**Performance Tuning & Query Optimization**

**Q1:** How do you optimize a slow-performing SQL query?  
✅ Discuss indexing, partitioning, avoiding full table scans, proper joins, and execution plan analysis using EXPLAIN PLAN.

**Q2:** Given a large table with billions of rows, how would you speed up an aggregate query (like SUM, AVG)?  
✅ Answer: Use **indexes**, **materialized views**, or **query rewriting techniques (e.g., GROUP BY on indexed columns)**.

🔹 **Example Query: Optimize Aggregate Query**

sql

CopyEdit

SELECT customer\_id, SUM(amount)

FROM transactions

WHERE transaction\_date > SYSDATE - 30

GROUP BY customer\_id;

✅ Optimization: **Create an index on transaction\_date**, and consider **partitioning** if the table is huge.

**2️⃣ Complex Joins & Window Functions**

**Q3:** What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN?  
✅ Explain with examples and performance considerations.

🔹 **Example Query: Find Top 3 Highest Paid Employees in Each Department**

sql

CopyEdit

SELECT employee\_id, department\_id, salary,

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

FROM employees

WHERE rank <= 3;

✅ Uses RANK() for ranking employees **within each department**.

**3️⃣ Advanced Query Scenarios**

**Q4:** How would you **find duplicate records** in a table?  
🔹 **Example Query: Detect Duplicates in Orders Table**

sql

CopyEdit

SELECT order\_id, COUNT(\*)

FROM orders

GROUP BY order\_id

HAVING COUNT(\*) > 1;

**Q5:** How do you remove duplicates while keeping the latest record based on created\_at?

sql

CopyEdit

DELETE FROM orders o1

WHERE EXISTS (

SELECT 1 FROM orders o2

WHERE o1.order\_id = o2.order\_id

AND o1.created\_at < o2.created\_at

);

✅ Uses **EXISTS** to remove older duplicate entries.

**4️⃣ PL/SQL: Stored Procedures & Exception Handling**

**Q6:** How do you handle exceptions in PL/SQL?

🔹 **Example: Stored Procedure with Exception Handling**

sql

CopyEdit

CREATE OR REPLACE PROCEDURE update\_salary(emp\_id NUMBER, new\_salary NUMBER) AS

BEGIN

UPDATE employees SET salary = new\_salary WHERE employee\_id = emp\_id;

COMMIT;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee Not Found');

WHEN OTHERS THEN

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

ROLLBACK;

END;

✅ Explains error handling using **EXCEPTION** and **ROLLBACK**.

**5️⃣ Indexing & Partitioning**

**Q7:** What are different types of indexes, and when would you use them?  
✅ Discuss **B-tree Index, Bitmap Index, Unique Index, Function-based Index**.

🔹 **Example: Creating an Index on a Function-Based Query**

sql

CopyEdit

CREATE INDEX idx\_lower\_email ON users (LOWER(email));

✅ Optimizes case-insensitive searches.

**6️⃣ Advanced Query with Recursive CTEs**

**Q8:** How do you use **recursive CTEs** in SQL?

🔹 **Example: Hierarchical Query for Employee Reporting Structure**

sql

CopyEdit

WITH employee\_hierarchy AS (

SELECT employee\_id, manager\_id, 1 AS level

FROM employees

WHERE manager\_id IS NULL

UNION ALL

SELECT e.employee\_id, e.manager\_id, eh.level + 1

FROM employees e

INNER JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id

)

SELECT \* FROM employee\_hierarchy;

✅ Uses **recursive CTE** to traverse an **employee-manager hierarchy**.

**7️⃣ Handling Large Data Volumes (Partitioning & Bulk Processing)**

**Q9:** How would you efficiently delete **millions of rows** from a table?  
✅ Answer: Use **Partitioning, Truncate with Cascade, Bulk Delete in Batches**.

🔹 **Example: Delete Old Data in Batches**

sql

CopyEdit

DECLARE

v\_count NUMBER;

BEGIN

LOOP

DELETE FROM transactions WHERE transaction\_date < ADD\_MONTHS(SYSDATE, -12)

AND ROWNUM <= 10000;

v\_count := SQL%ROWCOUNT;

EXIT WHEN v\_count = 0;

COMMIT;

END LOOP;

END;

✅ Deletes data in **batches** to avoid table locks.

**🔹 Bonus Challenge Questions**

1. How do you handle **deadlocks** in SQL, and how can you prevent them?
2. What is the difference between **UNION** and **UNION ALL**, and when would you use each?
3. How do you identify and fix **missing indexes** on a slow-performing query?
4. How would you track **user login history** and show the last login date using SQL?
5. How do you perform **ETL operations using SQL (Extract, Transform, Load)?**

## **Performance Tuning & Query Optimization in SQL**

Performance tuning is essential to **speed up slow queries**, **reduce database load**, and **improve efficiency**. It involves **query optimization techniques**, **indexing strategies**, and **execution plan analysis**.

## **🔹 1️⃣ Key Techniques for Performance Tuning**

### ****1. Use Indexes Effectively****

✅ **Indexes speed up data retrieval** by allowing the database to find rows faster instead of scanning the whole table.

🔹 **Example: Creating an Index on Employee Name**

sql

CopyEdit

CREATE INDEX idx\_employee\_name ON employees(name);

🔹 **Use Case:** If you frequently search for employees by name:

sql

CopyEdit

SELECT \* FROM employees WHERE name = 'John Doe';

✅ This **reduces full table scans** and speeds up searches.

🔴 **When NOT to Use Indexes:**

* **On columns with high duplicate values (e.g., Gender, Status flags).**
* **When the table is small (~few hundred rows).**
* **For columns frequently updated, as indexes slow down INSERT/UPDATE/DELETE.**

### ****2. Avoid SELECT \*****

✅ Instead of selecting all columns, **only fetch required columns** to improve performance.

🔴 **Bad Query:**

sql

CopyEdit

SELECT \* FROM orders WHERE order\_date > '2024-01-01';

✅ **Optimized Query:**

sql

CopyEdit

SELECT order\_id, customer\_id, order\_total FROM orders WHERE order\_date > '2024-01-01';

**🔹 Why?**

* **Reduces memory usage** (fetching fewer columns).
* **Improves execution speed** (less data to transfer).

### ****3. Use EXPLAIN PLAN to Analyze Query Execution****

✅ **EXPLAIN PLAN** helps find performance issues **before execution**.

🔹 **Example:**

sql

CopyEdit

EXPLAIN ANALYZE SELECT \* FROM employees WHERE department\_id = 10;

✅ **What to Look for in Execution Plan?**

* **Index Usage** – If INDEX SCAN is present, the index is working.
* **Sequential Scan (FULL TABLE SCAN)** – Indicates missing indexes or inefficient queries.
* **Nested Loops** – Can be slow for large datasets; consider using joins properly.

### ****4. Optimize Joins (Avoid Cartesian Joins)****

✅ Joins should use **indexed columns** and avoid unnecessary cross joins.

🔴 **Bad Query (Slow due to Cartesian Product)**

sql

CopyEdit

SELECT \* FROM employees, departments WHERE employees.department\_id = departments.department\_id;

✅ **Optimized Query (Using INNER JOIN with Indexing)**

sql

CopyEdit

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

**🔹 Why?**

* **Avoids duplicate rows** caused by missing join conditions.
* **Uses indexes on department\_id** for fast lookup.

### ****5. Use Proper WHERE Conditions (Avoid Functions on Indexed Columns)****

✅ **Indexed columns should not be inside a function**, as it forces a full table scan.

🔴 **Bad Query:**

sql

CopyEdit

SELECT \* FROM users WHERE LOWER(email) = 'test@example.com';

✅ **Optimized Query (Using Function-Based Index)**

sql

CopyEdit

CREATE INDEX idx\_email\_lower ON users(LOWER(email));

SELECT \* FROM users WHERE email = 'test@example.com';

✅ Now, **the index is used**, speeding up the search.

### ****6. Use EXISTS Instead of IN for Large Subqueries****

✅ **EXISTS is faster than IN when dealing with large subqueries** because it stops searching once a match is found.

🔴 **Bad Query (Slow for Large Subqueries)**

sql

CopyEdit

SELECT \* FROM customers WHERE customer\_id IN (SELECT customer\_id FROM orders);

✅ **Optimized Query (Using EXISTS)**

sql

CopyEdit

SELECT \* FROM customers c WHERE EXISTS (

SELECT 1 FROM orders o WHERE o.customer\_id = c.customer\_id

);

✅ **Why?**

* EXISTS **stops searching after the first match**, improving performance.
* IN scans **all possible matches**, making it slower.

### ****7. Use UNION ALL Instead of UNION (If Duplicates Are Not a Concern)****

✅ **UNION removes duplicates, whereas UNION ALL keeps all records**, making it faster.

🔴 **Bad Query (Slow because of Duplicate Removal)**

sql

CopyEdit

SELECT name FROM employees WHERE department = 'HR'

UNION

SELECT name FROM managers;

✅ **Optimized Query (Faster Using UNION ALL)**

sql

CopyEdit

SELECT name FROM employees WHERE department = 'HR'

UNION ALL

SELECT name FROM managers;

✅ **Why?**

* UNION **performs a costly DISTINCT operation**.
* UNION ALL **simply appends results** without checking duplicates.

### ****8. Optimize Pagination Queries (Using LIMIT/OFFSET vs. Keyset Pagination)****

✅ **For large datasets, OFFSET can be slow; Keyset pagination is better.**

🔴 **Bad Query (Slow Pagination Using OFFSET)**

sql

CopyEdit

SELECT \* FROM orders ORDER BY order\_date DESC LIMIT 10 OFFSET 100000;

✅ **Optimized Query (Using Keyset Pagination - WHERE Clause on Indexed Column)**

sql

CopyEdit

SELECT \* FROM orders

WHERE order\_date < (SELECT order\_date FROM orders ORDER BY order\_date DESC LIMIT 1 OFFSET 100000)

ORDER BY order\_date DESC

LIMIT 10;

✅ **Why?**

* OFFSET **still loads all previous rows but discards them**.
* Keyset Pagination only **retrieves the required rows**.

## **🔹 2️⃣ Indexing Best Practices**

✅ **Use Indexes on Frequently Queried Columns**

* Good candidates: WHERE, JOIN, ORDER BY columns.

✅ **Use Composite Indexes When Querying Multiple Columns Together**

sql

CopyEdit

CREATE INDEX idx\_orders\_date\_status ON orders(order\_date, status);

✅ Speeds up:

sql

CopyEdit

SELECT \* FROM orders WHERE order\_date > '2024-01-01' AND status = 'Shipped';

✅ **Avoid Indexes on Highly Updated Columns**

* Indexes **slow down inserts/updates** because they must be maintained.
* Don’t index **frequently changing fields like "last\_login\_time"**.

## **🔹 3️⃣ Advanced Performance Tuning Tips**

1️⃣ **Partition Large Tables**

* **Partition by Date (Range Partitioning)**

sql

CopyEdit

CREATE TABLE transactions (

id NUMBER,

transaction\_date DATE

) PARTITION BY RANGE (transaction\_date) (

PARTITION p\_2023 VALUES LESS THAN (TO\_DATE('2024-01-01', 'YYYY-MM-DD')),

PARTITION p\_2024 VALUES LESS THAN (TO\_DATE('2025-01-01', 'YYYY-MM-DD'))

);

✅ **Speeds up queries** when filtering by transaction\_date.

2️⃣ **Materialized Views for Precomputed Aggregates**

* Instead of running heavy queries frequently, use **materialized views**.

sql

CopyEdit

CREATE MATERIALIZED VIEW monthly\_sales AS

SELECT customer\_id, SUM(order\_total) AS total\_sales

FROM orders

GROUP BY customer\_id;

✅ **Refresh periodically** to keep it updated:

sql

CopyEdit

REFRESH MATERIALIZED VIEW monthly\_sales;

3️⃣ **Optimize Temp Table Usage**

* If using temp tables, **avoid frequent creation and dropping**.
* Use **CTEs (WITH queries)** as alternatives.

## **📌 Final Takeaways**

✅ **Always check EXPLAIN PLAN before executing queries.**  
✅ **Use indexing wisely** – Avoid over-indexing but optimize for frequent queries.  
✅ **Optimize Joins & WHERE conditions** – Ensure queries use indexes efficiently.  
✅ **Batch processing for large data deletions** – Avoid table locks.  
✅ **Use UNION ALL instead of UNION when duplicates aren’t an issue.**

## **Performance Tuning & Query Optimization in SQL**

Performance tuning is essential to **speed up slow queries**, **reduce database load**, and **improve efficiency**. It involves **query optimization techniques**, **indexing strategies**, and **execution plan analysis**.

## **🔹 1️⃣ Key Techniques for Performance Tuning**

### ****1. Use Indexes Effectively****

✅ **Indexes speed up data retrieval** by allowing the database to find rows faster instead of scanning the whole table.

🔹 **Example: Creating an Index on Employee Name**

sql

CopyEdit

CREATE INDEX idx\_employee\_name ON employees(name);

🔹 **Use Case:** If you frequently search for employees by name:

sql

CopyEdit

SELECT \* FROM employees WHERE name = 'John Doe';

✅ This **reduces full table scans** and speeds up searches.

🔴 **When NOT to Use Indexes:**

* **On columns with high duplicate values (e.g., Gender, Status flags).**
* **When the table is small (~few hundred rows).**
* **For columns frequently updated, as indexes slow down INSERT/UPDATE/DELETE.**

### ****2. Avoid SELECT \*****

✅ Instead of selecting all columns, **only fetch required columns** to improve performance.

🔴 **Bad Query:**

sql

CopyEdit

SELECT \* FROM orders WHERE order\_date > '2024-01-01';

✅ **Optimized Query:**

sql

CopyEdit

SELECT order\_id, customer\_id, order\_total FROM orders WHERE order\_date > '2024-01-01';

**🔹 Why?**

* **Reduces memory usage** (fetching fewer columns).
* **Improves execution speed** (less data to transfer).

### ****3. Use EXPLAIN PLAN to Analyze Query Execution****

✅ **EXPLAIN PLAN** helps find performance issues **before execution**.

🔹 **Example:**

sql

CopyEdit

EXPLAIN ANALYZE SELECT \* FROM employees WHERE department\_id = 10;

✅ **What to Look for in Execution Plan?**

* **Index Usage** – If INDEX SCAN is present, the index is working.
* **Sequential Scan (FULL TABLE SCAN)** – Indicates missing indexes or inefficient queries.
* **Nested Loops** – Can be slow for large datasets; consider using joins properly.

### ****4. Optimize Joins (Avoid Cartesian Joins)****

✅ Joins should use **indexed columns** and avoid unnecessary cross joins.

🔴 **Bad Query (Slow due to Cartesian Product)**

sql

CopyEdit

SELECT \* FROM employees, departments WHERE employees.department\_id = departments.department\_id;

✅ **Optimized Query (Using INNER JOIN with Indexing)**

sql

CopyEdit

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

**🔹 Why?**

* **Avoids duplicate rows** caused by missing join conditions.
* **Uses indexes on department\_id** for fast lookup.

### ****5. Use Proper WHERE Conditions (Avoid Functions on Indexed Columns)****

✅ **Indexed columns should not be inside a function**, as it forces a full table scan.

🔴 **Bad Query:**

sql

CopyEdit

SELECT \* FROM users WHERE LOWER(email) = 'test@example.com';

✅ **Optimized Query (Using Function-Based Index)**

sql

CopyEdit

CREATE INDEX idx\_email\_lower ON users(LOWER(email));

SELECT \* FROM users WHERE email = 'test@example.com';

✅ Now, **the index is used**, speeding up the search.

### ****6. Use EXISTS Instead of IN for Large Subqueries****

✅ **EXISTS is faster than IN when dealing with large subqueries** because it stops searching once a match is found.

🔴 **Bad Query (Slow for Large Subqueries)**

sql

CopyEdit

SELECT \* FROM customers WHERE customer\_id IN (SELECT customer\_id FROM orders);

✅ **Optimized Query (Using EXISTS)**

sql

CopyEdit

SELECT \* FROM customers c WHERE EXISTS (

SELECT 1 FROM orders o WHERE o.customer\_id = c.customer\_id

);

✅ **Why?**

* EXISTS **stops searching after the first match**, improving performance.
* IN scans **all possible matches**, making it slower.

### ****7. Use UNION ALL Instead of UNION (If Duplicates Are Not a Concern)****

✅ **UNION removes duplicates, whereas UNION ALL keeps all records**, making it faster.

🔴 **Bad Query (Slow because of Duplicate Removal)**

sql

CopyEdit

SELECT name FROM employees WHERE department = 'HR'

UNION

SELECT name FROM managers;

✅ **Optimized Query (Faster Using UNION ALL)**

sql

CopyEdit

SELECT name FROM employees WHERE department = 'HR'

UNION ALL

SELECT name FROM managers;

✅ **Why?**

* UNION **performs a costly DISTINCT operation**.
* UNION ALL **simply appends results** without checking duplicates.

### ****8. Optimize Pagination Queries (Using LIMIT/OFFSET vs. Keyset Pagination)****

✅ **For large datasets, OFFSET can be slow; Keyset pagination is better.**

🔴 **Bad Query (Slow Pagination Using OFFSET)**

sql

CopyEdit

SELECT \* FROM orders ORDER BY order\_date DESC LIMIT 10 OFFSET 100000;

✅ **Optimized Query (Using Keyset Pagination - WHERE Clause on Indexed Column)**

sql

CopyEdit

SELECT \* FROM orders

WHERE order\_date < (SELECT order\_date FROM orders ORDER BY order\_date DESC LIMIT 1 OFFSET 100000)

ORDER BY order\_date DESC

LIMIT 10;

✅ **Why?**

* OFFSET **still loads all previous rows but discards them**.
* Keyset Pagination only **retrieves the required rows**.

## **🔹 2️⃣ Indexing Best Practices**

✅ **Use Indexes on Frequently Queried Columns**

* Good candidates: WHERE, JOIN, ORDER BY columns.

✅ **Use Composite Indexes When Querying Multiple Columns Together**

sql

CopyEdit

CREATE INDEX idx\_orders\_date\_status ON orders(order\_date, status);

✅ Speeds up:

sql

CopyEdit

SELECT \* FROM orders WHERE order\_date > '2024-01-01' AND status = 'Shipped';

✅ **Avoid Indexes on Highly Updated Columns**

* Indexes **slow down inserts/updates** because they must be maintained.
* Don’t index **frequently changing fields like "last\_login\_time"**.

## **🔹 3️⃣ Advanced Performance Tuning Tips**

1️⃣ **Partition Large Tables**

* **Partition by Date (Range Partitioning)**

sql

CopyEdit

CREATE TABLE transactions (

id NUMBER,

transaction\_date DATE

) PARTITION BY RANGE (transaction\_date) (

PARTITION p\_2023 VALUES LESS THAN (TO\_DATE('2024-01-01', 'YYYY-MM-DD')),

PARTITION p\_2024 VALUES LESS THAN (TO\_DATE('2025-01-01', 'YYYY-MM-DD'))

);

✅ **Speeds up queries** when filtering by transaction\_date.

2️⃣ **Materialized Views for Precomputed Aggregates**

* Instead of running heavy queries frequently, use **materialized views**.

sql

CopyEdit

CREATE MATERIALIZED VIEW monthly\_sales AS

SELECT customer\_id, SUM(order\_total) AS total\_sales

FROM orders

GROUP BY customer\_id;

✅ **Refresh periodically** to keep it updated:

sql

CopyEdit

REFRESH MATERIALIZED VIEW monthly\_sales;

3️⃣ **Optimize Temp Table Usage**

* If using temp tables, **avoid frequent creation and dropping**.
* Use **CTEs (WITH queries)** as alternatives.

## **📌 Final Takeaways**

✅ **Always check EXPLAIN PLAN before executing queries.**  
✅ **Use indexing wisely** – Avoid over-indexing but optimize for frequent queries.  
✅ **Optimize Joins & WHERE conditions** – Ensure queries use indexes efficiently.  
✅ **Batch processing for large data deletions** – Avoid table locks.  
✅ **Use UNION ALL instead of UNION when duplicates aren’t an issue.**

## **QL Hints, Parallel Processing, and Index Types Explained**

SQL **hints** are special directives that guide the database optimizer on how to execute queries. They **override the default execution plan** and help improve performance when used correctly.

## **🔹 1️⃣ What Are Hints in SQL?**

🔹 **SQL hints** are instructions placed inside queries to influence execution plans.  
🔹 They are mainly used to **force indexes, parallel execution, or join methods**.

### ****🔹 Common SQL Hints****

| **Hint Type** | **Description** |
| --- | --- |
| INDEX | Forces the query to use a specific index. |
| FULL | Forces a full table scan (useful if full scans are faster than indexed scans). |
| PARALLEL | Runs queries using multiple CPU threads. |
| ORDERED | Forces the join order in multi-table joins. |
| USE\_HASH | Forces a **hash join** instead of nested loop join. |

## **🔹 2️⃣** PARALLEL **Hint (For Faster Query Execution)**

✅ **Parallel execution allows queries to run across multiple CPU threads** to improve performance on large datasets.

🔹 **Example: Enable Parallel Execution on a Query**

sql

CopyEdit

SELECT /\*+ PARALLEL(orders, 4) \*/ \*

FROM orders

WHERE order\_date > SYSDATE - 30;

✅ **What This Does?**

* The orders table query will run using **4 parallel threads** instead of a single-threaded execution.
* **Best for large table scans, aggregations, and reports.**

🔹 **Example: Creating a Table with Parallel Processing**

sql

CopyEdit

CREATE TABLE sales\_data PARALLEL 8 AS

SELECT \* FROM sales;

✅ Uses **8 parallel threads** to create the table.

🔹 **When to Use PARALLEL?**  
✔️ When querying **large tables** (millions of rows).  
✔️ When performing **aggregations (SUM, COUNT, AVG)** on large datasets.  
✔️ When running **batch data processing or reports**.

🔴 **When NOT to Use PARALLEL?**  
❌ If the table is **small** (parallelism overhead is higher than benefits).  
❌ If the system has **limited CPU resources** (it may slow down other operations).

## **🔹 3️⃣** INDEX **Hint (Forcing Index Usage)**

✅ **If the optimizer isn’t choosing an index, you can force it with the INDEX hint.**

🔹 **Example: Forcing an Index on the customers Table**

sql

CopyEdit

SELECT /\*+ INDEX(customers idx\_customer\_email) \*/ \*

FROM customers

WHERE email = 'test@example.com';

✅ **What This Does?**

* Ensures the query **uses the idx\_customer\_email index** instead of a full table scan.

🔹 **Example: Forcing a Full Table Scan Instead of Index**

sql

CopyEdit

SELECT /\*+ FULL(customers) \*/ \*

FROM customers;

✅ **Forces a full table scan**, which may be faster for **small tables**.

## **🔹 4️⃣ Index Types (Clustered, Non-Clustered, Bitmap, Unique, Function-Based)**

### ****🔹 Clustered Index vs. Non-Clustered Index****

| **Index Type** | **Description** |
| --- | --- |
| **Clustered Index** | Physically arranges table rows in **sorted order** based on the index key. |
| **Non-Clustered Index** | Stores a separate structure pointing to the actual table rows (does not reorder the data). |

### ****🔹 Clustered Index (Faster for Sorting & Range Queries)****

✅ **Each table can have only ONE clustered index** (because it dictates physical row storage).  
✅ **Best for queries that filter on a range of values (BETWEEN, ORDER BY).**

🔹 **Example: Creating a Clustered Index in SQL Server**

sql

CopyEdit

CREATE CLUSTERED INDEX idx\_employee\_id ON employees(employee\_id);

✅ Now, rows are **physically sorted by employee\_id**.

🔹 **Example: Using Clustered Index for Fast Range Query**

sql

CopyEdit

SELECT \* FROM employees WHERE employee\_id BETWEEN 100 AND 200;

✅ **Fast because data is stored in sorted order**.

### ****🔹 Non-Clustered Index (Faster for Exact Lookups)****

✅ **Stores index separately from table data.**  
✅ **Best for queries that search for a specific value (e.g., WHERE email = 'test@example.com').**

🔹 **Example: Creating a Non-Clustered Index**

sql

CopyEdit

CREATE INDEX idx\_email ON users(email);

✅ **Lookups are fast**, but it adds an extra step to retrieve data.

### ****🔹 Unique Index (Enforces Uniqueness)****

✅ **Ensures that values in a column are unique.**

🔹 **Example: Creating a Unique Index on Email**

sql

CopyEdit

CREATE UNIQUE INDEX idx\_unique\_email ON users(email);

✅ **Prevents duplicate emails** from being inserted.

### ****🔹 Function-Based Index (For Optimizing Function-Based Queries)****

✅ **When a column is used inside a function in WHERE, normal indexes won’t work!**

🔴 **Bad Query (Index Not Used):**

sql

CopyEdit

SELECT \* FROM users WHERE LOWER(email) = 'test@example.com';

✅ **Solution: Function-Based Index**

sql

CopyEdit

CREATE INDEX idx\_lower\_email ON users(LOWER(email));

✅ **Now, the index is used, making searches faster!**

### ****🔹 Bitmap Index (For Low-Cardinality Columns)****

✅ **Used for columns with very few unique values (e.g., Gender, Status Flags).**

🔹 **Example: Creating a Bitmap Index on Status**

sql

CopyEdit

CREATE BITMAP INDEX idx\_status ON users(status);

✅ **Ideal for status = 'Active' OR status = 'Inactive' queries.**

🔴 **When NOT to Use Bitmap Indexes?**  
❌ When the column has **high uniqueness** (e.g., email, customer\_id).  
❌ When the table **has frequent updates** (Bitmap indexes slow down updates).

## **🔹 5️⃣ Other Important Hints for Performance Tuning**

| **Hint** | **Usage** | **Example** |
| --- | --- | --- |
| **FIRST\_ROWS(n)** | Optimizes query to return the first n rows faster. | SELECT /\*+ FIRST\_ROWS(10) \*/ \* FROM orders; |
| **LEADING(table\_name)** | Forces a specific table to be processed first in a join. | SELECT /\*+ LEADING(customers) \*/ \* FROM customers JOIN orders ON customers.id = orders.customer\_id; |
| **USE\_NL(table1 table2)** | Forces a **nested loop join** instead of hash join. | SELECT /\*+ USE\_NL(employees departments) \*/ \* FROM employees JOIN departments ON employees.department\_id = departments.department\_id; |
| **USE\_HASH(table1 table2)** | Forces a **hash join** for large data sets. | SELECT /\*+ USE\_HASH(products sales) \*/ \* FROM products JOIN sales ON products.product\_id = sales.product\_id; |

## **🔹 Final Takeaways**

✅ **Use PARALLEL for big queries** when CPU resources are available.  
✅ **Force INDEX hints when the optimizer is not using indexes efficiently.**  
✅ **Choose the right index type** (Clustered, Non-Clustered, Bitmap, Function-Based).  
✅ **Use EXPLAIN PLAN to validate if hints are improving query performance.**  
✅ **Avoid forcing hints unless necessary**, as the optimizer usually makes good choices.

### ****Q1: What is the difference between PL/SQL and SQL?****

✅ **PL/SQL** (Procedural Language SQL) supports **procedural programming**, loops, and exception handling, while **SQL** is a **declarative query language** for data retrieval and manipulation.

### ****Q2: Explain the different types of exceptions in PL/SQL.****

✅ **Predefined Exceptions:** Automatically raised by Oracle (e.g., NO\_DATA\_FOUND, TOO\_MANY\_ROWS).  
✅ **User-Defined Exceptions:** Defined explicitly using EXCEPTION and RAISE.

🔹 **Example:** Handling exceptions

sql

CopyEdit

DECLARE

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE employee\_id = 9999;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

WHEN OTHERS THEN

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

END;

✅ **Handles missing records and unknown errors.**

## **🔹 2️⃣ Stored Procedures, Functions & Packages**

### ****Q3: What is the difference between a Procedure and a Function?****

✅ **Function:** Returns a value, used in SELECT.  
✅ **Procedure:** Does not return a value, used for transactions.

🔹 **Example:** Function vs. Procedure

sql

CopyEdit

-- Function

CREATE OR REPLACE FUNCTION get\_salary(emp\_id NUMBER) RETURN NUMBER AS

v\_salary NUMBER;

BEGIN

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

RETURN v\_salary;

END get\_salary;

sql

CopyEdit

-- Procedure

CREATE OR REPLACE PROCEDURE update\_salary(emp\_id NUMBER, new\_salary NUMBER) AS

BEGIN

UPDATE employees SET salary = new\_salary WHERE employee\_id = emp\_id;

COMMIT;

END update\_salary;

✅ **Procedures modify data; functions return values.**

### ****Q4: What are Packages in PL/SQL?****

✅ **Packages** group related procedures, functions, and variables.  
✅ **Benefits:** Code reusability, encapsulation, and performance.

🔹 **Example:** Creating a Package

sql

CopyEdit

CREATE OR REPLACE PACKAGE emp\_pkg AS

FUNCTION get\_salary(emp\_id NUMBER) RETURN NUMBER;

PROCEDURE update\_salary(emp\_id NUMBER, new\_salary NUMBER);

END emp\_pkg;

sql

CopyEdit

CREATE OR REPLACE PACKAGE BODY emp\_pkg AS

FUNCTION get\_salary(emp\_id NUMBER) RETURN NUMBER IS

v\_salary NUMBER;

BEGIN

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

RETURN v\_salary;

END get\_salary;

PROCEDURE update\_salary(emp\_id NUMBER, new\_salary NUMBER) IS

BEGIN

UPDATE employees SET salary = new\_salary WHERE employee\_id = emp\_id;

COMMIT;

END update\_salary;

END emp\_pkg;

✅ **Package improves modularity and performance.**

## **🔹 3️⃣ Cursors & Bulk Data Processing**

### ****Q5: What are Explicit and Implicit Cursors?****

✅ **Implicit Cursor:** Automatically created for SELECT INTO.  
✅ **Explicit Cursor:** Manually declared and controlled.

🔹 **Example:** Explicit Cursor

sql

CopyEdit

DECLARE

CURSOR emp\_cursor IS SELECT employee\_id, name FROM employees;

v\_id employees.employee\_id%TYPE;

v\_name employees.name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO v\_id, v\_name;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_id || ', Name: ' || v\_name);

END LOOP;

CLOSE emp\_cursor;

END;

✅ **Explicit cursors allow fine control over query execution.**

### ****Q6: How do you improve cursor performance?****

✅ **Use BULK COLLECT and FORALL instead of row-by-row loops.**

🔹 **Example: Bulk Collect & FORALL for High Performance**

sql

CopyEdit

DECLARE

TYPE emp\_table\_type IS TABLE OF employees%ROWTYPE;

emp\_table emp\_table\_type;

BEGIN

-- Fetch all employees into collection

SELECT \* BULK COLLECT INTO emp\_table FROM employees WHERE department\_id = 10;

-- Process in bulk instead of row-by-row

FORALL i IN 1..emp\_table.COUNT

INSERT INTO employees\_backup VALUES emp\_table(i);

COMMIT;

END;

✅ **Improves performance when dealing with millions of rows.**

## **🔹 4️⃣ Triggers & Auditing**

### ****Q7: What is a Trigger? Types of Triggers?****

✅ **Triggers** are automatic PL/SQL blocks executed before/after INSERT, UPDATE, or DELETE.  
✅ **Types:**

* **Row-Level Trigger:** Executes once per row.
* **Statement-Level Trigger:** Executes once per statement.

🔹 **Example: Logging Salary Changes with a Trigger**

sql

CopyEdit

CREATE TABLE salary\_audit (

employee\_id NUMBER,

old\_salary NUMBER,

new\_salary NUMBER,

change\_date TIMESTAMP DEFAULT SYSTIMESTAMP

);

CREATE OR REPLACE TRIGGER trg\_salary\_update

BEFORE UPDATE ON employees

FOR EACH ROW

WHEN (NEW.salary <> OLD.salary)

BEGIN

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

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

END;

✅ **Automatically logs salary changes for auditing.**

## **🔹 5️⃣ Performance Tuning in PL/SQL**

### ****Q8: How do you optimize PL/SQL code for performance?****

✅ Use **BULK COLLECT & FORALL** instead of loops.  
✅ Use **Indexing** and **EXPLAIN PLAN** for SQL queries.  
✅ Use **Bind Variables** to prevent SQL parsing overhead.  
✅ Avoid **excessive context switching** between PL/SQL and SQL.

🔹 **Example: Using Bind Variables for Performance**

sql

CopyEdit

DECLARE

v\_salary NUMBER;

BEGIN

EXECUTE IMMEDIATE 'SELECT salary FROM employees WHERE employee\_id = :id'

INTO v\_salary USING 101;

END;

✅ **Bind variables reduce SQL parsing and improve performance.**

## **🔹 6️⃣ Advanced PL/SQL Interview Questions**

### ****Q9: How do you handle Deadlocks in PL/SQL?****

✅ **Deadlocks occur when two sessions block each other.**  
✅ **Solution:**

* **Commit frequently** to release locks.
* **Access tables in the same order across transactions.**
* **Use NOWAIT or SKIP LOCKED in SELECT FOR UPDATE.**

🔹 **Example: Avoiding Deadlocks**

sql

CopyEdit

SELECT \* FROM employees

WHERE department\_id = 10

FOR UPDATE SKIP LOCKED;

✅ **Prevents session blocking by skipping locked rows.**

### ****Q10: What are Materialized Views? How do they improve performance?****

✅ **Materialized Views** store precomputed query results for faster retrieval.  
✅ **Best used for Reporting & Aggregations.**

🔹 **Example: Creating a Materialized View**

sql

CopyEdit

CREATE MATERIALIZED VIEW sales\_summary AS

SELECT customer\_id, SUM(order\_total) AS total\_sales

FROM orders

GROUP BY customer\_id;

✅ **Speeds up analytics by precomputing totals.**