SQL Joins and Aggregation Functions Guide

1. Aggregation Functions 📊

Common Aggregation Functions

- **COUNT()**: Counts number of rows or non-null values
- **SUM()**: Calculates sum of numeric values
- **AVG()**: Calculates arithmetic mean
- **MIN()**: Finds minimum value
- **MAX()**: Finds maximum value
- **STDDEV()**: Calculates standard deviation
- **VARIANCE()**: Calculates statistical variance

Examples

```sql
-- Basic aggregation
SELECT
    COUNT(*) as total_orders,
    SUM(amount) as total_amount,
    AVG(amount) as average_order,
    MIN(amount) as smallest_order,
    MAX(amount) as largest_order
FROM orders;

-- Grouping with aggregation
SELECT
    category,
    COUNT(*) as items,
    AVG(price) as avg_price
FROM products
GROUP BY category
HAVING COUNT(*) > 5;
```

2. SQL Joins Overview 🔄

Types of Joins

1. **INNER JOIN**

2. **LEFT (OUTER) JOIN**
3. **RIGHT (OUTER) JOIN**
4. **FULL (OUTER) JOIN**
5. **CROSS JOIN**
6. **SELF JOIN**

Inner Join

Matches records from both tables based on join condition.

```sql
SELECT customers.name, orders.order_date
FROM customers
INNER JOIN orders
    ON customers.id = orders.customer_id;
```

Left (Outer) Join

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

```sql
SELECT products.name, reviews.rating
FROM products
LEFT JOIN reviews
    ON products.id = reviews.product_id;
```

Right (Outer) Join

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

```sql
SELECT employees.name, departments.dept_name
FROM employees
RIGHT JOIN departments
    ON employees.dept_id = departments.id;
```

Full (Outer) Join

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

```sql
SELECT students.name, courses.course_name
FROM students
FULL OUTER JOIN enrollments
    ON students.id = enrollments.student_id
FULL OUTER JOIN courses
    ON enrollments.course_id = courses.id;
```

Cross Join

Creates Cartesian product of both tables.

```sql
SELECT products.name, categories.category_name
FROM products
CROSS JOIN categories;
```

Self Join

Joins a table with itself.

```sql
SELECT e1.name as employee, e2.name as manager
FROM employees e1
LEFT JOIN employees e2
    ON e1.manager_id = e2.id;
```

3. Set Operations 🔍

UNION

Combines results of two SELECT statements and removes duplicates.

```sql
SELECT product_id FROM orders_2023
UNION
SELECT product_id FROM orders_2024;
```

UNION ALL

Combines results including duplicates.

```sql
SELECT amount FROM sales_north
UNION ALL
SELECT amount FROM sales_south;
```

INTERSECT

Returns only rows that appear in both result sets.

```sql
SELECT customer_id FROM active_accounts
INTERSECT
SELECT customer_id FROM premium_subscribers;
```

EXCEPT

Returns rows from first query that don't appear in second query's result.

```sql
SELECT product_id FROM all_products
EXCEPT
SELECT product_id FROM discontinued_products;
```

4. Semi Joins 🎯

EXISTS (Semi Join)

Checks for existence of related records.

```sql
SELECT customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.id
    AND o.amount > 1000
);
```

NOT EXISTS (Anti Semi Join)

Finds records without related records.

```sql
SELECT product_name
FROM products p
WHERE NOT EXISTS (
    SELECT 1
    FROM order_items oi
    WHERE oi.product_id = p.id
);
```

IN (Alternative Semi Join)

Another way to write semi joins.

```sql
SELECT supplier_name
FROM suppliers
WHERE id IN (
    SELECT supplier_id
    FROM products
    WHERE stock > 0
);
```

5. Practice Examples 💡

Complex Join Example

```sql
-- Find customers who have placed orders with total amount > average order amount
SELECT
    c.customer_name,
    COUNT(o.id) as order_count,
    SUM(o.amount) as total_spent
FROM customers c
INNER JOIN orders o ON c.id = o.customer_id
GROUP BY c.customer_name
HAVING SUM(o.amount) > (
    SELECT AVG(amount)
    FROM orders
)
ORDER BY total_spent DESC;
```

Multiple Joins with Aggregation

```sql
-- Get product sales statistics by category
SELECT
    c.category_name,
    COUNT(DISTINCT p.id) as product_count,
    COUNT(o.id) as order_count,
    AVG(o.amount) as avg_order_amount
FROM categories c
LEFT JOIN products p ON c.id = p.category_id
LEFT JOIN order_items oi ON p.id = oi.product_id
LEFT JOIN orders o ON oi.order_id = o.id
GROUP BY c.category_name
ORDER BY order_count DESC;
```

6. Best Practices 📌

1. Always specify JOIN type explicitly
2. Use table aliases for better readability
3. Include proper JOIN conditions
4. Be careful with CROSS JOINs on large tables
5. Use appropriate indexes on join columns
6. Consider performance with multiple joins
7. Use GROUP BY with aggregation functions
8. Include HAVING for filtering aggregated results

7. Common Pitfalls to Avoid ⚠️

1. Forgetting WHERE clause in outer joins
2. Incorrect join conditions leading to Cartesian products
3. Not handling NULL values properly
4. Mixing up LEFT and RIGHT joins
5. Forgetting GROUP BY when using aggregations
6. Using DISTINCT unnecessarily
7. Not considering index usage in join conditions