The Most Confusing SQL Functions

Made clear and easy to understand

Prepared by: Asfaw Gedamu

Do you find the following SQL functions confusing?

1. RANK vs DENSE_RANK vs ROW_NUMBER

2. HAVING vs WHERE Clause

3. UNION vs UNION ALL

4. JOIN vs UNION

5. DELETE vs DROP vs TRUNCATE

6. CTE vs TEMP TABLE

7. SUBQUERIES vs CTE

8. ISNULL vs COALESCE

9. INTERSECT vs INNER JOIN

10. EXCEPT vs NOT IN

11. INNER JOIN vs LEFT JOIN vs RIGHT JOIN vs FULL JOIN

12. LAG() vs LEAD() Functions

You're not alone! Below is a concise guide to help visualize the key differences between these SQL commands and functions, complete with sample input data, SQL scripts, and expected outputs.

1. RANK vs DENSE_RANK vs ROW_NUMBER

- RANK(): Leaves gaps after ties (1,1,3). This function gives tied rows the same rank and then skips subsequent numbers.
- DENSE_RANK(): No gaps (1,1,2). This function gives tied rows the same rank without skipping numbers.

- ROW_NUMBER assigns a unique, sequential number to each row.

👉 Use DENSE_RANK for leaderboards where gaps don't make sense!

Sample Table: players

```
| name  | score |
|-------|-------|
| Alice | 95    |
| Bob   | 95    |
| Carol | 90    |
```

SQL Query to Compare All Three Functions:

```sql
SELECT
    name,
    score,
    RANK () OVER (ORDER BY score DESC) AS rank,
    DENSE_RANK () OVER (ORDER BY score DESC) AS dense_rank,
    ROW_NUMBER () OVER (ORDER BY score DESC) AS row_num
FROM players;
```

Expected Output Explanation:

RANK:

- Assigns the same rank to tied values but leaves gaps afterward.

Result:

```
    | name  | score | rank |
    |-------|-------|------|
    | Alice | 95    | 1    |
    | Bob   | 95    | 1    |
    | Carol | 90    | 3    |
```

DENSE_RANK:

- Assigns the same rank to ties and does not leave gaps.

Result:

```
| name  | score | dense_rank |
|-------|-------|------------|
| Alice | 95    | 1          |
| Bob   | 95    | 1          |
| Carol | 90    | 2          |
```

ROW_NUMBER:

- Provides a unique sequential number to each row regardless of ties.

Result:

```
| name  | score | row_num |
|-------|-------|---------|
| Alice | 95    | 1       |
| Bob   | 95    | 2       |
| Carol | 90    | 3       |
```

2. HAVING vs WHERE Clause

- WHERE: Filters rows (before grouping).
- HAVING: Filters groups (after GROUP BY).

👉 "Show departments WHERE salary > 5000" vs "Show departments HAVING AVG(salary) > 5000."

Sample Table: employees

```
| department  | salary|
|------------|-------|
```

```
| HR         | 60000 |
| IT         | 55000 |
| HR         | 50000 |
| IT         | 45000 |
| IT         | 70000 |
| Sales      | 40000 |
| Sales      | 45000 |
```

WHERE Clause (filters rows before GROUP BY):

```sql
SELECT department, COUNT() AS count
FROM employees
WHERE salary > 50000
GROUP BY department;
```

Expected Output:

```
| department | count |
|------------|-------|
| HR         | 1     |
| IT         | 2     |
```

HAVING Clause (filters groups after GROUP BY):

```sql
SELECT department, COUNT () AS emp_count
FROM employees
GROUP BY department
HAVING COUNT () > 1;
```

Expected Output:

```
| department | emp_count |
|------------|-----------|
| HR         | 2         |
```

```
| IT          | 3          |
| Sales       | 2          |
```

3. UNION vs UNION ALL

- UNION: "Merge and deduplicate."
- UNION ALL: "Merge and keep duplicates (way faster)."

👉 Use UNION ALL unless you explicitly need uniqueness.

Sample Data:

```
Table1 (names):
| name     |
|---------|
| Alice    |
| Bob      |
| Charlie |
```

Table2 (names):

```
| name   |
|-------|
| Alice |
| David |
```

UNION (removes duplicates):

```sql
SELECT name FROM Table1
UNION
SELECT name FROM Table2;
```

Expected Output:

```
| name    |
|---------|
| Alice   |
| Bob     |
| Charlie |
| David   |

```

UNION ALL (includes duplicates):

```
SELECT name FROM Table1
UNION ALL
SELECT name FROM Table2;

```

Expected Output:

```
| name    |
|---------|
| Alice   |
| Bob     |
| Charlie |
| Alice   |
| David   |

```

4. JOIN vs UNION

JOIN (combines columns from related rows):

Sample Tables: employees

```
| emp_id | name  |
|--------|-------|
| 1      | Alice |
```

```
| 2        | Bob    |
```

salaries

```
| emp_id | salary|
|--------|-------|
| 1        | 50000 |
| 2        | 60000 |
| 3        | 70000 |
```

INNER JOIN Example:

```
SELECT a.emp_id, a.name, b.salary
FROM employees a
INNER JOIN salaries b ON a.emp_id = b.emp_id;
```

Expected Output:

```
| emp_id | name   | salary|
|--------|-------|-------|
| 1        | Alice | 50000 |
| 2        | Bob    | 60000 |
```

UNION (stacks similar rows):

Sample Queries:

```
SELECT name, 'employee' AS type FROM employees
UNION
SELECT name, 'manager' FROM (SELECT 2 AS emp_id, 'Bob' AS name UNION SELECT 3, 'Carol');
```

Expected Output:

```
| name  | type     |
|-------|----------|
| Alice | employee |
| Bob   | employee |
| Bob   | manager  |
| Carol | manager  |
```

5. DELETE vs DROP vs TRUNCATE

- DELETE: "I'll remove these specific rows (and log every change)."
- TRUNCATE: "I'll wipe ALL rows (and reset the counter)."
- DROP: "I'll nuke the entire table (RIP)."

👉 Need speed? TRUNCATE. Need precision? DELETE.

Sample Table: orders

```
| order_id | order_date   | amount |
|----------|--------------|--------|
| 1        | 2020-12-31   | 100    |
| 2        | 2021-01-15   | 150    |
| 3        | 2021-02-20   | 200    |
```

DELETE (removes specific rows):

```
DELETE FROM orders
WHERE order_date < '2021-01-01';
```

Expected Orders Table After DELETE:

```
| order_id | order_date   | amount |
|----------|--------------|--------|
| 2        | 2021-01-15   | 150    |
| 3        | 2021-02-20   | 200    |
```

DROP (removes the table entirely):

```sql
DROP TABLE orders;
```

TRUNCATE (removes all rows, retains structure):

```sql
TRUNCATE TABLE orders;
```

Expected Orders Table After TRUNCATE:

```
| order_id | order_date | amount |
|----------|------------|--------|
| (empty) |
```

6. CTE vs TEMP TABLE

CTE (temporary result set within a single query):

- CTE: Disposable, single-query use.
- Temp Table: Reusable, session-persistent.

👉 CTEs for readability, Temp Tables for complex workflows.

Sample Table: employees

```
| emp_id | name   | salary |
|--------|-------|--------|
| 1      | Alice | 120000 |
| 2      | Bob   | 90000  |
| 3      | Carol | 110000 |

```

CTE Example:

```sql
WITH HighEarners AS (
  SELECT  FROM employees WHERE salary > 100000
)
SELECT  FROM HighEarners;

```

Expected Output:

```
| emp_id | name   | salary |
|--------|-------|--------|
| 1      | Alice | 120000 |
| 3      | Carol | 110000 |

```

TEMP TABLE (persists for session):

```sql
CREATE TEMPORARY TABLE TempHighEarners AS
SELECT  FROM employees WHERE salary > 100000;

SELECT  FROM TempHighEarners;

```

Expected Output:

Same as the CTE output above.

7. SUBQUERIES vs CTE

Using SUBQUERIES:

Sample Tables: employees

```
| emp_id | name  | dept_id |
|--------|-------|---------|
| 1      | Alice | 10      |
| 2      | Bob   | 20      |
| 3      | Carol | 10      |
```

departments

```
| dept_id | location |
|---------|----------|
| 10      | NY       |
| 20      | LA       |
```

Subquery Example:

```sql
SELECT name FROM employees
WHERE dept_id IN (SELECT dept_id FROM departments WHERE location = 'NY');
```

Expected Output:

```
| name  |
|-------|
| Alice |
| Carol |
```

CTE Example:

```
WITH NYDepts AS (
  SELECT dept_id FROM departments WHERE location = 'NY'
)
SELECT name FROM employees WHERE dept_id IN (SELECT dept_id FROM NYDepts);

```

Expected Output:

Same as the subquery output.

8. ISNULL vs COALESCE

Sample Table: contacts

```
| contact_id | phone    | mobile   |
|------------|---------|---------|
| 1          | NULL     | 123456   |
| 2          | 987654   | NULL     |
| 3          | NULL     | NULL     |

```

ISNULL (SQL Server specific):

```
SELECT contact_id, ISNULL(phone, 'N/A') AS phone
FROM contacts;

```

Expected Output:

```
| contact_id | phone   |
|------------|--------|
```

```
| 1          | N/A    |
| 2          | 987654 |
| 3          | N/A    |

```

COALESCE (standard SQL):

```sql
SELECT contact_id, COALESCE(phone, mobile, 'N/A') AS contact_number
FROM contacts;

```

Expected Output:

```
| contact_id | contact_number |
|------------|----------------|
| 1          | 123456         |
| 2          | 987654         |
| 3          | N/A            |

```

9. INTERSECT vs INNER JOIN

Sample Tables for INTERSECT:

Table1

```
| id |
|----|
| 1  |
| 2  |
```

```
| 3  |
```

Table2

```
| id |
|----|
| 2  |
| 3  |
| 4  |
```

INTERSECT Example:

```
SELECT id FROM Table1
INTERSECT
SELECT id FROM Table2;
```

Expected Output:

```
| id |
|----|
| 2  |
| 3  |
```

Sample Tables for INNER JOIN:

Table1

```
| id | name |
|----|------|
| 1  | A    |
| 2  | B    |
| 3  | C    |
```

Table2

```
| id | value |
|----|-------|
| 2  | X     |
| 3  | Y     |
| 4  | Z     |
```

INNER JOIN Example:

```sql
SELECT a.id, a.name, b.value
FROM Table1 a
INNER JOIN Table2 b ON a.id = b.id;
```

Expected Output:

```
| id | name | value |
|----|------|-------|
| 2  | B    | X     |
| 3  | C    | Y     |
```

10. EXCEPT vs NOT IN

Sample Tables:

Table1

```
| id |
|----|
| 1  |
| 2  |
| 3  |
| 4  |
```

Table2

```
| id |
|----|
| 2  |
| 4  |
```

EXCEPT Example:

```sql
SELECT id FROM Table1
EXCEPT
SELECT id FROM Table2;
```

Expected Output:

```
| id |
|----|
| 1  |
| 3  |
```

NOT IN Example:

```sql
SELECT id FROM Table1
WHERE id NOT IN (SELECT id FROM Table2);

```

Expected Output:

```
| id |
|----|
| 1  |
| 3  |

```

11. INNER JOIN vs LEFT JOIN vs RIGHT JOIN vs FULL JOIN

Definitions:

- INNER JOIN: Returns rows when there is a match between the tables.
- LEFT JOIN: Returns all rows from the left table and matching rows from the right table. Unmatched rows in the right table return NULL.
- RIGHT JOIN: Returns all rows from the right table and matching rows from the left table. Unmatched rows in the left table return NULL.
- FULL JOIN: Returns rows when there is a match in either the left or the right table, filling unmatched rows with NULL.

Sample Tables:

Customers Table

```
| CustomerID | Name    | City   |
|------------|---------|--------|
| 1          | Alice   | Paris  |
| 2          | Bob     | Tokyo  |
| 3          | Charlie| Delhi  |
```

Orders Table:

```
| OrderID | CustomerID | Product |
|---------|------------|---------|
| 101     | 1          | Laptop  |
| 102     | 2          | Phone   |
| 103     | 4          | Camera  |
```

SQL Queries and Outputs:

INNER JOIN

```sql
SELECT Customers.Name, Orders.Product
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Output:

```
| Name  | Product |
|-------|---------|
| Alice | Laptop  |
| Bob   | Phone   |
```

LEFT JOIN

```sql
SELECT Customers.Name, Orders.Product
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Output:

```
| Name    | Product |
|---------|---------|
| Alice   | Laptop  |
| Bob     | Phone   |
| Charlie | NULL    |
```

RIGHT JOIN

```sql
SELECT Customers.Name, Orders.Product
FROM Customers
RIGHT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Output:

```
| Name  | Product |
|-------|---------|
| Alice | Laptop  |
| Bob   | Phone   |
| NULL  | Camera  |
```

FULL JOIN

```sql
SELECT Customers.Name, Orders.Product
FROM Customers
FULL JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Output:

```
| Name    | Product |
|---------|---------|
| Alice   | Laptop  |
| Bob     | Phone   |
| Charlie | NULL    |
| NULL    | Camera  |
```

12. LAG() vs LEAD() Functions

Definitions:

- LAG(): Accesses data from the previous row in the result set.

- LEAD(): Accesses data from the next row in the result set.

Sample Table:

Sales Table:

```
| SaleID | Product | SaleAmount | SaleDate   |
|--------|---------|------------|------------|
| 1      | Laptop  | 1000       | 2023-01-01 |
| 2      | Phone   | 500        | 2023-01-02 |
| 3      | Tablet  | 700        | 2023-01-03 |
```

SQL Queries and Outputs:

LAG() Example

```sql
SELECT Product, SaleAmount, LAG(SaleAmount) OVER (ORDER BY
SaleDate) AS PreviousSale
FROM Sales;
```

Output:

```
| Product | SaleAmount | PreviousSale |
|---------|------------|--------------|
| Laptop  | 1000       | NULL         |
| Phone   | 500        | 1000         |
| Tablet  | 700        | 500          |
```

LEAD() Example

```sql
SELECT Product, SaleAmount, LEAD(SaleAmount) OVER (ORDER BY
SaleDate) AS NextSale
FROM Sales;
```

Output:

```
| Product | SaleAmount | NextSale |
|---------|------------|----------|
| Laptop  | 1000       | 500      |
| Phone   | 500        | 700      |
| Tablet  | 700        | NULL     |
```

That's a great follow-up! Even though LAG() and LEAD() may return NULL in some cases (especially at the boundaries of data), they are still incredibly valuable for many use cases.

Why Use LAG() and LEAD()?

1. Comparing Sequential Rows:

In datasets where order matters (like time-series data), you often need to compare the current row with a previous or next row to analyze trends.

E.g., "Has the sale increased or decreased compared to the previous day?"

2. Trend Analysis:

They are essential for calculating moving averages, growth rates, or detecting anomalies.

3. Handling Gaps in Data:

Even when NULL is returned, it's a helpful indicator that there is no relevant value available (like missing events or boundary conditions).

4. Flexible Calculation:

You can handle NULL gracefully using functions like COALESCE() to replace them with default values (like 0 or the current row's value).

Practical Demonstration

Sales Data Example

```
| SaleDate   | Product | SaleAmount |
|-----------|---------|------------|
| 2023-01-01 | Laptop  | 1000       |
| 2023-01-02 | Phone   | 800        |
| 2023-01-03 | Tablet  | 1200       |
```

SQL Query

```sql
SELECT
    SaleDate, Product, SaleAmount,
    LAG(SaleAmount) OVER (ORDER BY SaleDate) AS PreviousSale,
    LEAD(SaleAmount) OVER (ORDER BY SaleDate) AS NextSale,
    SaleAmount - LAG(SaleAmount) OVER (ORDER BY SaleDate) AS
SaleChange
FROM Sales;
```

Result:

```
|SaleDate    |Product   |SaleAmount| PreviousSale| NextSale| SaleChange|
|------------|----------|----------|------------------
| 2023-01-01 | Laptop   | 1000     | NULL         | 800     | NULL       |
| 2023-01-02 | Phone    | 800      | 1000         | 1200    | -200       |
| 2023-01-03 | Tablet   | 1200     | 800          | NULL    | 400        |
```

Why the NULLs Are Useful

- In row 1, NULL for PreviousSale clearly shows it's the first row with no earlier reference.
- In row 3, NULL for NextSale indicates it's the last row, so there's no subsequent sale to compare.
- The SaleChange column provides meaningful insights for the rows that have valid comparisons.

Even though NULLs appear at boundaries, they are indicators of valid structural limits in your dataset. Handling them with logic (like COALESCE() or conditional checks) ensures these functions remain powerful for insights. Would you like suggestions on how to visualize or analyze this further?

What are moving averages, growth rates, or detecting anomalies?

1. Moving Averages

A moving average is a calculation that takes the average of a fixed number of recent data points. It's useful for smoothing out short-term fluctuations and identifying long-term trends.

Use Case: Smoothing daily sales data over a 3-day period

Sample Data:

```
| SaleDate   | SaleAmount |
|------------|------------|
| 2023-01-01 | 100        |
| 2023-01-02 | 200        |
| 2023-01-03 | 300        |
| 2023-01-04 | 400        |
| 2023-01-05 | 500        |
```

SQL Query for 3-Day Moving Average

```
SELECT
    SaleDate,
    SaleAmount,
    AVG(SaleAmount) OVER (ORDER BY SaleDate ROWS BETWEEN 2
PRECEDING AND CURRENT ROW) AS MovingAvg3Days
FROM Sales;
```

Result:

```
| SaleDate   | SaleAmount | MovingAvg3Days |
|------------|------------|----------------|
| 2023-01-01 | 100        | 100            |
| 2023-01-02 | 200        | 150            |
| 2023-01-03 | 300        | 200            |
| 2023-01-04 | 400        | 300            |
| 2023-01-05 | 500        | 400            |
```

Explanation: The average smooths out the sharp changes and shows a more stable trend.

2. Growth Rates

A growth rate measures the percentage change from one period to the next.

Use Case: Calculating month-over-month sales growth

Sample Data:

```
| Month      | SaleAmount |
|------------|------------|
| 2023-01    | 1000       |
| 2023-02    | 1500       |
| 2023-03    | 2000       |
```

SQL Query for Growth Rate

```
SELECT
    Month,
    SaleAmount,
    (SaleAmount - LAG(SaleAmount) OVER (ORDER BY Month)) * 100.0
/ LAG(SaleAmount) OVER (ORDER BY Month) AS GrowthRate
FROM Sales;
```

Result:

```
| Month        | SaleAmount | GrowthRate |
|------------|------------|------------|
| 2023-01     | 1000        | NULL        |
| 2023-02     | 1500        | 50.00       |
| 2023-03     | 2000        | 33.33       |
```

Explanation: In February, sales grew by 50% compared to January, and in March, they grew by 33.33% compared to February.

3. Detecting Anomalies

An anomaly is an unusual or unexpected data point that differs significantly from the rest.

Use Case: Detecting days when sales are significantly higher than the moving average

Sample Data:

```
| SaleDate    | SaleAmount |
|------------|------------|
| 2023-01-01 | 100         |
| 2023-01-02 | 200         |
| 2023-01-03 | 3000        |  -- Anomaly
| 2023-01-04 | 400         |
| 2023-01-05 | 500         |
```

SQL Query to Detect Anomalies

```
SELECT
    SaleDate,
    SaleAmount,
    AVG(SaleAmount) OVER (ORDER BY SaleDate ROWS BETWEEN 2
PRECEDING AND CURRENT ROW) AS MovingAvg3Days,
    CASE
        WHEN SaleAmount > 2 * AVG(SaleAmount) OVER (ORDER BY
SaleDate ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        THEN 'Anomaly'
        ELSE 'Normal'
    END AS AnomalyStatus
FROM Sales;
```

Result:

```
| SaleDate   | SaleAmount | MovingAvg3Days | AnomalyStatus |
|------------|------------|----------------|---------------|
| 2023-01-01 | 100        | 100            | Normal        |
| 2023-01-02 | 200        | 150            | Normal        |
| 2023-01-03 | 3000       | 1100           | Anomaly       |
| 2023-01-04 | 400        | 1200           | Normal        |
| 2023-01-05 | 500        | 1400           | Normal        |
```

Explanation: The sales on `2023-01-03` are much higher than the moving average, flagging it as an anomaly.

Summary

- Moving Averages: Smooth out fluctuations for better trend analysis.
- Growth Rates: Show percentage changes between periods for performance tracking.
- Anomaly Detection: Helps identify outliers for further investigation.
- LAG() vs LEAD(): Work within a single table to provide relative row data for analytical queries.
- **LAG()** looks backward (to the previous row) from the current row. Therefore, for the first row, there's no previous row, which results in NULL.
- **LEAD()** looks forward (to the next row) from the current row. For the last row, there is no next row, which results in NULL.
- The reason **LAG()** and **LEAD()** can return NULL is due to their access pattern in the result set.

This guide, complete with sample input and output data, should help novice users clearly understand the differences between these SQL commands and functions. Enjoy exploring and practicing these concepts!