DAY- 4

📌 SQL SUBQUERIES & SET OPERATIONS - Study & Revision Notes

◇ Introduction

What is a Subquery?

A **subquery** is a **query inside another query**. It is executed first, and its result is used by the outer query.

Types of Subqueries

1. **Single-row subquery** → Returns **one value**.
2. **Multiple-row subquery** → Returns **multiple values**.
3. **Correlated subquery** → Inner query depends on the outer query.
4. **Nested subquery** → Subquery inside another subquery.
5. **Set operations** → Combines results from multiple queries.

🛠 1. Single-Row Subqueries

Returns **one value** and is used with comparison operators like =, <, >, etc.

☑ Find the Employee with the Lowest Salary

```
sql
CopyEdit
SELECT ENAME, SAL
FROM SCOTT.EMP
WHERE SAL = (SELECT MIN(SAL) FROM SCOTT.EMP);
```

🔹 **Explanation**:

- `(SELECT MIN(SAL) FROM SCOTT.EMP)` → Finds **lowest salary**.
- `SAL = (subquery result)` → Matches employee **with that salary**.

✅ Find Employees Earning More than the Average Salary

```
sql
CopyEdit
SELECT ENAME, SAL
FROM SCOTT.EMP
WHERE SAL > (SELECT AVG(SAL) FROM SCOTT.EMP);
```

🔹 **Explanation**:

- `(SELECT AVG(SAL) FROM SCOTT.EMP)` → Finds **average salary**.
- `SAL > (subquery result)` → Filters **employees earning above average**.

🛠 2. Multiple-Row Subqueries

Returns **multiple values** and uses operators like `IN`, `ANY`, `ALL`.

❌ Incorrect Query (Error: Subquery returns multiple rows)

```
sql
CopyEdit
SELECT *
FROM SCOTT.EMP
WHERE SAL > (SELECT SAL FROM SCOTT.EMP WHERE ENAME='ALLEN' OR
ENAME='JONES');
```

🔹 **Problem**:

- The subquery returns **multiple salaries**.
- The > operator **expects only one value**.

☑ Correct Query Using ALL

```
sql
CopyEdit
SELECT *
FROM SCOTT.EMP
WHERE SAL > ALL (SELECT SAL FROM SCOTT.EMP WHERE ENAME='ALLEN' OR
ENAME='JONES');
```

🔹 **Explanation**:

- `SAL > ALL(subquery)` → Finds employees **earning more than both Allen and Jones**.

☑ Find Employees Working in King's or Scott's Department

```
sql
CopyEdit
SELECT *
FROM SCOTT.EMP
WHERE DEPTNO = ANY (SELECT DEPTNO FROM SCOTT.EMP WHERE ENAME IN
('KING', 'SCOTT'));
```

🔹 **Explanation**:

- `(SELECT DEPTNO FROM SCOTT.EMP WHERE ENAME IN ('KING', 'SCOTT'))` → Finds **departments of King & Scott**.
- `DEPTNO = ANY (subquery result)` → Filters **employees working in those departments**.

🔹 **Alternate Query Using IN (More Readable)**

```
sql
CopyEdit
SELECT *
FROM SCOTT.EMP
WHERE DEPTNO IN (SELECT DEPTNO FROM SCOTT.EMP WHERE ENAME IN
('KING', 'SCOTT'));
```

🛠 3. Using Subqueries for Additional Columns

```
sql
CopyEdit
SELECT SCOTT.EMP.*,
       (SELECT MIN(SAL) FROM SCOTT.EMP) AS MIN_SAL,
       (SELECT MAX(SAL) FROM SCOTT.EMP) AS MAX_SAL
FROM SCOTT.EMP;
```

🔹 **Explanation**:

- Adds **MIN_SAL and MAX_SAL** as additional columns.
- The subqueries return **single values**, so they can be used in SELECT.

🛠 4. Using Subqueries in FROM Clause

☑ Find Salary Statistics per Department

```
sql
CopyEdit
SELECT *
FROM (SELECT DEPTNO,
             MIN(SAL) AS MIN_SAL,
             MAX(SAL) AS MAX_SAL,
             ROUND(AVG(SAL),2) AS AVG_SAL,
             SUM(SAL) AS TOTAL_SAL
      FROM SCOTT.EMP
      GROUP BY DEPTNO);
```

🔹 **Explanation**:

- Groups by DEPTNO and calculates **salary statistics**.
- This inner query is treated as a **temporary table**.

🛠 5. WITH Clause (Common Table Expressions - CTE)

```
sql
CopyEdit
WITH EMP_SUMMARY AS (
    SELECT DEPTNO,
           MIN(SAL) AS MIN_SAL,
           MAX(SAL) AS MAX_SAL,
           ROUND(AVG(SAL),2) AS AVG_SAL,
           SUM(SAL) AS TOTAL_SAL
    FROM SCOTT.EMP
    GROUP BY DEPTNO)
SELECT * FROM SCOTT.EMP JOIN EMP_SUMMARY USING(DEPTNO) ORDER BY
DEPTNO;
```

🔹 **Explanation**:

- `WITH EMP_SUMMARY AS (...)` creates a **temporary result set**.
- Joins the EMP table with EMP_SUMMARY on DEPTNO.

🛠 6. EXISTS vs NOT EXISTS

☑ Find Departments with Employees

```
sql
CopyEdit
SELECT *
FROM SCOTT.DEPT
WHERE EXISTS (SELECT * FROM SCOTT.EMP WHERE SCOTT.EMP.DEPTNO =
SCOTT.DEPT.DEPTNO);
```

🔹 **Explanation**:

- If employees exist in a department, it is **included** in the result.

☑ Find Departments Without Employees

```
sql
CopyEdit
SELECT *
FROM SCOTT.DEPT
WHERE NOT EXISTS (SELECT * FROM SCOTT.EMP WHERE SCOTT.EMP.DEPTNO =
SCOTT.DEPT.DEPTNO);
```

🔹 **Explanation**:

- If no employees exist in a department, it is **included**.

🛠 7. Correlated Subqueries

A **correlated subquery** depends on the outer query.

☑ Find Employees Earning More than the Average Salary of Their Department

```
sql
CopyEdit
SELECT *
FROM SCOTT.EMP e1
WHERE SAL > (SELECT AVG(e2.SAL) FROM SCOTT.EMP e2 WHERE e1.DEPTNO =
e2.DEPTNO);
```

🔹 **Explanation**:

- The inner query calculates **average salary for the same department**.
- Each employee's salary is compared **within their department**.

☑ Find the Highest-Paid Employee in Each Department

```
sql
CopyEdit
```

```
SELECT *
FROM SCOTT.EMP e1
WHERE e1.SAL = (SELECT MAX(e2.SAL) FROM SCOTT.EMP e2 WHERE e1.DEPTNO
= e2.DEPTNO);
```

🔹 **Explanation**:

- The subquery finds **max salary per department**.
- The outer query filters employees **who match the max salary**.

🛠 8. SET OPERATIONS

(Combining Results from Multiple Queries)

✅ Creating Sample Tables

```
sql
CopyEdit
CREATE TABLE emp1(empno NUMBER, ename VARCHAR2(20), deptno NUMBER);
INSERT INTO emp1 VALUES(1001, 'Ajit',10);
INSERT INTO emp1 VALUES(1002,'Bob',10);
INSERT INTO emp1 VALUES(1003,'John',20);
INSERT INTO emp1 VALUES(1004,'Els',20);

CREATE TABLE emp2(empno NUMBER, ename VARCHAR2(20), deptno NUMBER);
INSERT INTO emp2 VALUES(1001, 'John',20);
INSERT INTO emp2 VALUES(1002,'Elsa',20);
INSERT INTO emp2 VALUES(1003,'Raju',30);
INSERT INTO emp2 VALUES(1004,'Sunny',30);
```

🛠 9. UNION (Removes Duplicates)

```
sql
CopyEdit
SELECT * FROM emp1
UNION
```

```
SELECT * FROM emp2;
```

🔹 **Combines both tables, removes duplicates**.

🛠 10. UNION ALL (Includes Duplicates)

```
sql
CopyEdit
SELECT * FROM emp1
UNION ALL
SELECT * FROM emp2;
```

🔹 **Combines both tables, keeps duplicates**.

🛠 11. INTERSECT (Find Common Records)

```
sql
CopyEdit
SELECT * FROM emp1
INTERSECT
SELECT * FROM emp2;
```

🔹 **Returns only matching rows** from both tables.

🛠 12. MINUS (Find Unique Records in emp1)

```
sql
CopyEdit
SELECT * FROM emp1
MINUS
SELECT * FROM emp2;
```

🔹 **Returns records in `emp1` that are NOT in `emp2`.**

🚀 Summary

- **Subqueries** extract data dynamically.
- **Single-row vs Multiple-row subqueries**.
- **Correlated subqueries** iterate per row.
- **EXISTS vs NOT EXISTS** filters presence/absence.
- **Set operations (UNION, INTERSECT, MINUS)** combine results.

📌 SQL HIERARCHICAL RETRIEVAL - Study & Revision Notes

◇ Introduction

Hierarchical queries are used to retrieve data that is stored in a tree-like structure. In **Oracle SQL**, the `CONNECT BY` clause is used to define parent-child relationships.

💡 Key Concepts

- **`CONNECT BY PRIOR column = column`** → Defines the parent-child relationship.
- **`START WITH condition`** → Defines the root node.
- **`LEVEL`** → Represents the depth in the hierarchy.
- **`ORDER SIBLINGS BY`** → Sorts records within the same level.
- **`SYS_CONNECT_BY_PATH`** → Shows the path from root to each node.
- **`CONNECT_BY_ISLEAF`** → Identifies leaf nodes (employees with no subordinates).

🛠 1. Basic Hierarchical Query

☑ Find the Hierarchy Under 'KING'

```
sql
CopyEdit
SELECT EMPNO, ENAME, MGR
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING';
```

🔹 **Explanation**:

- **START WITH ENAME='KING'** → Starts hierarchy from King.
- **CONNECT BY PRIOR EMPNO = MGR** → Builds hierarchy by linking MGR (manager ID) to EMPNO (employee ID).
- Returns **all employees reporting directly or indirectly** to King.

🛠 2. Adding Levels in Hierarchy

```
sql
CopyEdit
SELECT EMPNO, ENAME, MGR, LEVEL
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING'
ORDER BY LEVEL;
```

🔹 **Explanation**:

- LEVEL → Represents the depth of the employee in the hierarchy.
- ORDER BY LEVEL → Sorts the output based on levels.

✅ **Output Example**:

EMPNO	ENAME	MGR	LEVEL
7839	KING	NULL	1
7566	JONES	7839	2

7698	BLAKE	7839	2
7782	CLARK	7839	2

🛠 3. Sorting Siblings Alphabetically

sql
CopyEdit

```
SELECT EMPNO, ENAME, MGR, LEVEL
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING'
ORDER SIBLINGS BY ENAME;
```

🔹 **Explanation**:

- `ORDER SIBLINGS BY ENAME` → Sorts **employees at the same hierarchy level** alphabetically.

✅ **Example Output (Sorted Alphabetically by Name)**

EMPNO	ENAME	MGR	LEVEL
7839	KING	NULL	1
7698	BLAKE	7839	2
7782	CLARK	7839	2
7566	JONES	7839	2

🛠 4. Sorting Siblings in Descending Order

sql
CopyEdit

```
SELECT EMPNO, ENAME, MGR, LEVEL
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING'
ORDER SIBLINGS BY ENAME DESC;
```

🔹 **Explanation**:

- `ORDER SIBLINGS BY ENAME DESC` → Sorts **siblings in reverse alphabetical order**.

✅ **Example Output (Sorted in Reverse Order)**

EMPNO	ENAME	MGR	LEVEL
7839	KING	NULL	1
7566	JONES	7839	2
7782	CLARK	7839	2
7698	BLAKE	7839	2

🛠 5. Displaying the Path in Hierarchy

```
sql
CopyEdit
SELECT SYS_CONNECT_BY_PATH(ENAME, '/')
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING';
```

🔹 **Explanation**:

- **`SYS_CONNECT_BY_PATH(column, delimiter)`** → Displays **full path** from root to each employee.
- **Example Output**:

```
swift
CopyEdit
/KING
/KING/JONES
/KING/JONES/SCOTT
/KING/BLAKE
/KING/CLARK
```

📌 **Use Case:** Helps visualize reporting structure.

🛠 6. Identifying Leaf Nodes

```
sql
CopyEdit
SELECT ENAME, CONNECT_BY_ISLEAF
FROM SCOTT.EMP
CONNECT BY PRIOR EMPNO = MGR
START WITH ENAME = 'KING';
```

🔹 **Explanation**:

- `CONNECT_BY_ISLEAF` → Returns 1 if the employee has **no subordinates**, otherwise `0`.

✅ **Example Output**:

ENAME	CONNECT_BY_ISLEAF
KING	0
JONES	0
SCOTT	1
BLAKE	0
CLARK	1

📌 **Use Case:** Helps find employees who are **not managers**.

🚀 Summary

- **Hierarchical queries** are useful for **organizational structures**.
- **`START WITH`** → Defines the **root** of the hierarchy.
- **`CONNECT BY PRIOR`** → Establishes **parent-child relationships**.
- **`LEVEL`** → Represents **depth** in the hierarchy.
- **`ORDER SIBLINGS BY`** → Sorts **employees at the same level**.
- **`SYS_CONNECT_BY_PATH`** → Displays **full hierarchy path**.
- **`CONNECT_BY_ISLEAF`** → Identifies **employees with no subordinates**.

📌 SQL Views - Study & Revision Notes

◇ Introduction to Views

- **What is a View?**
  - A **view** is a **virtual table** based on the result set of an SQL query.
  - It does **not store data** itself but displays data stored in other tables.
- **Why Use Views?**
  - **Fine-grained access control:** Restrict access to specific data.
  - **Simplify complex queries:** Abstract complicated joins or calculations.
  - **Data security:** Hide sensitive data.
- **Types of Views**
  - **Simple View:** Based on **one table**, no functions, aggregations, or joins.
  - **Complex View:** Based on **multiple tables** or uses functions, aggregations, or joins.

🛠 1. Creating and Inserting into a Table

Create EMP Table

```
sql
CopyEdit
CREATE TABLE EMP(
    EMPNO NUMBER(4,0) NOT NULL,
    ENAME VARCHAR2(10),
    JOB VARCHAR2(9),
    MGR NUMBER(4,0),
    HIREDATE DATE,
    SAL NUMBER(7,2),
    COMM NUMBER(7,2),
    DEPTNO NUMBER(2,0)
);
```

🔹 **Explanation:**

- **Defines the EMP table** with columns for employee details.

Insert Data from SCOTT.EMP

```
sql
CopyEdit
INSERT INTO EMP (SELECT * FROM SCOTT.EMP);
```

🔹 **Explanation:**

- **Copies data** from the `SCOTT.EMP` table into the new EMP table.

🛠 2. Creating a Simple View

Create a View for Department 10

```
sql
CopyEdit
CREATE VIEW DEPT_10_VIEW AS SELECT * FROM EMP WHERE DEPTNO=10;
```

🔹 **Explanation:**

- **Simple view** showing all employees **in department 10**.

Modify the View

```
sql
CopyEdit
CREATE OR REPLACE VIEW DEPT_10_VIEW AS SELECT * FROM EMP WHERE
DEPTNO=10;
```

🔹 **Explanation:**

- **Replaces** the existing view if it exists, or **creates** it if not.

❌ Attempt to Alter a View

```
sql
CopyEdit
```

```
ALTER VIEW DEPT_10_VIEW ADD TEST_COL NUMBER;
```

❌ **Error:**

- **Views cannot be altered** to add columns. You must recreate them.

🛠 3. Performing DML Operations on Views

Insert into the View

```
sql
CopyEdit
INSERT INTO DEPT_10_VIEW VALUES(9999, 'DNYANESH', 'ANALYST', 7839,
'04-JUL-22', 1000, NULL, 10);
```

🔹 **Explanation:**

- Inserts a **new employee** into DEPT_10_VIEW.
- **Changes reflect** in the base EMP table since it's a **simple view**.

❌ Inserting with Incorrect Department

```
sql
CopyEdit
INSERT INTO DEPT_10_VIEW VALUES(8888, 'RAHUL', 'ANALYST', 7839, '04-
JUL-22', 1000, NULL, 20);
```

❌ **Error:**

- Department 20 **violates the view condition** (DEPTNO=10).

Delete from the View

```
sql
CopyEdit
DELETE FROM DEPT_10_VIEW WHERE EMPNO=8888;
```

🔹 **Explanation:**

- **Deletes the record** both from the view and the base table.

🛠 4. Restricting DML with View Options

Read-Only View

```
sql
CopyEdit
CREATE OR REPLACE VIEW DEPT_10_VIEW AS SELECT * FROM EMP WHERE
DEPTNO=10 WITH READ ONLY;
```

🔹 **Explanation:**

- **Prevents any DML** operations (INSERT, UPDATE, DELETE).

❌ Attempting to Insert into Read-Only View

```
sql
CopyEdit
INSERT INTO DEPT_10_VIEW VALUES(9999, 'DNYANESH', 'ANALYST', 7839,
'04-JUL-22', 1000, NULL, 10);
```

❌ **Error:**

- **Cannot perform DML** on a read-only view.

Using `WITH CHECK OPTION`

```
sql
CopyEdit
CREATE OR REPLACE VIEW DEPT_10_VIEW AS SELECT * FROM EMP WHERE
DEPTNO=10 WITH CHECK OPTION;
```

🔹 **Explanation:**

- Ensures that **all DML operations** conform to the **view's WHERE clause**.

❌ Inserting a Record that Violates the View Condition

```
sql
CopyEdit
INSERT INTO DEPT_10_VIEW VALUES(7777, 'RAHUL', 'ANALYST', 7839, '04-JUL-22', 1000, NULL, 20);
```

❌ **Error:**

- **Rejected** because DEPTNO=20 does **not match** the view's condition (DEPTNO=10).

🛠 5. Updating Data via Views

Updating Salary in the Base Table

```
sql
CopyEdit
UPDATE EMP SET SAL = SAL * 1.10 WHERE ENAME = 'KING';
```

- **Increases KING's salary by 10%.**

Verify Update in Both View and Table

```
sql
CopyEdit
SELECT * FROM DEPT_10_VIEW WHERE ENAME = 'KING';
SELECT * FROM EMP WHERE ENAME = 'KING';
```

- **Check both the view and the table** to confirm the update.

🛠 6. Creating a Complex View

Create an Aggregate View

```
sql
CopyEdit
```

```
CREATE OR REPLACE VIEW EMP_SUMMARY AS
SELECT DEPTNO,
       COUNT(*) AS TOTAL_EMPLOYEES,
       MIN(SAL) AS MIN_SAL,
       MAX(SAL) AS MAX_SAL,
       SUM(SAL) AS TOTAL_SAL,
       AVG(SAL) AS AVG_SAL
FROM EMP
GROUP BY DEPTNO;
```

🔹 **Explanation:**

- **Aggregates data** by department.
- **Complex view** due to aggregation.

✖ Attempt to Update a Complex View

```
sql
CopyEdit
UPDATE EMP_SUMMARY SET TOTAL_EMPLOYEES = 10 WHERE DEPTNO = 30;
```

❌ **Error:**

- **DML operations** are **not allowed** on views with aggregations.

🛠 7. Combining Data from Multiple Tables

Join View

```
sql
CopyEdit
CREATE VIEW EMP_DEPT_DATA AS
SELECT * FROM SCOTT.EMP JOIN SCOTT.DEPT USING(DEPTNO);
```

🔹 **Explanation:**

- **Combines employee and department data** in one view.
- **Useful** for seeing related data together.

🚀 Summary

- **Views** are flexible tools for data abstraction and security.
- **Simple Views:** Allow basic DML operations.
- **Complex Views:** Typically **read-only** due to aggregations or joins.
- **WITH CHECK OPTION:** Ensures data integrity in DML operations.
- **Read-Only Views:** Prevent any data modifications.

📌 SQL Query Optimization & EXPLAIN PLAN - Study & Revision Notes

This guide covers **query execution plans**, **indexes**, and **performance optimization** using `EXPLAIN PLAN` in Oracle SQL.

◇ Understanding `EXPLAIN PLAN`

- `EXPLAIN PLAN` **shows how Oracle executes a query**.
- Helps in **query tuning** by identifying **full table scans, index usage, and joins**.
- **DBMS_XPLAN.DISPLAY()** retrieves the **execution plan** from memory.

🛠 1. Basic Query Execution

```
sql
CopyEdit
SELECT * FROM EMP;
```

🔹 **Retrieves all rows** from the EMP table.

Check Execution Plan

```
sql
CopyEdit
EXPLAIN PLAN FOR SELECT * FROM EMP WHERE DEPTNO=10;
SELECT * FROM DBMS_XPLAN.DISPLAY();
```

🔹 **Breakdown:**

1. **`EXPLAIN PLAN FOR`** - Prepares the query execution plan.
2. **`DBMS_XPLAN.DISPLAY()`** - Displays the execution plan.

📌 **Key Observations:**

- If there is **no index on DEPTNO**, the query may **perform a full table scan**.
- A **full table scan is slow** when the table has many rows.

🛠 2. Checking Synonyms

```
sql
CopyEdit
SELECT * FROM USER_SYNONYMS;
```

🔹 **Lists all synonyms** (aliases) in the user's schema.

🛠 3. Creating and Dropping an Index

```
sql
CopyEdit
CREATE INDEX DEPT_IDX ON EMP(DEPTNO);
```

🔹 **Creates an index** on the DEPTNO column.

```
sql
CopyEdit
```

```
DROP INDEX DEPT_IDX;
```

🔹 **Removes the index.**

📌 **Why Create an Index?**

- **Speeds up queries** that use `WHERE DEPTNO=10` by **avoiding full table scans**.
- **Indexes work best** when filtering a small subset of data.

🛠 4. Checking Query Optimization with Index

```
sql
CopyEdit
EXPLAIN PLAN FOR SELECT * FROM EMP WHERE DEPTNO=10;
SELECT * FROM DBMS_XPLAN.DISPLAY();
```

🔹 **After creating an index, this query should use an INDEX SCAN instead of a FULL TABLE SCAN.**

🛠 5. Checking Execution Plan for JOB Column

```
sql
CopyEdit
EXPLAIN PLAN FOR SELECT * FROM EMP WHERE JOB='MANAGER';
SELECT * FROM DBMS_XPLAN.DISPLAY();
```

📌 **Key Notes:**

- If `JOB` is **not indexed**, it may still perform a **full table scan**.
- **Indexing JOB could improve performance** if filtering on this column is frequent.

🛠 6. Checking Execution Plan for `CROSS JOIN`

```
sql
```

```
CopyEdit
EXPLAIN PLAN FOR SELECT * FROM SCOTT.EMP CROSS JOIN SCOTT.DEPT;
SELECT * FROM DBMS_XPLAN.DISPLAY();
```

📌 **Key Observations:**

- **CROSS JOIN produces a Cartesian product** (every row in EMP is combined with every row in DEPT).
- **Inefficient for large tables**. Use **INNER JOIN** or filters to optimize.

🛠 7. Checking Execution Plan for Aggregations

```
sql
CopyEdit
EXPLAIN PLAN FOR
SELECT DEPTNO, COUNT(*) AS TOTAL_EMPLOYEES,
       MIN(SAL) AS MIN_SAL,
       MAX(SAL) AS MAX_SAL,
       SUM(SAL) AS TOTAL_SAL,
       AVG(SAL) AS AVG_SAL
FROM EMP
GROUP BY DEPTNO;
SELECT * FROM DBMS_XPLAN.DISPLAY();
```

📌 **Key Observations:**

- **Uses GROUP BY**, which may trigger a **SORT operation**.
- **Indexes may not help much** in aggregation queries.
- **Adding indexes on frequently grouped columns** (DEPTNO) may improve performance.

🚀 Summary

- **EXPLAIN PLAN** helps analyze query performance.
- **Indexes** improve filtering but are ineffective for aggregation.
- **Avoid CROSS JOINs** unless necessary.

- **Always check execution plans** before optimizing queries.