SQL Joins Guide: From Basics to Advanced 🔄

Table of Contents

Basic Join Types

Sample Tables

We'll use these tables for our examples:

```sql
-- Customers table
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100)
);

-- Orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10,2)
);

-- Sample data
INSERT INTO customers VALUES
(1, 'John Doe', 'john@example.com'),
(2, 'Jane Smith', 'jane@example.com'),
(3, 'Bob Wilson', 'bob@example.com');

INSERT INTO orders VALUES
(1, 1, '2024-01-01', 100.00),
(2, 1, '2024-01-15', 200.00),
(3, 2, '2024-01-20', 150.00);
```

1. INNER JOIN

Returns only matching records from both tables.

```
-- Basic INNER JOIN
SELECT
    c.name,
    o.order_id,
    o.amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;

-- INNER JOIN with multiple conditions
SELECT
    c.name,
    o.order_id,
    o.amount
FROM customers c
INNER JOIN orders o
    ON c.customer_id = o.customer_id
    AND o.amount > 100;
```

2. LEFT JOIN

Returns all records from left table and matching records from right table.

```
-- Basic LEFT JOIN
SELECT
    c.name,
    COALESCE(o.order_id, 'No Order') as order_id,
    COALESCE(o.amount, 0) as amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;

-- LEFT JOIN with filtering
SELECT
    c.name,
    COUNT(o.order_id) as order_count
FROM customers c
```

```sql
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.name;
```

3. RIGHT JOIN

Returns all records from right table and matching records from left table.

```sql
-- Basic RIGHT JOIN
SELECT
    COALESCE(c.name, 'Unknown Customer') as customer_name,
    o.order_id,
    o.amount
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;

-- RIGHT JOIN with conditions
SELECT
    COALESCE(c.name, 'Unknown Customer') as customer_name,
    o.order_id,
    o.amount
FROM customers c
RIGHT JOIN orders o
    ON c.customer_id = o.customer_id
WHERE o.amount > 100;
```

4. FULL JOIN

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

```sql
-- Basic FULL JOIN
SELECT
    COALESCE(c.name, 'Unknown Customer') as customer_name,
    COALESCE(o.order_id, 'No Order') as order_id
FROM customers c
FULL JOIN orders o ON c.customer_id = o.customer_id;

-- FULL JOIN with COALESCE
SELECT
    COALESCE(c.name, 'Unknown Customer') as customer_name,
    COALESCE(o.order_id::TEXT, 'No Order') as order_id,
    COALESCE(o.amount, 0) as amount
```

```sql
FROM customers c
FULL JOIN orders o ON c.customer_id = o.customer_id;
```

Advanced Join Types

5. CROSS JOIN

Creates a Cartesian product of both tables.

```sql
-- Basic CROSS JOIN
SELECT
    c.name,
    p.product_name
FROM customers c
CROSS JOIN products p;

-- CROSS JOIN with filtering
SELECT
    c.name,
    p.product_name
FROM customers c
CROSS JOIN products p
WHERE p.category = 'Electronics';
```

6. SELF JOIN

Joins a table with itself.

```sql
-- Employee hierarchy example
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100),
    manager_id INT
);

-- Self join to find employee-manager relationships
SELECT
    e1.name as employee,
    e2.name as manager
FROM employees e1
LEFT JOIN employees e2 ON e1.manager_id = e2.emp_id;
```

```
-- Find employees with same manager
SELECT
    e1.name as employee1,
    e2.name as employee2,
    m.name as manager
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.manager_id
JOIN employees m ON e1.manager_id = m.emp_id
WHERE e1.emp_id < e2.emp_id;
```

Set Theory Operations

7. Set Operations

```
-- UNION: Combines results and removes duplicates
SELECT customer_id FROM orders_2023
UNION
SELECT customer_id FROM orders_2024;

-- UNION ALL: Combines results including duplicates
SELECT amount FROM north_sales
UNION ALL
SELECT amount FROM south_sales;

-- INTERSECT: Returns only common rows
SELECT customer_id FROM active_customers
INTERSECT
SELECT customer_id FROM premium_members;

-- EXCEPT: Returns rows in first set but not in second
SELECT customer_id FROM all_customers
EXCEPT
SELECT customer_id FROM opted_out_customers;
```

8. Subqueries

```
-- Subquery in SELECT
SELECT
    customer_id,
    amount,
    (SELECT AVG(amount) FROM orders) as avg_order_amount
```

```sql
FROM orders;

-- Subquery in WHERE
SELECT name
FROM customers
WHERE customer_id IN (
    SELECT customer_id
    FROM orders
    WHERE amount > 1000
);

-- Correlated subquery
SELECT
    customer_id,
    amount
FROM orders o1
WHERE amount > (
    SELECT AVG(amount)
    FROM orders o2
    WHERE o2.customer_id = o1.customer_id
);
```

9. Semi Joins

```sql
-- EXISTS (Semi Join)
SELECT name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
    AND o.amount > 1000
);

-- NOT EXISTS (Anti Semi Join)
SELECT name
FROM customers c
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
);

-- IN as Semi Join
```

```sql
SELECT name
FROM customers
WHERE customer_id IN (
    SELECT DISTINCT customer_id
    FROM orders
    WHERE amount > 1000
);
```

Best Practices 📌

1. Always use meaningful table aliases
2. Include appropriate JOIN conditions
3. Handle NULL values with COALESCE or IFNULL
4. Use indexes on join columns
5. Consider query performance with multiple joins
6. Use appropriate join type for your use case
7. Test queries with sample data
8. Document complex joins

Common Pitfalls ⚠️

1. Forgetting join conditions (creating Cartesian products)
2. Incorrect join type selection
3. Not handling NULL values
4. Missing indexes on join columns
5. Overcomplicating joins when simpler solutions exist
6. Not considering data volume in CROSS JOINs
7. Incorrect use of correlation in subqueries