Oracle - Advanced SQL

Joins, Aggregations, Subqueries & Set Operators

By
Narasimha Rao T
Microsoft.Net FSD Trainer
Professional Development Trainer
tnrao.trainer@gmail.com

1. What are Joins?

Definition:

Joins are used to combine rows from two or more tables based on a related column between them. They allow you to retrieve data from multiple tables in a single query.

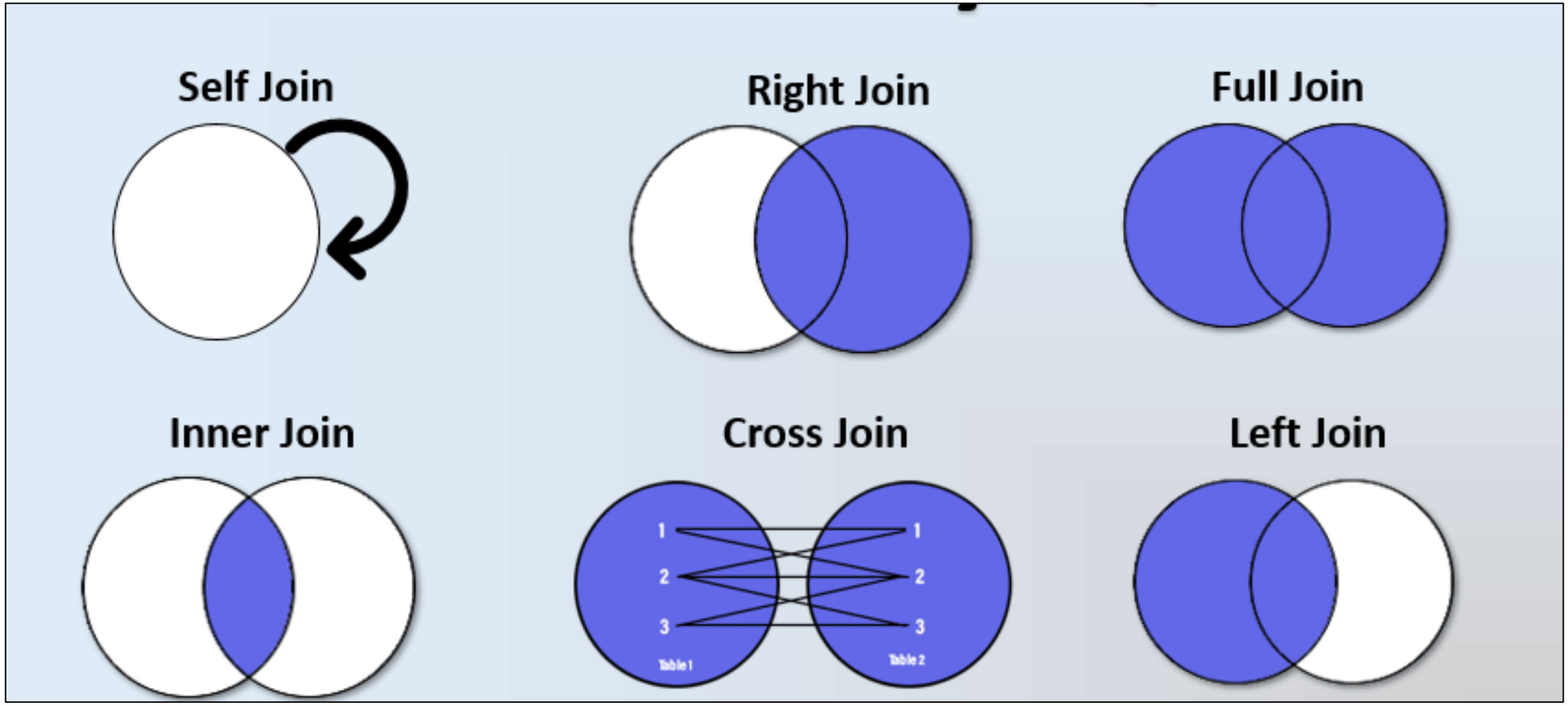
Why We Need Joins:

- Data is often spread across multiple tables (normalization)
- Avoids data redundancy
- Maintains data integrity
- Enables complex data retrieval

2. Advantages of Using Joins

Benefits:

1. **Data Integration**: Combine related data from multiple tables
2. **Performance**: Single query vs multiple queries
3. **Data Consistency**: Maintains relationships between tables
4. **Flexibility**: Complex data retrieval capabilities
5. **Efficiency**: Reduces application-level processing


Self Join
Right Join
Full Join
Inner Join
Cross Join
Left Join
1
2
3
Table 1
1
2
3
Table 2

3. Types of JOINs

INNER JOIN

Returns only the rows that have matching values in both tables.

```
-- Basic INNER JOIN
SELECT e.employee_id, e.first_name, e.last_name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;

-- Equivalent using WHERE clause (older syntax)
SELECT e.employee_id, e.first_name, e.last_name, d.department_name
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

LEFT OUTER JOIN

Returns all rows from the left table and matched rows from the right table. Unmatched rows from right table contain NULL.

```
-- LEFT JOIN - All employees, even if no department
SELECT e.employee_id, e.first_name, e.last_name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;

-- LEFT JOIN with WHERE to find employees without departments
SELECT e.employee_id, e.first_name, e.last_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id
WHERE d.department_id IS NULL;
```

RIGHT OUTER JOIN

Returns all rows from the right table and matched rows from the left table. Unmatched rows from left table contain NULL.

```sql
-- RIGHT JOIN - All departments, even if no employees
SELECT d.department_name, e.employee_id, e.first_name, e.last_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;

-- RIGHT JOIN with WHERE to find departments without employees
SELECT d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id
WHERE e.employee_id IS NULL;
```

FULL OUTER JOIN

Returns all rows when there's a match in either left or right table.

```
-- FULL OUTER JOIN
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d ON e.department_id = d.department_id;
```

SELF JOIN

Joins a table to itself. Useful for hierarchical data or comparing rows within the same table.

```sql
-- Find employees and their managers
SELECT e.employee_id, e.first_name AS employee_name,
       m.first_name AS manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;

-- Find employees in the same department
SELECT e1.first_name AS employee1, e2.first_name AS employee2, e1.department_id
FROM employees e1
JOIN employees e2 ON e1.department_id = e2.department_id
WHERE e1.employee_id < e2.employee_id;  -- Avoids duplicates and self-comparison
```

CROSS JOIN

Returns Cartesian product of both tables (every row from first table combined with every row from second table).

```
-- CROSS JOIN
SELECT e.first_name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

4. What are Aggregations?

Definition:

Aggregation operations process multiple rows to return a single summary value. They are used to perform calculations on sets of rows.

Common Use Cases:

- Counting records
- Calculating averages
- Finding maximum/minimum values
- Summing values
- Grouping data for analysis

5. Aggregate Functions and GROUP BY

Common Aggregate Functions:

```sql
-- COUNT - Count number of rows
SELECT COUNT(*) FROM employees;
SELECT COUNT(department_id) FROM employees;  -- Excludes NULLs
SELECT COUNT(DISTINCT department_id) FROM employees;

-- SUM - Calculate total
SELECT SUM(salary) FROM employees;
SELECT SUM(salary) FROM employees WHERE department_id = 50;

-- AVG - Calculate average
SELECT AVG(salary) FROM employees;
SELECT AVG(NVL(salary, 0)) FROM employees;  -- Handle NULLs
```

```sql
-- MAX/MIN - Find maximum/minimum values
SELECT MAX(salary), MIN(salary) FROM employees;
SELECT MAX(hire_date), MIN(hire_date) FROM employees;
```

GROUP BY Clause

Groups rows that have the same values into summary rows.

```sql
-- Basic GROUP BY
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id;

-- Multiple columns GROUP BY
SELECT department_id, job_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id, job_id;
```

```sql
-- GROUP BY with aggregate functions
SELECT department_id,
       COUNT(*) as employee_count,
       AVG(salary) as avg_salary,
       MAX(salary) as max_salary,
       MIN(salary) as min_salary
FROM employees
GROUP BY department_id;
```

HAVING Clause

Filters groups based on aggregate conditions (WHERE filters rows, HAVING filters groups).

```sql
-- HAVING with aggregate condition
SELECT department_id, COUNT(*) as employee_count
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;

-- Multiple HAVING conditions
SELECT department_id, AVG(salary) as avg_salary
FROM employees
GROUP BY department_id
HAVING AVG(salary) > 5000 AND COUNT(*) > 3;
```

```sql
-- WHERE and HAVING together
SELECT department_id, AVG(salary) as avg_salary
FROM employees
WHERE hire_date > DATE '2020-01-01'
GROUP BY department_id
HAVING AVG(salary) > 6000;
```

6. Working with Filtering and Sorting

Advanced Filtering Techniques:

```sql
-- Multiple conditions
SELECT * FROM employees
WHERE salary BETWEEN 5000 AND 10000
   AND department_id IN (10, 20, 30)
   AND hire_date > DATE '2015-01-01';

-- Pattern matching with LIKE
SELECT * FROM employees
WHERE first_name LIKE 'A%';         -- Starts with A
WHERE first_name LIKE '%a%';        -- Contains 'a'
WHERE first_name LIKE '_a%';        -- Second letter is 'a'
WHERE first_name LIKE 'J_n%';       -- Starts with J, third letter n
```

```sql
-- NULL handling
SELECT * FROM employees
WHERE commission_pct IS NOT NULL;

-- EXISTS with subquery
SELECT * FROM departments d
WHERE EXISTS (
    SELECT 1 FROM employees e
    WHERE e.department_id = d.department_id
);
```

Advanced Sorting:

```sql
-- Basic sorting
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary DESC;

-- Multiple column sorting
SELECT department_id, first_name, last_name, salary
FROM employees
ORDER BY department_id ASC, salary DESC;

-- Sorting by expression
SELECT first_name, last_name, salary, commission_pct
FROM employees
ORDER BY NVL(commission_pct, 0) DESC;
```

```sql
-- Sorting by column position
SELECT first_name, last_name, salary
FROM employees
ORDER BY 3 DESC, 2 ASC;  -- Sort by 3rd column (salary), then 2nd (last_name)

-- CASE statement in ORDER BY
SELECT first_name, last_name, department_id, salary
FROM employees
ORDER BY
    CASE
        WHEN department_id = 50 THEN 1
        WHEN department_id = 60 THEN 2
        ELSE 3
    END,
    salary DESC;
```

7. What are Subqueries?

Definition:

A subquery (inner query or nested query) is a query within another SQL query. It's used to return data that will be used in the main query.

Types of Subqueries:

1. **Single-row subquery**: Returns one row
2. **Multiple-row subquery**: Returns multiple rows

8. Examples of Subqueries

Single-Row Subqueries:

```sql
-- Employees with salary greater than average
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Employee with highest salary
SELECT first_name, last_name, salary
FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);
```

Multiple-Row Subqueries:

```sql
-- Using IN
SELECT first_name, last_name, department_id
FROM employees
WHERE department_id IN (
    SELECT department_id
    FROM departments
    WHERE location_id = 1700
);

-- Using ANY/SOME
SELECT first_name, last_name, salary
FROM employees
WHERE salary > ANY (
    SELECT salary
    FROM employees
    WHERE department_id = 50
);
```

Correlated Subqueries:

```sql
-- Employees whose salary is greater than department average
SELECT e1.first_name, e1.last_name, e1.salary, e1.department_id
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
    WHERE e2.department_id = e1.department_id
);

-- Departments that have employees
SELECT department_name
FROM departments d
WHERE EXISTS (
    SELECT 1
    FROM employees e
    WHERE e.department_id = d.department_id
);
```

9. What are Set Operators?

Definition:

Set operators combine the results of two or more SELECT statements. All SELECT statements must have the same number of columns and compatible data types.

Types of Set Operators:

UNION

Combines results and removes duplicates.

```
-- Employees and managers from different tables
SELECT employee_id, first_name, last_name, 'Employee' as type
FROM employees
UNION
SELECT manager_id, first_name, last_name, 'Manager' as type
FROM managers;
```

UNION ALL

Combines results and keeps all duplicates.

```
-- All locations including duplicates
SELECT city FROM old_locations
UNION ALL
SELECT city FROM new_locations;
```

INTERSECT

Returns only common rows from both queries.

```
-- Products available in both warehouses
SELECT product_id FROM warehouse_a
INTERSECT
SELECT product_id FROM warehouse_b;
```

MINUS (EXCEPT in other databases)

Returns rows from first query that are not in second query.

```
-- Products only in warehouse A
SELECT product_id FROM warehouse_a
MINUS
SELECT product_id FROM warehouse_b;
```

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com