUTE IMMEDIATE 'SELECT COUNT(\*) FROM ' || v\_table\_name INTO v\_count;

-- Print the total count

DBMS\_OUTPUT.PUT\_LINE('Total Employees: ' || v\_count);

END;

[**Powered By**](https://www.datacamp.com/datalab)

Ref Cursors are dynamic cursors that can be opened, fetched, and closed at runtime, enabling the passing of query results between program units.

The PL/SQL block below uses a REF CURSOR to fetch and print employee names from department 20. The cursor is opened dynamically, iterated through using a loop, and closed after processing.

DECLARE

-- Define a REF CURSOR type

TYPE emp\_ref\_cursor IS REF CURSOR;

v\_cursor emp\_ref\_cursor; -- Declare a cursor variable

v\_name employees.employee\_name%TYPE; -- Variable to store employee name

BEGIN

-- Open the cursor for employees in department 20

OPEN v\_cursor FOR SELECT employee\_name FROM employees WHERE department\_id = 20;

-- Fetch and print employee names in a loop

LOOP

FETCH v\_cursor INTO v\_name;

EXIT WHEN v\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_name);

END LOOP;

-- Close the cursor

CLOSE v\_cursor;

END;

[**Powered By**](https://www.datacamp.com/datalab)

**13. How do you handle mutating table errors?**

Mutating table errors occur when a trigger attempts to modify the table it is triggered on. A compound trigger allows breaking trigger execution into multiple phases such as BEFORE, AFTER, and FOR EACH ROW to prevent direct modification issues.

The compound trigger below logs salary changes in the salary\_audit table efficiently by collecting data before each row update and performing a bulk insert after the statement, reducing context switches and improving performance.

CREATE OR REPLACE TRIGGER trg\_salary\_audit

FOR UPDATE OF salary ON employees -- Trigger fires on salary updates

COMPOUND TRIGGER

-- Declare an associative array to store audit records

TYPE t\_salary\_audit IS TABLE OF salary\_audit%ROWTYPE INDEX BY PLS\_INTEGER;

v\_audit\_data t\_salary\_audit;

v\_idx PLS\_INTEGER := 0;

-- Before updating each row, store old and new salary details

BEFORE EACH ROW IS

BEGIN

v\_idx := v\_idx + 1;

v\_audit\_data(v\_idx).employee\_id := :OLD.employee\_id;

v\_audit\_data(v\_idx).old\_salary := :OLD.salary;

v\_audit\_data(v\_idx).new\_salary := :NEW.salary;

v\_audit\_data(v\_idx).change\_date := SYSDATE;

END BEFORE EACH ROW;

-- After the statement, insert all audit records in bulk

AFTER STATEMENT IS

BEGIN

FORALL i IN 1..v\_idx

INSERT INTO salary\_audit VALUES v\_audit\_data(i);

END AFTER STATEMENT;

END trg\_salary\_audit;

[**Powered By**](https://www.datacamp.com/datalab)

**14. What is overloading in PL/SQL?**

Overloading allows multiple procedures or functions with the same name but different parameters to be defined within a package. This enhances code readability and maintainability by providing multiple ways to perform similar operations.

In the query below, the package body implements two overloaded procedures named update\_salary: One increases an employee's salary by a specified amount, while the other sets a new salary with an effective date, updating the employees table accordingly.

-- Create the package specification

CREATE OR REPLACE PACKAGE emp\_pkg AS

-- Procedure to increment salary by a specified amount

PROCEDURE update\_salary(p\_emp\_id NUMBER, p\_increment NUMBER);

-- Overloaded procedure to set a new salary with an effective date

PROCEDURE update\_salary(p\_emp\_id NUMBER, p\_new\_salary NUMBER, p\_effective\_date DATE);

END emp\_pkg;

/

[**Powered By**](https://www.datacamp.com/datalab)

-- Create the package body

CREATE OR REPLACE PACKAGE BODY emp\_pkg AS

-- Procedure to increment salary by a specified amount

PROCEDURE update\_salary(p\_emp\_id NUMBER, p\_increment NUMBER) AS

BEGIN

UPDATE employees

SET salary = salary + p\_increment

WHERE employee\_id = p\_emp\_id;

END update\_salary;

-- Overloaded procedure to set a new salary with an effective date

PROCEDURE update\_salary(p\_emp\_id NUMBER, p\_new\_salary NUMBER, p\_effective\_date DATE) AS

BEGIN

UPDATE employees

SET salary = p\_new\_salary, last\_update = p\_effective\_date

WHERE employee\_id = p\_emp\_id;

END update\_salary;

END emp\_pkg;

/

[**Powered By**](https://www.datacamp.com/datalab)

**15. What are the compiler directives and pragmas in PL/SQL?**

PL/SQL provides compiler directives (PRAGMA) to optimize code and handle exceptions. The common pragmas include:

* PRAGMA EXCEPTION\_INIT: Associates a user-defined exception with an Oracle error code.
* PRAGMA SERIALLY\_REUSABLE: Optimizes package memory usage for scalability.

The PL/SQL block below handles the insertion of an employee with an invalid salary by using a custom exception e\_invalid\_salary mapped to error code -20001. If the exception is raised, it prints an error message.

DECLARE

e\_invalid\_salary EXCEPTION; -- Declare custom exception for invalid salary

PRAGMA EXCEPTION\_INIT(e\_invalid\_salary, -20001); -- Associate exception with error code -20001

BEGIN

-- Attempt to insert an employee with invalid salary

INSERT INTO employees (employee\_id, salary) VALUES (999, -1000);

EXCEPTION

-- Handle the custom exception and print a message

WHEN e\_invalid\_salary THEN

DBMS\_OUTPUT.PUT\_LINE('Invalid salary detected!');

END;

[**Powered By**](https://www.datacamp.com/datalab)

**16. What are the different methods to trace and debug PL/SQL code?**

PL/SQL provides several built-in packages for tracing and debugging code performance. Common methods include using DBMS\_TRACE to track execution flow, DBMS\_APPLICATION\_INFO to monitor session activity, and DBMS\_SESSION to gather session-level diagnostic information.

The query below first enables SQL tracing using DBMS\_SESSION.set\_sql\_trace(TRUE), then sets client-specific session information with DBMS\_APPLICATION\_INFO.set\_client\_info() for monitoring. The PL/SQL block is executed, which simulates an UPDATE operation. Finally, it disables the tracing after the execution using DBMS\_SESSION.set\_sql\_trace(FALSE).

-- Enable tracing for the current session using DBMS\_SESSION

BEGIN

-- Start session-level tracing

DBMS\_SESSION.set\_sql\_trace(TRUE);

END;

/

-- Set application information using DBMS\_APPLICATION\_INFO

BEGIN

-- Set the application name and action for session monitoring

DBMS\_APPLICATION\_INFO.set\_client\_info('Trace Debug Session');

DBMS\_APPLICATION\_INFO.set\_action('Debugging PL/SQL Code');

END;

/

-- Example PL/SQL block that simulates a process for debugging

DECLARE

v\_employee\_id NUMBER := 100;

BEGIN

-- Example query to fetch employee details

FOR rec IN (SELECT first\_name, last\_name FROM employees WHERE employee\_id = v\_employee\_id) LOOP

DBMS\_OUTPUT.put\_line('Employee: ' || rec.first\_name || ' ' || rec.last\_name);

END LOOP;

-- Simulate some logic that could be traced

IF v\_employee\_id = 100 THEN

DBMS\_OUTPUT.put\_line('Employee ID is 100');

END IF;

END;

/

-- Disable tracing after the session is complete

BEGIN

-- Stop session-level tracing

DBMS\_SESSION.set\_sql\_trace(FALSE);

END;

/

[**Powered By**](https://www.datacamp.com/datalab)

**Scenario-Based PL/SQL Interview Questions**

In real-world enterprise applications, PL/SQL is used to handle complex business logic. Interviewers often assess a candidate’s ability to apply PL/SQL in practical scenarios. Below are some key challenges, along with strategies to solve them.

For this next and final section, if you are interviewing, I would encourage you to think about how to answer questions following this template, but try, of course, to re-orient your answer for your specific industry or to reflect your own experience. The same goes if you are interviewing a candidate and want ideas on how to ask a question and judge how thorough the answer is, in which case I hope this section serves as inspiration for both your questions and evaluation criteria.

**Boost Your Team's SQL Proficiency**

Train your team in SQL with DataCamp for Business. Comprehensive training, hands-on projects, and detailed performance metrics for your organization.

[**Request a Demo Today!**](https://www.datacamp.com/business/demo)

**17. How do you design triggers in heavily concurrent environments?**

Assume a scenario where you need to enforce a business rule where an employee’s salary cannot be updated more than once per day. However, the database experiences high transaction concurrency, and a simple trigger could lead to contention or performance issues.

Instead of using a row-level trigger that fires for every update and can slow down performance, use a statement-level trigger with a log table to prevent multiple salary updates within the same day.

For example, the trigger below prevents multiple salary updates for an employee on the same day by checking the salary\_update\_log table before allowing an update. If the salary has already been updated today, an error is raised; otherwise, the update date is logged.

-- Create a table to log the last salary update for each employee

CREATE TABLE salary\_update\_log (

employee\_id NUMBER PRIMARY KEY, -- Employee ID as primary key

last\_update DATE -- Date of the last salary update

);

-- Create or replace a compound trigger to prevent multiple salary updates on the same day

CREATE OR REPLACE TRIGGER trg\_prevent\_multiple\_salary\_update

FOR UPDATE OF salary ON employees

COMPOUND TRIGGER

-- Declare a variable to store last salary update date per row

TYPE emp\_log\_type IS TABLE OF DATE INDEX BY PLS\_INTEGER;

emp\_log emp\_log\_type;

BEFORE STATEMENT IS

BEGIN

-- Load existing salary update logs into memory for reference

FOR rec IN (SELECT employee\_id, last\_update FROM salary\_update\_log) LOOP

emp\_log(rec.employee\_id) := rec.last\_update;

END LOOP;

END BEFORE STATEMENT;

BEFORE EACH ROW IS

BEGIN

-- Check if an entry exists for this employee

IF emp\_log.EXISTS(:NEW.employee\_id) THEN

-- Validate if salary was updated today

IF emp\_log(:NEW.employee\_id) = TRUNC(SYSDATE) THEN

RAISE\_APPLICATION\_ERROR(-20010, 'Salary can only be updated once per day');

ELSE

-- Update log in memory

emp\_log(:NEW.employee\_id) := TRUNC(SYSDATE);

END IF;

ELSE

-- Insert a new log entry into memory

emp\_log(:NEW.employee\_id) := TRUNC(SYSDATE);

END IF;

END BEFORE EACH ROW;

AFTER STATEMENT IS

BEGIN

-- Apply changes to the log table after all row updates

FOR i IN emp\_log.FIRST .. emp\_log.LAST LOOP

MERGE INTO salary\_update\_log l

USING (SELECT i AS employee\_id, emp\_log(i) AS last\_update FROM DUAL) s

ON (l.employee\_id = s.employee\_id)

WHEN MATCHED THEN

UPDATE SET l.last\_update = s.last\_update

WHEN NOT MATCHED THEN

INSERT (employee\_id, last\_update) VALUES (s.employee\_id, s.last\_update);

END LOOP;

END AFTER STATEMENT;

END trg\_prevent\_multiple\_salary\_update;

[**Powered By**](https://www.datacamp.com/datalab)

The above example would reduce row-level locking and minimize transaction conflicts during bulk updates.

**18. How do you split large transactions into smaller chunks for reliability?**

Imagine a scenario where a banking system requires a bulk update to adjust interest rates for millions of customer accounts. Executing a single large transaction might lock tables for too long or lead to rollback failures.

For this case, use bulk processing with COMMIT in batches to process the data incrementally and avoid contention.

The PL/SQL block below implements the solution by using BULK COLLECT to fetch account IDs and then iterates through them to update the interest rate by 5%. It commits the changes after every 1000 updates to improve performance and reduce resource usage. A final commit ensures any remaining updates are saved.

DECLARE

-- Define a collection type for account IDs

TYPE t\_accounts IS TABLE OF NUMBER; -- Use NUMBER instead of referencing accounts.account\_id

v\_account\_ids t\_accounts; -- Variable to store account IDs

v\_batch\_size CONSTANT NUMBER := 1000; -- Batch size for commits

BEGIN

-- Bulk collect all account IDs into the collection

SELECT account\_id BULK COLLECT INTO v\_account\_ids FROM accounts;

-- Loop through each account ID to update the interest rate

FOR i IN 1 .. v\_account\_ids.COUNT LOOP

UPDATE accounts

SET interest\_rate = interest\_rate \* 1.05 -- Increase interest rate by 5%

WHERE account\_id = v\_account\_ids(i);

-- Commit after every 1000 updates

IF MOD(i, v\_batch\_size) = 0 THEN

COMMIT;

END IF;

END LOOP;

COMMIT; -- Final commit for any remaining updates

END;

/

[**Powered By**](https://www.datacamp.com/datalab)

This solution would ensure reliability in financial systems like payroll by preventing transaction failures.

**19. How do you organize complex logic into packages for maintainability?**

A retail company needs to implement an order processing system that involves multiple steps: Checking inventory availability, reserving stock, calculating discounts, and logging transaction history.

Instead of writing separate standalone procedures, you need to organize this logic in a structured, maintainable way. Therefore, use PL/SQL packages to encapsulate related procedures and functions, improving code reusability and maintainability

The package specification below defines functions and procedures for order processing, including checking inventory, reserving stock, calculating discounts, and logging transaction statuses. It provides a modular approach to handling order-related tasks.

-- Create a package specification for order processing functions and procedures

CREATE PACKAGE order\_processing\_pkg AS

-- Function to check if enough inventory is available for the product

FUNCTION check\_inventory(p\_product\_id NUMBER, p\_quantity NUMBER) RETURN BOOLEAN;

-- Procedure to reserve stock for a specific order

PROCEDURE reserve\_stock(p\_order\_id NUMBER, p\_product\_id NUMBER, p\_quantity NUMBER);

-- Function to calculate discount based on customer and total amount

FUNCTION calculate\_discount(p\_customer\_id NUMBER, p\_total\_amount NUMBER) RETURN NUMBER;

-- Procedure to log the status of a transaction

PROCEDURE log\_transaction(p\_order\_id NUMBER, p\_status VARCHAR2);

END order\_processing\_pkg;

[**Powered By**](https://www.datacamp.com/datalab)

We then use the following package body to implement the functions and procedures defined in the package specification. It handles key tasks like checking inventory availability, reserving stock, calculating discounts, and logging transaction statuses, ensuring smooth order processing operations.

-- Create the package body implementing the functions and procedures for order processing

CREATE PACKAGE BODY order\_processing\_pkg AS

-- Function to check if sufficient inventory is available for the product

FUNCTION check\_inventory(p\_product\_id NUMBER, p\_quantity NUMBER) RETURN BOOLEAN AS

v\_available\_qty NUMBER; -- Variable to store available quantity

BEGIN

-- Retrieve the available quantity from inventory

SELECT stock\_quantity INTO v\_available\_qty FROM inventory WHERE product\_id = p\_product\_id;

-- Return true if enough stock is available, otherwise false

RETURN v\_available\_qty >= p\_quantity;

END check\_inventory;

-- Procedure to reserve stock for a specific order

PROCEDURE reserve\_stock(p\_order\_id NUMBER, p\_product\_id NUMBER, p\_quantity NUMBER) AS

BEGIN

-- Deduct the ordered quantity from the inventory

UPDATE inventory SET stock\_quantity = stock\_quantity - p\_quantity WHERE product\_id = p\_product\_id;

END reserve\_stock;

-- Function to calculate a discount based on the total order amount

FUNCTION calculate\_discount(p\_customer\_id NUMBER, p\_total\_amount NUMBER) RETURN NUMBER AS

v\_discount NUMBER := 0; -- Initialize discount to 0

BEGIN

-- Apply 10% discount if the total amount is greater than 500

IF p\_total\_amount > 500 THEN

v\_discount := p\_total\_amount \* 0.10;

END IF;

-- Return the calculated discount

RETURN v\_discount;

END calculate\_discount;

-- Procedure to log the transaction status

PROCEDURE log\_transaction(p\_order\_id NUMBER, p\_status VARCHAR2) AS

BEGIN

-- Insert a log entry for the order status

INSERT INTO order\_log (order\_id, status, log\_date) VALUES (p\_order\_id, p\_status, SYSDATE);

END log\_transaction;

END order\_processing\_pkg;

[**Powered By**](https://www.datacamp.com/datalab)

**20. How do you handle deadlocks in high-transaction systems?**

Assume a scenario where a financial system frequently updates multiple related tables simultaneously. Deadlocks occur when two transactions wait on each other’s locked resources, causing performance bottlenecks.

To solve this problem, always lock rows in a consistent order across transactions. Also, use the NOWAIT or SKIP LOCKED clause to prevent waiting indefinitely.

For example, the PL/SQL block below attempts to lock a specific row in the accounts table for the update using the FOR UPDATE NOWAIT clause, which causes the transaction to fail immediately if another session already locks the row. After locking, it updates the transaction status and commits the changes. If an error occurs, it catches the exception and prints an error message.

DECLARE

v\_balance NUMBER(15,2); -- Declare variable to store the account balance

BEGIN

-- Lock the account row for update to prevent other sessions from modifying it

SELECT balance INTO v\_balance FROM accounts

WHERE account\_id = 101 FOR UPDATE NOWAIT;

-- Update the transaction status to 'Processed' for all transactions related to this account

UPDATE transactions

SET status = 'Processed'

WHERE account\_id = 101 AND status = 'Pending'; -- Update only pending transactions

COMMIT; -- Commit the changes

EXCEPTION

-- Handle errors, such as locking issues or unexpected exceptions

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Transaction failed: ' || SQLERRM);

ROLLBACK; -- Ensure rollback in case of failure

END;

/

[**Powered By**](https://www.datacamp.com/datalab)

Similarly, this PL/SQL block processes pending transactions by locking them for update with FOR UPDATE SKIP LOCKED, allowing it to skip over rows that are already locked by other sessions. It updates each transaction's status to 'Processing' and commits the changes at the end.

DECLARE

-- Declare a cursor to select pending transactions and lock rows for update

CURSOR c\_pending\_txns IS

SELECT transaction\_id FROM transactions WHERE status = 'Pending' FOR UPDATE SKIP LOCKED; -- Skip locked rows

BEGIN

-- Loop through the pending transactions

FOR txn IN c\_pending\_txns LOOP

-- Update the status of each transaction to 'Processing'

UPDATE transactions SET status = 'Processing' WHERE transaction\_id = txn.transaction\_id;

END LOOP;

COMMIT; -- Commit the changes to finalize the transaction updates

END;

### Business Logic & Analytics Scenarios

Business logic and analytics scenario questions are critical in advanced SQL interviews because they assess your ability to translate real-world requirements into efficient queries:

**1.**[**How would you formulate a query to calculate the average annual retention for each subsequent year at the end of the year?**](https://www.interviewquery.com/questions/annual-retention)

To calculate the average annual retention, the solution involves creating several common table expressions (CTEs) to track the status of subscriptions over the years. The process includes determining the year of payment, checking the previous year’s status, and identifying active subscriptions at the end of each year. The final step involves calculating the retention rate by comparing active subscriptions at the end of the current year with those from the previous year, excluding new users.

**2.**[**Write a query to get the number of customers who were upsold**](https://www.interviewquery.com/questions/upsell-transactions)

To determine the number of upsold customers, identify users who made more than one purchase on different days. This involves grouping transactions by user and filtering for those with multiple purchase dates.

**Solution:**

SELECT user\_id

FROM transactions

GROUP BY user\_id

HAVING COUNT(DISTINCT DATE(created\_at)) > 1

**3.**[**Write a SQL query to find the average number of right swipes for different ranking algorithms**](https://www.interviewquery.com/questions/swipe-precision)

To solve this, join the **swipes** and **variants** tables on **user\_id**, filter users who have swiped at least 10 times, and group by **variant** and **swipe\_threshold**. Calculate the average of **is\_right\_swipe** for each group to get the mean number of right swipes, and count the number of users in each group.

**4.**[**Write a query to get the current salary for each employee after an ETL error.**](https://www.interviewquery.com/questions/employee-salaries-etl-error)

To solve this, you need to identify the most recent salary entry for each employee, which can be done by selecting the maximum **id** for each unique combination of **first\_name** and **last\_name**. This ensures that you retrieve the latest salary record for each employee.

**Solution:**

SELECT e.first\_name, e.last\_name, e.salary

FROM employees AS e

INNER JOIN (

SELECT first\_name, last\_name, MAX(id) AS max\_id

FROM employees

GROUP BY 1,2

) AS m

ON e.id = m.max\_id

**5.**[**Given the transactions table below, write a query that finds the third purchase of every user.**](https://www.interviewquery.com/questions/third-purchase)

To find the third purchase of every user, you can use a window function like ROW\_NUMBER() to assign a sequential number to each purchase per user, ordered by the **created\_at** and **id** fields. Then, filter the results to only include rows where this sequential number is 3, indicating the third purchase.

**Solution:**

SELECT user\_id, created\_at, product\_id

FROM (

SELECT \*,

RANK() OVER (PARTITION BY user\_id ORDER BY created\_at, id) AS rnk

FROM transactions

) t

WHERE rnk = 3

**6.**[**Write a query to find the top five paired products and their names**](https://www.interviewquery.com/questions/paired-products)

To find paired products often purchased together, join the **transactions** and **products** tables to associate transactions with product names. Use a self-join on the combined table to identify pairs of products bought together by the same user at the same time. Ensure the product names are ordered alphabetically to avoid duplicate pairs, and count the occurrences of each pair to find the top five most common pairs.

**7.**[**Write a query to forecast the budget for all projects and return a label of “overbudget” if it is over budget and “within budget” otherwise.**](https://www.interviewquery.com/questions/over-budget-projects)

To determine if a project is over budget, calculate the total prorated salary of employees working on the project and compare it to the project’s budget. If the prorated salary exceeds the budget, label the project as “overbudget”; otherwise, label it as “within budget”.

SELECT

title,

CASE WHEN (DATEDIFF(end\_date, start\_date) / 365.0) \* SUM(COALESCE(salary, 0)) > budget

THEN 'overbudget' ELSE 'within budget'

END AS project\_forecast

FROM projects p

LEFT JOIN employee\_projects ep ON p.id = ep.project\_id

LEFT JOIN employees e ON e.id = ep.employee\_id

GROUP BY title, end\_date, start\_date, budget

**8.**[**Given a table of students and their SAT test scores, write a query to return the two students with the closest test scores with the score difference.**](https://www.interviewquery.com/questions/closest-sat-scores)

To solve this, you need to calculate the absolute difference between each pair of scores and find the pair with the smallest difference. If there are multiple pairs with the same minimum difference, select the pair with student names that are higher alphabetically.

**Solution:**

SELECT

s1.student AS one\_student

, s2.student AS other\_student

, ABS(s1.score - s2.score) AS score\_diff

FROM scores AS s1

INNER JOIN scores AS s2

ON s1.id != s2.id

AND s1.id < s2.id

ORDER BY 3 ASC, 1 ASC

LIMIT 1

### Joins, Nulls, and Edge Case Handling

These types of questions, solvable with joins but requiring edge case handling, are essential in advanced SQL interviews because they test your ability to accurately track user behavior and ensure data integrity in attribution and analytics workflows:

**9.**[**Calculate the first touch attribution for each user\_id that converted.**](https://www.interviewquery.com/questions/first-touch-attribution)

To determine the first touch attribution, join the **attribution** and **user\_sessions** tables on **session\_id**. Filter for sessions where **conversion** is true, and then find the earliest session for each **user\_id** to identify the channel through which they first discovered the website.

**Solution:**

SELECT user\_id, MIN(created\_at) AS first\_touch

FROM user\_sessions

WHERE user\_id IN (

SELECT us.user\_id

FROM attribution a

JOIN user\_sessions us ON a.session\_id = us.session\_id

WHERE conversion = 1

)

GROUP BY user\_id

**10.**[**Write a query to create a new table that displays unique pairs of two locations**](https://www.interviewquery.com/questions/flight-records)

To create a table of unique flight routes, use a **WITH** clause to select the minimum and maximum of **source\_location** and **destination\_location** for each flight. Then, group by these calculated columns to ensure each pair is unique, and select them as **destination\_one** and **destination\_two**.

**11.**[**Write a query to count how many users posted the same job more than once and how many users only posted unique jobs.**](https://www.interviewquery.com/questions/repeat-job-postings)

To solve this, create a subquery to count distinct job postings and total postings per user. Use **COUNT DISTINCT** for unique job IDs and **COUNT** for total postings, then group by **user\_id**. In the main query, use **CASE WHEN** to differentiate users with multiple postings from those with unique postings, summing them into **multiple\_posts** and **single\_post** respectively.

**12.**[**Find how many users logged in a certain number of times on a given day**](https://www.interviewquery.com/questions/daily-logins)

To solve this, count the number of logins each user had on January 1st, 2022, and then group these counts to determine how many users logged in the same number of times. This involves a subquery to count logins per user and a main query to group by the number of logins.

**13.**[**Given two tables: accounts, and downloads, find the average number of downloads for free vs paying accounts, broken down by day.**](https://www.interviewquery.com/questions/download-facts)

To solve this, join the **accounts** and **downloads** tables on **account\_id**, filter for accounts with at least one download, and group by **download\_date** and **paying\_customer**. Calculate the average downloads for each group and round the result to two decimal places.

**Solution:**

SELECT download\_date, paying\_customer, ROUND(AVG(downloads),2) AS average\_downloads

FROM accounts a

JOIN downloads b ON a.account\_id = b.account\_id

GROUP BY download\_date, paying\_customer

**14.**[**Given a table of song\_plays and a table of users, write a query to extract the earliest date each user played their third unique song.**](https://www.interviewquery.com/questions/third-unique-song)

To solve this, you can use a combination of window functions and common table expressions (CTEs) to rank the unique songs played by each user. Then, filter to find the third unique song and join with the users table to get the required output, handling cases where users have less than three unique songs by returning NULL for date and song name.

**15.**[**Given a table with event logs, find the top five users with the longest continuous streak of visiting the platform.**](https://www.interviewquery.com/questions/longest-streak-users)

To solve this, you need to calculate the number of consecutive days each user visits the platform. This involves grouping the events by user and date, then calculating the streaks by checking for consecutive days. Finally, sort the users by their longest streak and select the top five.

**Solution:**

SELECT user\_id, COUNT(\*) AS streak\_length

FROM grouped

GROUP BY user\_id, grp

ORDER BY streak\_length DESC

LIMIT 5

**16.**[**Given a table of transactions and a table of users, write a query to determine if users tend to order more to their primary address versus other addresses.**](https://www.interviewquery.com/questions/order-addresses)

To solve this, join the transactions and users tables on the user\_id and id columns, respectively. Then, calculate the percentage of transactions where the shipping\_address matches the user’s address, and return this as home\_address\_percent.

**Solutions:**

SELECT

ROUND(

SUM(CASE WHEN u.address = t.shipping\_address THEN 1 END)

/ COUNT(t.id)

,2) as home\_address\_percent

FROM transactions as t

JOIN users as u

ON t.user\_id = u.id

### SQL Server/Oracle/PL/SQL Specific Scenarios

This type of question is vital for SQL Server, Oracle, or PL/SQL interviews because it evaluates your command of window functions, partitioning, and query optimization:

**17.**[**Write a query to get the total three-day rolling average for deposits by day**](https://www.interviewquery.com/questions/rolling-bank-transactions)

To calculate the three-day rolling average for deposits, filter the transactions to include only deposits (positive values) and group them by day. Use a window function to compute the rolling average over a three-day window, ensuring the date format in the output is ‘%Y-%m-%d’.

**Solution:**

SELECT vt2.dt, AVG(vt1.total\_deposits) AS rolling\_three\_day

FROM valid\_transactions AS vt1

INNER JOIN valid\_transactions AS vt2

ON vt1.dt > DATE\_ADD(vt2.dt, INTERVAL -3 DAY)

AND vt1.dt <= vt2.dt

GROUP BY vt2.dt

**18.**[**Get the top 3 highest employee salaries by department**](https://www.interviewquery.com/questions/top-three-salaries)

To achieve this, join the **employees** and **departments** tables on the department ID. Use a window function like **ROW\_NUMBER()** to rank employees by salary within each department, and filter to include only the top 3 salaries per department. Finally, select the full name, department name, and salary, sorting by department name and salary.

**Solutions:**

SELECT CONCAT(first\_name, ' ', last\_name) AS employee\_name, department\_name, salary

FROM (

SELECT department\_id, first\_name, last\_name, salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank

FROM employees

) AS ranked

JOIN departments d ON ranked.department\_id = d.id

WHERE rank <= 3

ORDER BY department\_name, salary DESC

**19.**[**Write a SQL query to select the 2nd highest salary in the engineering department.**](https://www.interviewquery.com/questions/2nd-highest-salary)

To find the second-highest salary in the engineering department, join the employees and departments tables on the department\_id and filter for the engineering department. Use a subquery to rank the salaries and select the second-highest value.

**Solutions:**

SELECT salary

FROM employees

INNER JOIN departments ON employees.department\_id = [departments.id](http://departments.id/)

WHERE [departments.name](http://departments.name/) = 'engineering'

ORDER BY salary DESC

LIMIT 1 OFFSET 1

**20.**[**Write a query to report the sum of regular salaries, overtime pay, and total compensations for each role.**](https://www.interviewquery.com/questions/hr-salary-reporting)

To achieve this, group the data by job title and calculate the sum of regular salaries and overtime payments. The total compensation is the sum of these two values. Use SQL aggregate functions like SUM() to compute these totals for each job title.

**Solution:**

SELECT

job\_title,

SUM(salary) AS total\_salaries,

SUM(overtime\_hours \* overtime\_rate) AS total\_overtime\_payments,

SUM(salary + (overtime\_hours \* overtime\_rate)) AS total\_compensation

FROM

employees

GROUP BY

job\_title;

**21.**[**Write a query to select the top 3 departments with at least ten employees and rank them according to the percentage of their employees making over 100K in salary.**](https://www.interviewquery.com/questions/employee-salaries)

First, filter departments with at least ten employees, then calculate the percentage of employees earning over 100K. Use a subquery to count employees per department and another to count those earning over 100K, then rank the departments based on this percentage.

**Solution:**

SELECT AVG(CASE WHEN salary > 100000

THEN 1 ELSE 0 END) AS percentage\_over\_100k

, d.name as department\_name

, COUNT(\*) AS number\_of\_employees

FROM departments AS d

LEFT JOIN employees AS e

ON d.id = e.department\_id

GROUP BY d.name

HAVING COUNT(\*) >= 10

ORDER BY 1 DESC

LIMIT 3

**22.**[**Calculate the percentage of total revenue to date that was made during the first and last years recorded in the table.**](https://www.interviewquery.com/questions/percentage-of-revenue-by-year)

Identify the first and last years from the **created\_at** column, then calculate the total revenue for each of these years. Divide these by the total revenue across all years to get the percentage, and round the result to two decimal places for precision.

**Solution:**

SELECT

ROUND((SUM(CASE WHEN YEAR(created\_at) = (SELECT MIN(YEAR(created\_at)) FROM annual\_payments) THEN amount - amount\_refunded END)) \* 100 / SUM(amount - amount\_refunded), 2) AS percent\_first,

ROUND((SUM(CASE WHEN YEAR(created\_at) = (SELECT MAX(YEAR(created\_at)) FROM annual\_payments) THEN amount - amount\_refunded END)) \* 100 / SUM(amount - amount\_refunded), 2) AS percent\_last

FROM annual\_payments

### Query Optimization Scenarios

This scenario is critical in advanced SQL interviews because it tests your ability to optimize self-joins, handle date range logic, and write performant queries that scale:

**23.**[**Given a table of product subscriptions, write a query to determine if each user has overlapping subscription date ranges.**](https://www.interviewquery.com/questions/subscription-overlap)

To solve this, you need to compare each user’s subscription date range with others to check for overlaps. This can be done by joining the table with itself and checking if the start date of one subscription is before the end date of another and vice versa, ensuring the user IDs are different.

**Solution:**

SELECT

s1.user\_id

, MAX(CASE WHEN s2.user\_id IS NOT NULL THEN 1 ELSE 0 END) AS overlap

FROM subscriptions AS s1

LEFT JOIN subscriptions AS s2

ON s1.user\_id != s2.user\_id

AND s1.start\_date <= s2.end\_date

AND s1.end\_date >= s2.start\_date

GROUP BY 1

**24. Optimize a query using LIKE '%search%' on product descriptions**

Implement full-text indexing to replace inefficient wildcard searches, reducing full table scans. Use **CONTAINS()** in SQL Server or **tsvector** in PostgreSQL for faster pattern matching. Avoid leading wildcards when possible by leveraging reverse-string indexes.

**Solution:**

-- Step 1: Enable full-text indexing on the table

CREATE FULLTEXT CATALOG ProductCatalog AS DEFAULT;

-- Step 2: Create a full-text index on the ProductDescription column

CREATE FULLTEXT INDEX ON Products(ProductDescription)

KEY INDEX PK\_Products -- replace with your primary key name

WITH STOPLIST = SYSTEM;

-- Step 3: Search using CONTAINS instead of LIKE '%search%'

SELECT ProductID, ProductName, ProductDescription

FROM Products

WHERE CONTAINS(ProductDescription, '"wireless"');

**25.**[**Identify and label each event with its corresponding session number**](https://www.interviewquery.com/questions/identifying-user-sessions)

To label each event with a session number, sort the **events** table by **user\_id** and **created\_at**. Use a window function to calculate the time difference between consecutive events for each user, and increment the session number when the difference exceeds 60 minutes. This will group events into sessions based on the defined criteria.

**Solution:**

CASE

WHEN TIMESTAMPDIFF(MINUTE, LEAD(created\_at) OVER(PARTITION BY user\_id ORDER BY created\_at DESC), created\_at) > 60

OR TIMESTAMPDIFF(MINUTE, LEAD(created\_at) OVER(PARTITION BY user\_id ORDER BY created\_at DESC), created\_at) IS NULL

THEN 1

ELSE 0

END AS is\_new\_sesh

**26. Explain covering indexes for a query joining orders and products on product\_id**

Create a composite index on orders.product\_id including order\_date and quantity, and another on products.product\_id including price. This enables index-only scans and reduces I/O during hash joins

**Solution:**

CREATE NONCLUSTERED INDEX idx\_orders\_product\_covering

ON orders(product\_id)

INCLUDE (order\_date, quantity);

CREATE NONCLUSTERED INDEX