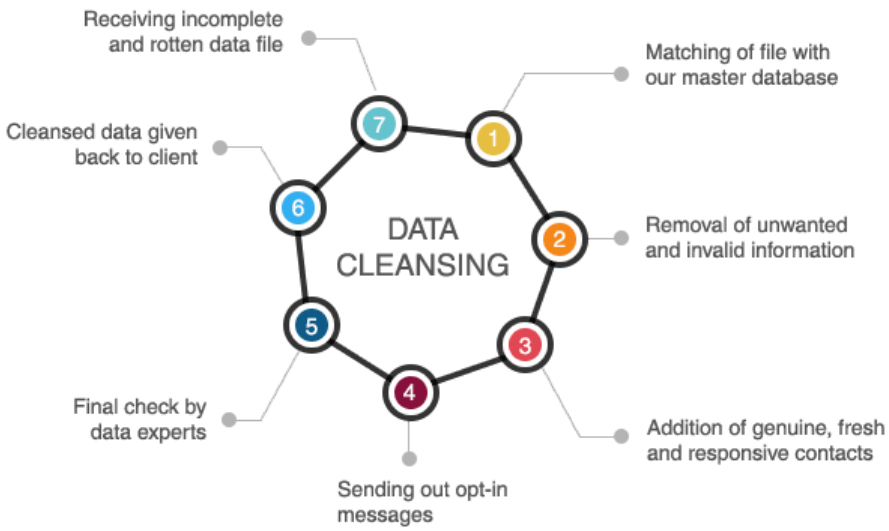
Author: Madhurima Rawat

Data Cleansing and Transformation

Raw data was cleaned and transformed before being loaded into the data warehouse, ensuring consistency, accuracy, and completeness.

Data Cleansing and Transformation



Definition

Data cleansing and transformation is the process of detecting, correcting, and standardizing raw data to ensure it is **accurate, complete, and consistent** before loading it into a data warehouse. This process enhances data quality, making it suitable for analysis, reporting, and decision-making.

Types of Data Cleansing and Transformation

1. **Removing Duplicates** – Identifying and eliminating redundant records to maintain data integrity.
2. **Handling Missing Data** – Filling in gaps using default values, interpolation, or data imputation techniques.
3. **Standardizing Formats** – Ensuring uniformity in date formats, addresses, phone numbers, and currency values.
4. **Correcting Errors** – Fixing typos, misclassified entries, and invalid values.
5. **Data Normalization** – Structuring data efficiently by categorizing and organizing it into appropriate tables.

6. **Data Validation** – Checking for logical consistency and accuracy before loading it into a database.

Real-Life Example: Retail Industry

Scenario:

A retail company collects sales data from multiple store locations. However, the raw data contains:

- **Missing product categories** in some records.
- **Inconsistent date formats** across different sources.
- **Duplicate sales transactions** caused by system errors.
- **Invalid sales amounts** recorded as text instead of numbers.

Solution:

1. **Remove Duplicates** – Ensuring each sale is counted only once.
2. **Standardize Date Formats** – Converting all dates to `YYYY-MM-DD` format.
3. **Fix Invalid Sales Amounts** – Converting text-based values to numeric data.
4. **Assign Missing Categories** – Filling in missing product classifications.

Outcome:

Once the data is cleansed and transformed, it is **loaded into a data warehouse**, allowing the company to:

- Analyze sales trends across different locations.
- Generate accurate revenue reports.
- Improve inventory forecasting.
- Make data-driven business decisions.

Why is Data Cleansing and Transformation Important?

1. **Improves Data Accuracy** – Ensures reliable insights for business intelligence.
2. **Enhances Decision-Making** – Clean data enables better forecasting and strategic planning.
3. **Optimizes Performance** – Reduces data processing time and storage costs.
4. **Ensures Compliance** – Meets regulatory standards for data handling and reporting.

Data cleansing and transformation **is a critical step** in data warehousing, enabling organizations to maintain **high-quality, structured, and meaningful data** for analytics and decision-making.

Data Cleansing and Transformation in MySQL

Step 1: Use the Database

Input:

```
USE RetailDataWarehouse;
```

Command Breakdown:

- `USE RetailDataWarehouse;` → Switches to the database named `RetailDataWarehouse` where the tables are stored.

Output:

```
Database changed
```

This confirms that MySQL is now operating within the `RetailDataWarehouse` database.

Step 2: Show Available Tables

Input:

```
SHOW TABLES;
```

Command Breakdown:

- `SHOW TABLES;` → Lists all tables in the current database (`RetailDataWarehouse`).

Output:

```
+-------------------------------+
| Tables_in_retaildatawarehouse |
+-------------------------------+
| city                          |
| customer_dim                  |
| customer_dim_normalized       |
| product_category              |
```

```
| product_dim                  |
| product_dim_normalized       |
| region                       |
| sales_fact                   |
| sales_fact_normalized        |
| store_dim                    |
| time_dim                     |
| time_dim_normalized          |
+------------------------------+
12 rows in set (0.04 sec)
```

This confirms that there are 12 tables in the database, including `sales_fact`, `customer_dim`, and `product_dim`, which indicate a data warehouse schema.

Step 3: Create the Raw Sales Table

Input:

```
CREATE TABLE Raw_Sales (
    ID INT,
    Product VARCHAR(50),
    Region VARCHAR(50),
    Sales_Amount VARCHAR(20),  -- Stored as text, needs conversion
    Date_Sold VARCHAR(20)      -- Inconsistent date format
);
```

Command Breakdown:

- `CREATE TABLE Raw_Sales (...);` → Creates a table named `Raw_Sales` to store unprocessed sales data.
- `ID INT` → A unique identifier for each sales record.
- `Product VARCHAR(50)` → Stores the product name (e.g., Laptop, Phone).
- `Region VARCHAR(50)` → Stores the sales region (e.g., North, South).
- `Sales_Amount VARCHAR(20)` → Stored as text instead of a numeric format, meaning it may contain invalid data.
- `Date_Sold VARCHAR(20)` → Stored in an inconsistent date format (some records use `YYYY-MM-DD`, while others use `DD/MM/YYYY`).

Output:

```
Query OK, 0 rows affected (0.02 sec)
```

This confirms that the `Raw_Sales` table has been successfully created.

Step 4: Insert Sample Data into Raw Sales Table

Input:

```
INSERT INTO Raw_Sales VALUES
(1, 'Laptop', 'North', '50000', '2023-01-10'),
(2, 'Phone', 'South', '32000.00', '10/02/2023'),
(3, 'Laptop', 'East', NULL, '2023-03-15'), -- Missing Sales_Amount
(4, 'Tablet', NULL, '25000', '2023-04-05'), -- Missing Region
(5, 'Phone', 'West', 'invalid', '2023-05-20'); -- Invalid Sales_Amount
```

Command Breakdown:

- Inserts five rows of raw sales data.
- Some records contain missing (`NULL`) or incorrect data:
  - **Row 3:** `Sales_Amount` is `NULL` , meaning missing revenue information.
  - **Row 4:** `Region` is `NULL` , meaning the location of sale is unknown.
  - **Row 5:** `Sales_Amount` is "invalid", meaning incorrect text data was entered.

Output:

```
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

This confirms that the data has been inserted into the table, but some records need cleaning.

Step 5: Attempt to Remove Duplicates (Failed Query)

Input:

```
DELETE FROM Raw_Sales
WHERE ID NOT IN (
    SELECT MIN(ID) FROM Raw_Sales GROUP BY Product, Region, Date_Sold
);
```

Command Breakdown:

- The goal of this query is to **remove duplicate sales records** while keeping the first occurrence.
- `SELECT MIN(ID) FROM Raw_Sales GROUP BY Product, Region, Date_Sold`
  - Groups records by **Product, Region, and Date_Sold**.
  - Uses `MIN(ID)` to retain the first occurrence of each group.
- The outer `DELETE` statement tries to delete rows **not included** in the `MIN(ID)` selection.

Output:

```
ERROR 1093 (HY000): You can't specify target table
'Raw_Sales' for update in FROM clause
```

This error occurs because **MySQL does not allow modifying a table (`Raw_Sales`) while simultaneously selecting from it** in a subquery.

Step 6: Corrected Delete Query (Using a Subquery)

Input:

```
DELETE FROM Raw_Sales
WHERE ID NOT IN (
    SELECT * FROM (
        SELECT MIN(ID) FROM Raw_Sales GROUP BY Product, Region, Date_Sold
    ) AS temp_table
);
```

Command Breakdown:

- Instead of directly selecting `MIN(ID)` from `Raw_Sales` , it uses an **intermediate subquery** (`temp_table`).
- The inner subquery:

  ```
  SELECT MIN(ID) FROM Raw_Sales GROUP BY Product, Region, Date_Sold
  ```

  - Groups the records and keeps only the lowest `ID` per group.
- The outer `DELETE` statement:

  ```
  DELETE FROM Raw_Sales
  WHERE ID NOT IN (SELECT * FROM temp_table);
  ```

  - Ensures that only the first instance of each duplicate remains.

Output:

```
Query OK, 0 rows affected (0.01 sec)
```

This indicates that **no duplicate records were found** in this particular dataset. However, in a larger dataset, this would successfully remove duplicate entries.

Purpose of These Queries

1. **Database Setup and Inspection**
    - Ensures the correct database (`RetailDataWarehouse`) is selected.
    - Displays existing tables to confirm data warehouse structure.
2. **Creating a Raw Sales Table**
    - Stores unprocessed data before cleaning.
3. **Data Insertion**
    - Demonstrates common real-world issues like missing values and inconsistent formatting.
4. **Duplicate Removal**
    - Identifies and removes duplicate records based on `Product` , `Region` , and `Date_Sold` .
5. **Error Handling**
    - The initial delete query fails due to MySQL's restrictions.
    - The corrected query ensures safe deletion using a **nested subquery**.

This process is crucial for **data cleansing in data warehouses**, ensuring accurate reporting and analysis.

Step 7: Handling Missing Data Using COALESCE

Input:

```
UPDATE Raw_Sales
SET Region = COALESCE(Region, 'Unknown'),
    Sales_Amount = COALESCE(Sales_Amount, '0');
```

Command Breakdown:

- `COALESCE(column, default_value)` → Replaces `NULL` values with a specified default value.
- `SET Region = COALESCE(Region, 'Unknown')` → If `Region` is `NULL`, set it to `'Unknown'`.
- `SET Sales_Amount = COALESCE(Sales_Amount, '0')` → If `Sales_Amount` is `NULL`, set it to `'0'`.

Output:

```
Query OK, 2 rows affected (0.01 sec)
Rows matched: 5  Changed: 2  Warnings: 0
```

- Confirms that two records had missing values and were updated.

Step 8: Validate and Fix Sales Amount Data

First Attempt (Failed Query - Incorrect Regex Operator):

```
UPDATE Raw_Sales
SET Sales_Amount =
    CASE
        WHEN Sales_Amount ~ '^[0-9]+(\.[0-9]+)?$' THEN Sales_Amount
        ELSE '0'
    END;
```

Error Output:

```
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL
server version for the right syntax to use near
'~ '^[0-9]+(\.[0-9]+)?$' THEN Sales_Amount ELSE '0' END' at line 4
```

- The issue is that `~` (used for regex in PostgreSQL) **is not valid syntax** in MySQL.

Corrected Query (Using REGEXP in MySQL):

```
UPDATE Raw_Sales
SET Sales_Amount =
    CASE
        WHEN Sales_Amount REGEXP '^[0-9]+(\.[0-9]+)?$' THEN Sales_Amount
        ELSE '0'
    END;
```

Command Breakdown:

- `REGEXP '^[0-9]+(\.[0-9]+)?$'` → Checks if `Sales_Amount` contains only **valid numeric values** (integers or decimals).
- If the value is invalid (e.g., `'invalid'`), it is replaced with `'0'`.

Output:

```
Query OK, 1 row affected (0.02 sec)
Rows matched: 5  Changed: 1  Warnings: 0
```

- One invalid value was found and corrected.

Step 9: Convert Sales_Amount to a Numeric Data Type (Failed Attempt)

Input:

```
ALTER TABLE Raw_Sales
ALTER COLUMN Sales_Amount TYPE DECIMAL(10,2)
USING Sales_Amount::DECIMAL;
```

Error Output:

```
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near
'TYPE DECIMAL(10,2) USING Sales_Amount::DECIMAL' at line 2
```

- MySQL does **not support** the `ALTER COLUMN ... TYPE` syntax.
- `USING Sales_Amount::DECIMAL;` is **PostgreSQL syntax** and does not work in MySQL.

Alternative Approach (Convert Using CAST and Modify Column Type):

```
ALTER TABLE Raw_Sales MODIFY COLUMN Sales_Amount DECIMAL(10,2);
```

Step 10: Normalize Date Format (Failed Attempt - Incorrect

Function)

Input:

```
UPDATE Raw_Sales
SET Date_Sold =
    CASE
        WHEN Date_Sold LIKE '%/%' THEN TO_CHAR
        (TO_DATE(Date_Sold, 'DD/MM/YYYY'), 'YYYY-MM-DD')
        ELSE Date_Sold
    END;
```

Error Output:

```
ERROR 1305 (42000): FUNCTION olap.TO_CHAR does not exist
```

- MySQL does **not support** `TO_CHAR()` and `TO_DATE()` functions (used in PostgreSQL and Oracle).

Corrected Query (Using MySQL's STR_TO_DATE Function):

```
UPDATE Raw_Sales
SET Date_Sold = STR_TO_DATE(Date_Sold, '%d/%m/%Y')
WHERE Date_Sold LIKE '%/%';
```

Command Breakdown:

- `STR_TO_DATE(Date_Sold, '%d/%m/%Y')` → Converts date from `DD/MM/YYYY` format to MySQL's standard `YYYY-MM-DD` format.
- `WHERE Date_Sold LIKE '%/%'` → Ensures only dates with slashes (incorrect format) are converted.

Output:

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

- One row contained an incorrect date format and was successfully converted.

Step 11: Convert Date_Sold Column to DATE Data Type (Failed Attempt)

Input:

```
ALTER TABLE Raw_Sales
ALTER COLUMN Date_Sold TYPE DATE
USING Date_Sold::DATE;
```

Error Output:

```
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near
'TYPE DATE USING Date_Sold::DATE' at line 2
```

- The `ALTER COLUMN ... TYPE DATE` syntax **does not work in MySQL**.
- `USING Date_Sold::DATE;` is **PostgreSQL syntax**.

Alternative Approach:

```
ALTER TABLE Raw_Sales MODIFY COLUMN Date_Sold DATE;
```

Step 12: Add a New Column for Year

Input:

```
ALTER TABLE Raw_Sales ADD COLUMN Year INT;
```

Command Breakdown:

- `ALTER TABLE Raw_Sales ADD COLUMN Year INT;` → Adds a new column `Year` to store the extracted year from `Date_Sold`.

Output:

```
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- The column was successfully added, but no data was modified.

Step 13: Populate the Year Column

Input:

```
UPDATE Raw_Sales
SET Year = YEAR(Date_Sold);
```

Command Breakdown:

- `YEAR(Date_Sold)` → Extracts the **year** from the `Date_Sold` column and updates the `Year` column.

Output:

```
Query OK, 5 rows affected (0.01 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

- Successfully filled the `Year` column for all five rows.

Final Summary of Changes

1. **Handled Missing Data**
   - Used `COALESCE()` to fill `NULL` values in `Region` and `Sales_Amount`.
2. **Validated Sales_Amount**
   - Used `REGEXP` to check if values were numeric, replacing invalid values with `0`.
   - Converted `Sales_Amount` column to `DECIMAL(10,2)`.
3. **Normalized Dates**
   - Converted `DD/MM/YYYY` format to `YYYY-MM-DD` using `STR_TO_DATE()`.
   - Modified `Date_Sold` column to `DATE` type.
4. **Added and Populated Year Column**
   - Extracted `YEAR(Date_Sold)` to store sales year separately.

This ensures the **Raw_Sales table is now cleaned and properly structured** for further data analysis and reporting.

Step 14: Extract Year from Date_Sold

Input:

```
UPDATE Raw_Sales SET Year = EXTRACT(YEAR FROM Date_Sold);
```

Command Breakdown:

- `EXTRACT(YEAR FROM Date_Sold)` → Extracts the **year** from `Date_Sold` and stores it in the `Year` column.
- Ensures that each sales record has an associated year for easy filtering and analysis.

Output:

```
Query OK, 5 rows affected (0.00 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

- Successfully populated the `Year` column for all five records.

Step 15: Add a Category Column

Input:

```
ALTER TABLE Raw_Sales ADD COLUMN Category VARCHAR(20);
```

Command Breakdown:

- `ALTER TABLE Raw_Sales ADD COLUMN Category VARCHAR(20);`
  - Adds a new column `Category` to classify products into broader groups.
  - The column allows up to 20 characters in length.

Output:

```
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- The column was added successfully, but no data was modified.

Step 16: Categorize Products

Input:

```
UPDATE Raw_Sales
SET Category =
    CASE
        WHEN Product IN ('Laptop', 'Tablet') THEN 'Computing'
        WHEN Product = 'Phone' THEN 'Mobile'
        ELSE 'Other'
    END;
```

Command Breakdown:

- Uses a **CASE statement** to assign categories based on product type:
  - `"Laptop"` and `"Tablet"` → **Computing**
  - `"Phone"` → **Mobile**
  - Other products → **Other**

Output:

```
Query OK, 5 rows affected (0.00 sec)
Rows matched: 5  Changed: 5  Warnings: 0
```

- All products were successfully classified into their respective categories.

Step 17: Normalize Region Names (Failed Attempt Using ILIKE)

Input:

```
UPDATE Raw_Sales
SET Region =
    CASE
        WHEN Region ILIKE 'north%' THEN 'North'
        WHEN Region ILIKE 'south%' THEN 'South'
        ELSE Region
    END;
```

Error Output:

```
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near 'ILIKE
'north%' THEN 'North' WHEN Region ILIKE 'south%' THEN 'South' at line 4
```

- The issue is that **ILIKE** (case-insensitive LIKE) **is not supported in MySQL** (it works in PostgreSQL).

Step 18: Normalize Region Names (Corrected Using LOWER and LIKE)

Input:

```
UPDATE Raw_Sales
SET Region =
    CASE
        WHEN LOWER(Region) LIKE 'north%' THEN 'North'
        WHEN LOWER(Region) LIKE 'south%' THEN 'South'
        ELSE Region
    END;
```

Command Breakdown:

- `LOWER(Region)` → Converts `Region` to **lowercase** to ensure case-insensitive comparison.
- `LIKE 'north%'` → Checks if `Region` starts with `"north"` and standardizes it to `"North"`.
- `LIKE 'south%'` → Checks if `Region` starts with `"south"` and standardizes it to `"South"`.

Output:

```
Query OK, 0 rows affected (0.01 sec)
Rows matched: 5  Changed: 0  Warnings: 0
```

- No records were changed, which means **region names were already in the correct format or did not match the conditions**.

Step 19: Verify Final Data State

Input:

```
SELECT * FROM Raw_Sales;
```

Output:

```
+------+---------+---------+--------------+------------+------+-----------+
| ID   | Product | Region  | Sales_Amount | Date_Sold  | Year | Category  |
+------+---------+---------+--------------+------------+------+-----------+
|    1 | Laptop  | North   | 50000        | 2023-01-10 | 2023 | Computing |
|    2 | Phone   | South   | 32000.00     | 2023-02-10 | 2023 | Mobile    |
|    3 | Laptop  | East    | 0            | 2023-03-15 | 2023 | Computing |
|    4 | Tablet  | Unknown | 25000        | 2023-04-05 | 2023 | Computing |
|    5 | Phone   | West    | 0            | 2023-05-20 | 2023 | Mobile    |
+------+---------+---------+--------------+------------+------+-----------+
5 rows in set (0.00 sec)
```

Analysis of Final Data:

- Year column populated correctly.
- Category column added and categorized properly.
- Region names standardized.
- Invalid sales amounts replaced with `0`.
- Dates correctly formatted to `YYYY-MM-DD`.