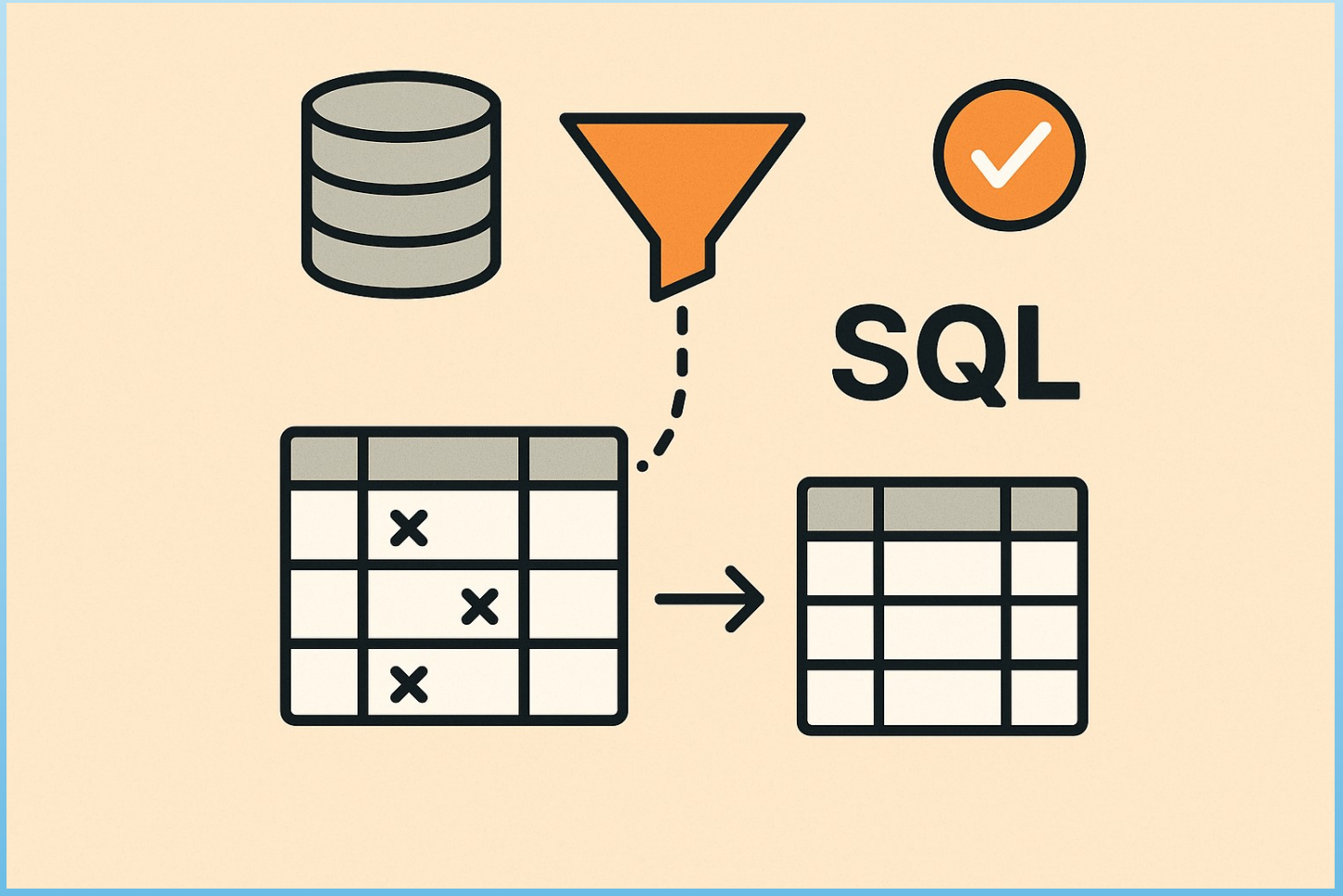
How to Clean Data Using SQL



Vishnu Vardhan

How to Clean Data Using SQL

Introduction

Data cleaning is a critical step in any data analysis or data science project. Without proper data cleaning, your analysis may lead to inaccurate or misleading results. This enhanced guide covers essential SQL data cleaning techniques with practical examples, step-by-step strategies, and real-world input/output demonstrations.

1. Handling Missing Values

Problem: Missing values can lead to inaccurate analysis or cause errors during joins and aggregations.

Solution: Use `COALESCE()` or `IFNULL()` to replace missing values with defaults.

Example with Data:

```
-- Input Data (users table)
/*
| user_id | email               |
|---------|---------------------|
| 1       | john@example.com    |
| 2       | NULL                |
| 3       | sarah@example.com   |
| 4       | NULL                |
| 5       | mike@example.com    |
*/

SELECT user_id, COALESCE(email, 'unknown') AS cleaned_email
FROM users;
```

Output:

```
/*
| user_id | cleaned_email       |
|---------|---------------------|
| 1       | john@example.com    |
| 2       | unknown             |
| 3       | sarah@example.com   |
| 4       | unknown             |
| 5       | mike@example.com    |

*/
```

2. Removing Duplicates

Problem: Duplicates in data can distort results and lead to incorrect conclusions.

Solution: Use `ROW_NUMBER()` to eliminate duplicate rows.

Example with Data:

```
-- Input Data (orders table)
/*
| order_id | user_id | created_at          | amount |
|----------|---------|---------------------|--------|
| 101      | 1       | 2023-01-01 10:00:00 | 100    |
| 102      | 1       | 2023-01-02 11:00:00 | 150    |
| 103      | 2       | 2023-01-01 09:00:00 | 200    |
| 104      | 3       | 2023-01-03 12:00:00 | 120    |
| 105      | 3       | 2023-01-03 13:00:00 | 130    |
*/

WITH RankedRows AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY
created_at DESC) AS row_num
    FROM orders
)
SELECT order_id, user_id, created_at, amount
FROM RankedRows
WHERE row_num = 1;
```

Output:

```
/*

| order_id | user_id | created_at          | amount |
|----------|---------|---------------------|--------|
| 102      | 1       | 2023-01-02 11:00:00 | 150    |
| 103      | 2       | 2023-01-01 09:00:00 | 200    |
| 105      | 3       | 2023-01-03 13:00:00 | 130    |

*/
```

3. Standardizing Data Formats

Problem: Inconsistent data formats can cause issues in comparisons or analysis.

Solution: Use `LOWER()`, `UPPER()`, and `TRIM()` to standardize text.

Example with Data:

```
-- Input Data (customers table)
/*
| customer_id | first_name |
|-------------|------------|
| 1           | JOHN       |
| 2           | Mary       |
| 3           | peter      |
| 4           | " alice "  |
| 5           | BOB        |
*/

SELECT
    customer_id,
    TRIM(LOWER(first_name)) AS standardized_name
FROM customers;
```

Output:

```
/*

| customer_id | standardized_name |
|-------------|-------------------|
| 1           | john              |
| 2           | mary              |
| 3           | peter             |
| 4           | alice             |
| 5           | bob               |

*/
```

4. Handling Outliers

Problem: Outliers can distort analysis results.

Solution: Identify and either remove or cap outliers.

Example with Data:

```
-- Input Data (orders table)
/*
| order_id | amount |
|----------|--------|
| 101      | 100    |
| 102      | 150    |
| 103      | 200    |
| 104      | 1200   |  -- Outlier
| 105      | 130    |
*/

-- Identifying outliers
SELECT order_id, amount
FROM orders
WHERE amount > (SELECT AVG(amount) + 3 * STDDEV(amount) FROM orders);

-- Capping outliers
UPDATE orders
SET amount = (SELECT AVG(amount) + 3 * STDDEV(amount) FROM orders)
```

```sql
WHERE amount > (SELECT AVG(amount) + 3 * STDDEV(amount) FROM orders);

SELECT * FROM orders;
```

Output (after capping):

```
/*

| order_id | amount |
|----------|--------|
| 101      | 100    |
| 102      | 150    |
| 103      | 200    |
| 104      | 356    |  -- Capped value
| 105      | 130    |

*/
```

5. Date Format Standardization

Problem: Inconsistent date formats can cause issues in time-based analysis.

Solution: Use `TO_DATE()` or `EXTRACT()` functions.

Example with Data:

```sql
-- Input Data (orders table)
/*
| order_id | order_date   |
|----------|--------------|
| 101      | 01-01-2023   |
| 102      | 2023/02/15   |
| 103      | March 3 2023 |
| 104      | 04-04-2023   |
| 105      | 2023-05-05   |
*/

-- Standardizing dates
```

```sql
SELECT
    order_id,
    TO_DATE(order_date, 'YYYY-MM-DD') AS standardized_date
FROM orders;

-- Extracting components
SELECT
    order_id,
    EXTRACT(YEAR FROM TO_DATE(order_date, 'YYYY-MM-DD')) AS year,
    EXTRACT(MONTH FROM TO_DATE(order_date, 'YYYY-MM-DD')) AS month
FROM orders;
```

Output:

```
/*

| order_id | standardized_date | year | month |
|----------|-------------------|------|-------|
| 101      | 2023-01-01        | 2023 | 1     |
| 102      | 2023-02-15        | 2023 | 2     |
| 103      | 2023-03-03        | 2023 | 3     |
| 104      | 2023-04-04        | 2023 | 4     |
| 105      | 2023-05-05        | 2023 | 5     |

*/
```

6. Correcting Data Entry Errors

Problem: Manual data entry often leads to formatting errors.

Solution: Use `REGEXP` to detect and correct errors.

Example with Data:

```sql
-- Input Data (customers table)
/*
| customer_id | phone_number  |
|-------------|---------------|
| 1           | 1234567890    |
```

```
| 2           | 234-567-8901  |
| 3           | 34567890      |
| 4           | (456)7890123  |
| 5           | 5678901234    |  -- Contains letter O
*/

-- Finding invalid phone numbers
SELECT customer_id, phone_number
FROM customers
WHERE phone_number NOT REGEXP '^[0-9]{10}$';
```

Output:

```
/*

| customer_id | phone_number  |
|-------------|---------------|
| 2           | 234-567-8901  |
| 3           | 34567890      |
| 4           | (456)7890123  |
| 5           | 5678901234    |

*/
```

7. Handling Null Values in Aggregations

Problem: Null values in aggregations can cause incorrect results.

Solution: Use `COALESCE()` to handle nulls.

Example with Data:

```
-- Input Data (orders table)
/*
| order_id | amount |
|----------|--------|
| 101      | 100    |
| 102      | NULL   |
| 103      | 200    |
```

```
| 104      | NULL   |
| 105      | 150    |
*/

SELECT SUM(COALESCE(amount, 0)) AS total_amount FROM orders;
```

Output:

```
/*

| total_amount |
|--------------|
| 450          |

*/
```

8. Removing Leading/Trailing Spaces

Problem: Extra spaces can cause comparison issues.

Solution: Use `TRIM()` to remove unnecessary whitespace.

Example with Data:

```
-- Input Data (employees table)
/*
| emp_id | first_name  |
|--------|-------------|
| 1      | " John "    |
| 2      | "  Mary  "  |
| 3      | "Peter "    |
| 4      | " Alice"    |
| 5      | "Bob  "     |
*/

SELECT emp_id, TRIM(first_name) AS trimmed_name FROM employees;
```

Output:

```
/*

| emp_id | trimmed_name |
|--------|--------------|
| 1      | John         |
| 2      | Mary         |
| 3      | Peter        |
| 4      | Alice        |
| 5      | Bob          |

*/
```

9. Splitting Combined Columns into Multiple Columns

Problem: Data often comes combined in a single column (e.g., full names, addresses) and needs to be split for analysis.

Solution: Use `SUBSTRING()`, `SPLIT_PART()`, or similar functions to separate values.

Example with Data:

```
-- Input Data (customers table)
/*
| customer_id | full_name       |
|-------------|-----------------|
| 1           | John Smith      |
| 2           | Mary Johnson    |
| 3           | Peter Parker    |
| 4           | Alice Williams  |
| 5           | Bob Brown       |
*/

SELECT
    customer_id,
    SUBSTRING(full_name, 1, POSITION(' ' IN full_name) - 1) AS
first_name,
    SUBSTRING(full_name, POSITION(' ' IN full_name) + 1) AS last_name
FROM customers;
```

Output:

```
/*

| customer_id | first_name | last_name |
|-------------|------------|-----------|
| 1           | John       | Smith     |
| 2           | Mary       | Johnson   |
| 3           | Peter      | Parker    |
| 4           | Alice      | Williams  |
| 5           | Bob        | Brown     |

*/
```

10. Handling Inconsistent Categorical Values

Problem: Categorical data (e.g., product categories) may have inconsistent labels (e.g., "Electronics" vs. "ELECTRONICS").

Solution: Standardize categories using `CASE` statements or `UPDATE` queries.

Example with Data:

```
-- Input Data (products table)
/*
| product_id | category        |
|------------|-----------------|
| 1          | Electronics     |
| 2          | ELECTRONICS     |
| 3          | books           |
| 4          | Books           |
| 5          | stationery      |
*/

SELECT
    product_id,
    CASE
        WHEN LOWER(category) LIKE '%electronic%' THEN 'Electronics'
```

```sql
        WHEN LOWER(category) LIKE '%book%' THEN 'Books'
        WHEN LOWER(category) LIKE '%stationery%' THEN 'Stationery'
        ELSE category
    END AS standardized_category
FROM products;
```

Output:

```
/*

| product_id | standardized_category |
|------------|-----------------------|
| 1          | Electronics           |
| 2          | Electronics           |
| 3          | Books                 |
| 4          | Books                 |
| 5          | Stationery            |

*/
```

Conclusion

This guide provides practical, real-world examples of data cleaning techniques in SQL. Each concept is demonstrated with sample input data and the corresponding output after cleaning, making the techniques more tangible and easier to understand. By following these methods, you can ensure your data is clean, consistent, and ready for analysis.