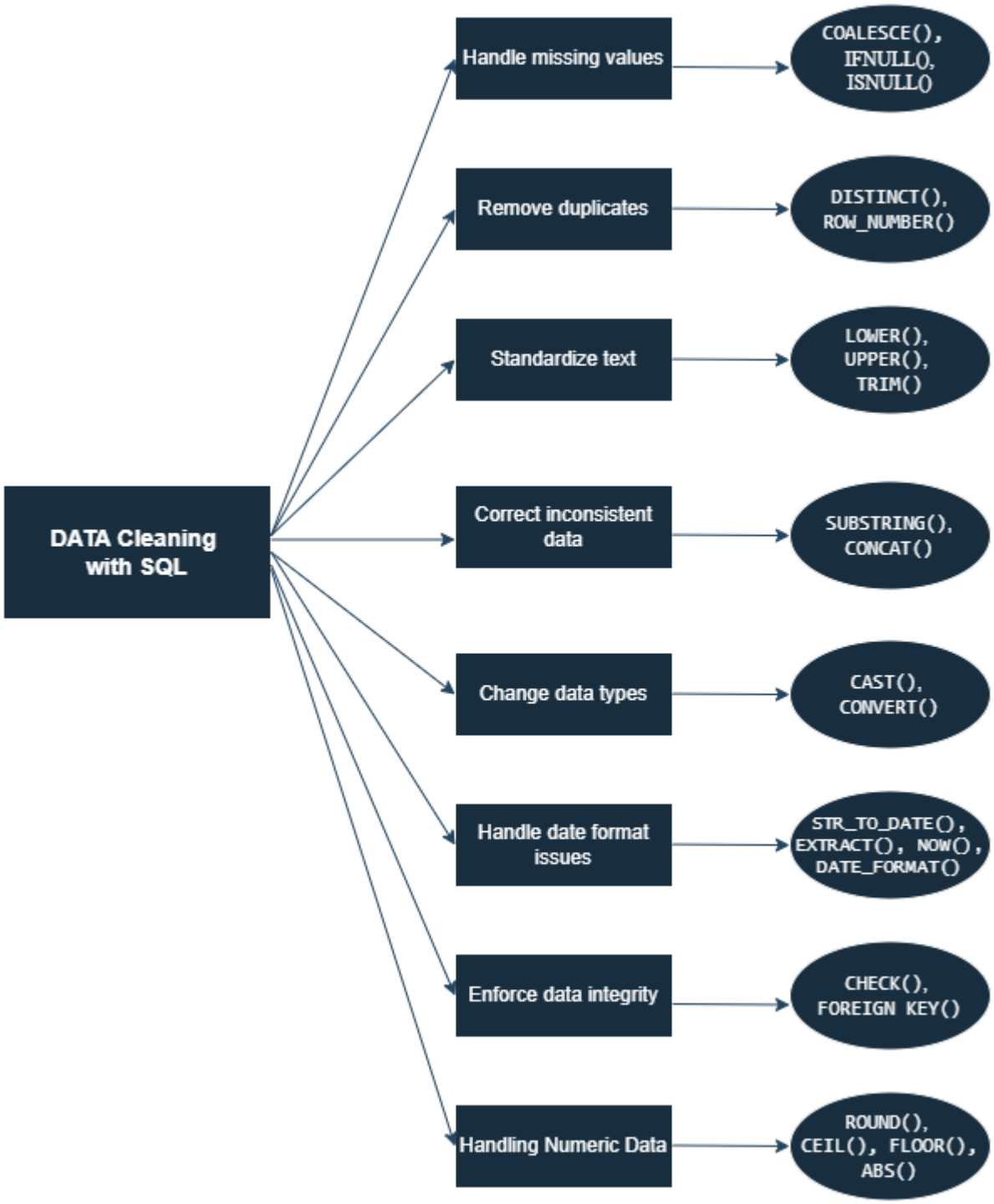
DATA CLEANING

Transform Raw DATA into Actionable Insights Using SQL

Created By: **Muhammad Umar Hanif**

WHAT IS DATA CLEANING?

Data cleaning in SQL involves identifying and correcting errors or inconsistencies in data to improve its quality and accuracy. It includes tasks like:



This process ensures that the data is clean, consistent, and ready for analysis.

01. Handle Missing Values:

The SQL functions

- `COALESCE()`
- `IFNULL()`
- `ISNULL()`

are used to handle missing or `NULL` values in databases. Let's explore how these functions work with a common example.

1.1 - COALESCE():

Returns the first non-`NULL` value from the list of arguments.

Example:

Suppose you have a table called employees with columns for `base_salary`, `bonus`, and `total_compensation`, where some values in `bonus` and `total_compensation` might be `NULL`.

id	name	base_salary	bonus	total_compensation
1	John	50000	NULL	NULL
2	Sarah	60000	5000	NULL
3	David	45000	NULL	NULL
4	Alice	70000	7000	77000

Case 01:

```
SELECT name,
       base_salary,
       COALESCE(bonus, 0) AS bonus,
       COALESCE(total_compensation, base_salary + COALESCE(bonus, 0)) AS
total_compensation
FROM employees;
```

name	base_salary	bonus	total_compensation
John	50000	0	50000
Sarah	60000	5000	65000
David	45000	0	45000
Alice	70000	7000	77000

Explanation:

- If bonus is `NULL`, it is replaced with 0.
- If total_compensation is `NULL`, it is replaced with `base_salary + bonus`.

Case 02:

```
SELECT name,
       base_salary,
       COALESCE(bonus, base_salary, 0) AS bonus,
       COALESCE(total_compensation, base_salary + COALESCE(bonus, 0)) AS
total_compensation
FROM employees;
```

name	base_salary	bonus	total_compensation
John	50000	50000	100000
Sarah	60000	5000	65000
David	45000	45000	90000
Alice	70000	7000	77000

Explanation:

Here, COALESCE() will give you the value from bonus if it's available, or it will take base_salary if bonus is missing. If both are missing, it will return 0.

1.2 - IFNULL():

IFNULL() is a function in SQL that checks if a value is NULL (empty) and, if it is, replaces it with a value you choose. If the value isn't NULL, it returns the original value.

Example:

Suppose you have a table called employees with columns for base_salary, bonus, and total_compensation, where some values in bonus and total_compensation might be NULL.

id	name	base_salary	bonus	total_compensation
1	John	50000	NULL	NULL
2	Sarah	60000	5000	NULL
3	David	45000	NULL	NULL
4	Alice	70000	7000	77000

```
SELECT name,
       base_salary,
       IFFNULL(bonus, 0) AS bonus,
       IFFNULL(total_compensation, base_salary + IFFNULL(bonus, 0)) AS
total_compensation
FROM employees;
```

name	base_salary	bonus	total_compensation
John	50000	0	50000
Sarah	60000	5000	65000

David	45000	0	45000
Alice	70000	7000	77000

Explanation:

- If bonus is NULL, it is replaced with 0.
- If total_compensation is NULL, it is replaced with base_salary + bonus.

Difference Between COALESCE() and IFNULL() in SQL:

- **COALESCE()**: Can take multiple arguments and returns the first non-NULL value.
- **IFNULL()**: Only takes two arguments — if the first one is NULL, it returns the second.

1.3 - ISNULL():

The ISNULL() function in SQL is used to check if a value is NULL. It returns a specified value if the original value is NULL. It's similar to IFNULL() but is more common in SQL Server.

IFNULL() and ISNULL() are simpler and more focused, but support only two arguments.

Example:

Suppose you have a table called employees with columns for base_salary, bonus, and total_compensation, where some values in bonus and total_compensation might be NULL.

id	name	base_salary	bonus	total_compensation
1	John	50000	NULL	NULL
2	Sarah	60000	5000	NULL
3	David	45000	NULL	NULL
4	Alice	70000	7000	77000

```
SELECT name,
       base_salary,
       ISNULL(bonus, 0) AS bonus,
       ISNULL(total_compensation, base_salary + ISNULL(bonus, 0)) AS
total_compensation
FROM employees;
```

name	base_salary	bonus	total_compensation
John	50000	0	50000
Sarah	60000	5000	65000
David	45000	0	45000
Alice	70000	7000	77000

Here, ISNULL() checks if the bonus is NULL. If it is, it returns 0, otherwise it returns the original bonus.

02. Remove Duplicates:

In SQL, both DISTINCT and ROW_NUMBER() can be used to remove duplicates, but they work in different ways and serve different purposes. Let's go through each method with an example and explain the difference between them.

2.1 - DISTINCT():

The DISTINCT keyword removes duplicate rows from the result set. It checks all the columns you specify and returns only unique rows.

Example:
Suppose you have the following table of employee records with some duplicate entries:

You want to remove the duplicate rows (same name, department, and salary).

employee_id	name	department	salary
1	John	HR	5000
2	Sarah	IT	6000
3	John	HR	5000
4	Mike	Sales	4500

Query:

```
SELECT DISTINCT(name), department, salary
FROM employees;
```

name	department	salary
John	HR	5000
Sarah	IT	6000
Mike	Sales	4500

Here, DISTINCT() has removed the duplicate row of John.

2.2 - ROW_NUMBER():

The ROW_NUMBER() function assigns a unique number to each row within a partition of data, based on an ORDER BY clause. You can use this to identify and remove duplicates by keeping only the first occurrence of each set of duplicates.

Example:
If you want to keep only one row for each employee based on name and department but remove duplicates based on the combination of those columns, you can use ROW_NUMBER().

```sql
WITH RankedEmployees AS (
    SELECT employee_id, name, department, salary,
           ROW_NUMBER() OVER (PARTITION BY name, department ORDER BY
employee_id) AS row_num
    FROM employees
)
SELECT employee_id, name, department, salary
FROM RankedEmployees
WHERE row_num = 1;
```

employee_id	name	department	salary
1	John	HR	5000
2	Sarah	IT	6000
4	Mike	Sales	4500

Here, ROW_NUMBER() assigns a unique number to each row partitioned by name and department. We only keep the rows where row_num = 1, effectively removing duplicates.

Key Differences between DISTINCT() and ROW_NUMBER():

- **DISTINCT**:
  - Removes duplicates by considering all specified columns.
  - It does not allow you to control which duplicate to keep.
  - Simpler to use if you just want to remove duplicates based on exact matches.
- **ROW_NUMBER()**:
  - Provides more flexibility by assigning a unique number to each row.
  - Allows you to remove duplicates based on more complex logic, such as deciding which duplicate to keep (based on ordering).
  - Useful when you need to retain more control over how duplicates are handled (e.g., keeping the latest or the earliest record based on another column).

When to Use Which?

- Use **DISTINCT** when you simply want to eliminate exact duplicate rows based on specific columns.
- Use **ROW_NUMBER()** when you need to remove duplicates but want more control over which duplicates to keep, especially when duplicates exist in a more complex way.

03. STANDARDIZE TEXT:

In SQL, functions like LOWER(), UPPER(), and TRIM() are commonly used to standardize text data. They help ensure that text is consistent, which is important for comparisons, storage, and presentation.

3.1 - LOWER():

Converts all characters in a string to lowercase.

Input Table:

name
John Doe
SARAH SMITH
MiKe JOHN

```
SELECT LOWER(name) AS standardized_name
FROM employees;
```

standardized_name
john doe
sarah smith
mike john

3.2 - UPPER():

Converts all characters in a string to uppercase.

Input Table:

name
John Doe
SARAH SMITH
MiKe JOHN

```
SELECT UPPER(name) AS standardized_name
FROM employees;
```

standardized_name
JOHN DOE
SARAH SMITH
MIKE JOHN

3.3 - TRIM():

Removes leading and trailing spaces from a string.

Example:

Suppose you have a table named customer_feedback, which contains customer reviews. Some of these reviews have leading and trailing spaces, which can affect data analysis and reporting.

feedback_id	review
1	Great product!
2	Excellent service!
3	Average quality.
4	Not satisfied with the service.
5	Would buy again!

```
SELECT
    feedback_id,
    TRIM(review) AS cleaned_review
FROM
    customer_feedback;
```

feedback_id	review
1	Great product!
2	Excellent service!
3	Average quality.
4	Not satisfied with the service.
5	Would buy again!

Explanation

- **Feedback IDs**: Remain unchanged.
- **Cleaned Reviews**: The TRIM() function removes any leading and trailing spaces from each review. This ensures that the feedback is clean and ready for further analysis, such as sentiment analysis or reporting.

04. CORRECT INCONSISTENT DATA:

Correcting inconsistent data in SQL can often involve string manipulation functions such as SUBSTR() and CONCAT(). Let's create a scenario where we specifically need to use SUBSTR() and CONCAT() together to correct inconsistent data.

Example:

Imagine you have a table named products, which stores product codes in inconsistent formats. Some product codes may have leading or trailing spaces, and some may have additional characters that need to be standardized.

product_id	product_code
1	ABC-123
2	def456
3	ghi-789
4	JKL-0-001
5	MNO_234

Objective

1. **Remove any leading or trailing spaces** from the product codes.
2. **Ensure all product codes follow a standard format**: the code should start with "PROD-", followed by a numeric part extracted from the existing product code.

SQL Query Using SUBSTR() and CONCAT()

The approach will involve trimming the spaces, extracting the relevant parts of the product code using SUBSTR(), and concatenating them into a standardized format using CONCAT().

```
UPDATE products
SET product_code = CONCAT('PROD-',
                   SUBSTR(TRIM(product_code),
                   INSTR(TRIM(product_code), '-') + 1));
```

product_id	product_code
1	PROD-123
2	PROD-456
3	PROD-789
4	PROD-001
5	PROD-234

Explanation of the Query

1. **TRIM(product_code)**: This removes leading and trailing spaces from each product code.
2. **INSTR(TRIM(product_code), '-') + 1**: This finds the position of the first - in the trimmed product code and adds 1 to get the starting position of the numeric part.
3. **SUBSTR(..., INSTR(...) + 1)**: This extracts the substring starting from the character immediately after the -, which will give us the numeric part of the product code.
4. **CONCAT('PROD-', ...)**: This concatenates "PROD-" with the extracted numeric part to create the standardized product code.

05. CHANGE DATA TYPES:

You can use CAST() and CONVERT() in SQL to change data types of columns or values, and they are often used for converting between string, numeric, and date formats. Below is an example that demonstrates both CAST() and CONVERT() functions.

Example Scenario

We have a table `sales` with columns for sale_id, sale_amount, and sale_date. You want to:

1. Convert the sale_id (which is an integer) into a string for some report.
2. Convert sale_date (which is a DATETIME) into a VARCHAR, but in a specific format: dd/mm/yyyy.

sale_id	sale_amount	sale_date
1	1000.50	2024-10-01 14:30:00
2	1500.00	2024-10-02 09:00:00
3	750.25	2024-10-03 16:45:00

SQL Query Using Both CAST() and CONVERT():

```
SELECT
    CAST(sale_id AS VARCHAR(10)) AS sale_id_string,
    CAST(sale_amount AS DECIMAL(10, 2)) AS sale_amount_decimal,
    CONVERT(VARCHAR(10), sale_date, 103) AS sale_date_formatted
FROM sales;
```

Explanation of the Query:

1. **CAST(sale_id AS VARCHAR(10))**:
    - Converts the sale_id (an integer) into a VARCHAR of up to 10 characters.
    - This is a simple, straightforward type conversion with no extra formatting options.
2. **CAST(sale_amount AS DECIMAL(10, 2))**:
    - Converts sale_amount to a DECIMAL(10, 2), ensuring the amount is represented with two decimal places.
    - This shows that CAST() can handle numeric type conversions as well.
3. **CONVERT(VARCHAR(10), sale_date, 103)**:
    - Converts the sale_date (a DATETIME type) to a string in dd/mm/yyyy format.
    - The third parameter 103 is a style number in SQL Server that specifies the exact format (103 represents dd/mm/yyyy).
    - CONVERT() is used here because it allows formatting options that CAST() does not.

sale_id_string	sale_amount_decimal	sale_date_formatted
1	1000.50	01/10/2024
2	1500.00	02/10/2024
3	750.25	03/10/2024

Key Differences Between CAST() and CONVERT():

1. **CAST()**:

   - **Basic Usage**: Converts one data type to another.
   - **Simple**: Mostly used when you don't need specific formatting.
   - **Portable**: Works across many SQL databases (ANSI SQL compliant).

   Example: Converting an integer to a string:

   ```
   SELECT CAST(sale_id AS VARCHAR(10)) FROM sales;
   ```

2. **CONVERT()**:

   - **Versatile**: Allows additional formatting, particularly with DATETIME types.
   - **Specific to SQL Server**: Offers flexibility for converting and formatting dates, numbers, etc.

   Example: Converting DATETIME to string with a specific format:

   ```
   SELECT CONVERT(VARCHAR(10), sale_date, 103) FROM sales;
   ```

Summary:

- Use **CAST()** when you need simple, straightforward data type conversion that is portable across different database systems.
- Use **CONVERT()** in SQL Server when you need to apply specific formatting, especially for DATETIME values or when you need more control over the output format.

06. HANDLE DATE FORMAT ISSUES:

When handling date format issues in SQL, particularly in MySQL, we use functions like STR_TO_DATE(), EXTRACT(), NOW(), and DATE_FORMAT() to manipulate and extract dates from various formats.

Let's explore these functions with a practical example using a table named orders.

Example Scenario:

You have an orders table where:

- order_id stores the order identification numbers.
- order_date stores the date as a string in inconsistent formats (e.g., DD/MM/YYYY, MM-DD-YYYY).
- You need to:
    1. Convert these string-formatted dates into actual DATE types.
    2. Extract specific parts of the date (like year or month).
    3. Format the date into a more user-friendly format for reporting purposes.
    4. Get the current date for comparison purposes.

order_id	order_date	amount
1	26/09/2024	100
2	09-27-2024	150
3	28-09-2024	200

SQL Queries Demonstrating STR_TO_DATE(), EXTRACT(), NOW(), and DATE_FORMAT():

Query 1: STR_TO_DATE() to Convert Strings into Dates

- STR_TO_DATE() is used to convert strings into proper DATE types by specifying the input format.

```
SELECT
    order_id,
    STR_TO_DATE(order_date, '%d/%m/%Y') AS formatted_date_1,
    STR_TO_DATE(order_date, '%m-%d-%Y') AS formatted_date_2
FROM orders;
```

order_id	formatted_date_1	formatted_date_2
1	2024-09-26	NULL
2	NULL	2024-09-27
3	2024-09-28	NULL

- For order_id = 1, the date '26/09/2024' is converted using '%d/%m/%Y'.
- For order_id = 2, the date '09-27-2024' is converted using '%m-%d-%Y'.
- order_id = 3 is already in '%d/%m/%Y' format.

Query 2: EXTRACT() to Extract Specific Parts of the Date

```
SELECT
    order_id,
    EXTRACT(YEAR FROM STR_TO_DATE(order_date, '%d/%m/%Y')) AS order_year,
    EXTRACT(MONTH FROM STR_TO_DATE(order_date, '%d/%m/%Y')) AS order_month
FROM orders
WHERE order_id = 1;
```

order_id	order_year	order_month
1	2024	9

order_id = 1: The year is 2024 and the month is 9 extracted from the date '26/09/2024'.

Query 3: NOW() to Get the Current Date and Time

```
SELECT
    NOW() AS current_datetime
FROM orders
LIMIT 1;
```

current_datetime
2024-10-01 12:30:00

This would return the current system date and time at the time of query execution. In this case, it is assumed to be 2024-10-01 12:30:00.

Query 4: DATE_FORMAT() to Format Dates

```
SELECT
    order_id,
    DATE_FORMAT(STR_TO_DATE(order_date, '%d/%m/%Y'), '%M %d, %Y') AS
formatted_order_date
FROM orders
WHERE order_id = 1;
```

order_id	formatted_order_date
1	September 26, 2024

The date '26/09/2024' is reformatted as 'September 26, 2024'.

Complete Query Combining All Functions:

```
SELECT
    order_id,
    STR_TO_DATE(order_date, '%d/%m/%Y') AS formatted_date1,
    STR_TO_DATE(order_date, '%m-%d-%Y') AS formatted_date2,
    EXTRACT(YEAR FROM STR_TO_DATE(order_date, '%d/%m/%Y')) AS order_year,
    EXTRACT(MONTH FROM STR_TO_DATE(order_date, '%d/%m/%Y')) AS order_month,  -
    DATE_FORMAT(STR_TO_DATE(order_date, '%d/%m/%Y'), '%M %d, %Y') AS formatted_date3,
    NOW() AS current_datetime
FROM orders;
```

order_id	formatted_date1	formatted_date2	order_year	order_month	formatted_date3	current_datetime
1	2024-09-26	NULL	2024	9	September 26, 2024	2024-10-01 12:30:00
2	NULL	2024-09-27	NULL	NULL	NULL	2024-10-01 12:30:00
3	2024-09-28	NULL	2024	9	September 28, 2024	2024-10-01 12:30:00

- For order_id = 1, formatted_date1 is populated because it's in '%d/%m/%Y' format.
- For order_id = 2, formatted_date2 is populated because it's in '%m-%d-%Y' format.
- Current system date and time (NOW()) are the same for all rows.

07. ENFORCE DATA INTEGRITY:

Data integrity can be enforced using constraints like CHECK and FOREIGN KEY. Here's an example to demonstrate how these constraints work.

CHECK Constraint:
The CHECK constraint ensures that a condition must be true for each row in the table. For example, let's create a Customers table where the age must be between 18 and 100.

01. Creating the Customers Table with CHECK Constraint:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Age INT,
    Email VARCHAR(100),
    CHECK (Age >= 18 AND Age <= 100)
);
```

02. Creating the Orders Table with FOREIGN KEY Constraint

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    Amount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

03. Inserting Data into Customers Table

```
INSERT INTO Customers (CustomerID, FirstName, LastName, Age, Email)
VALUES
(1, 'John', 'Doe', 30, 'john.doe@example.com'),
(2, 'Jane', 'Smith', 25, 'jane.smith@example.com'),
(3, 'Emily', 'Johnson', 45, 'emily.johnson@email.com');
```

CustomerID	FirstName	LastName	Age	Email
1	John	Doe	30	john.doe@example.com
2	Jane	Smith	25	jane.smith@example.com
3	Emily	Johnson	45	emily.johnson@email.com

04. Attempt to Insert Invalid Data (Will Fail Due to CHECK Constraint)

```
INSERT INTO Customers (CustomerID, FirstName, LastName, Age, Email)
VALUES (4, 'Jake', 'Williams', 16, 'jake.williams@example.com');
```

Error Message:

```
Error: CHECK constraint violation on Age. Age must be between 18 and
100.
```

5. Inserting Data into Orders Table

```
INSERT INTO Orders (OrderID, OrderDate, CustomerID, Amount)
VALUES
(1001, '2024-10-16', 1, 250.75),
(1002, '2024-10-17', 2, 100.00);
```

OrderID	OrderDate	CustomerID	Amount
1001	2024-10-16	1	250.75
1002	2024-10-17	2	100.00

06. Attempt to Insert Invalid Order (Will Fail Due to FOREIGN KEY Constraint)

```
INSERT INTO Orders (OrderID, OrderDate, CustomerID, Amount)
VALUES (1003, '2024-10-18', 4, 150.00);
```

Error Message:

```
Error: FOREIGN KEY violation. CustomerID 4 does not exist in the
Customers table.
```

Summary

- The **CHECK constraint** on the Customers table ensures that only customers with valid ages (18-100) are inserted.
- The **FOREIGN KEY constraint** on the Orders table ensures that orders can only reference valid customers from the Customers table.

By enforcing these constraints, the database maintains integrity and prevents invalid or inconsistent data from being entered.

08. HANDLE NUMERIC VALUES:

You can handle numeric values using the functions ROUND(), CEIL(), FLOOR(), and ABS() in SQL. Here is a single dataset with examples of how each function works.

Consider the following example with table: sales_data,

sale_id	sale_amount
1	234.567
2	-78.423
3	456.789
4	123.001
5	-65.999

I. ROUND() Function

The ROUND() function is used to round a number to a specified number of decimal places.

```
SELECT
    sale_id,
    sale_amount,
    ROUND(sale_amount, 2) AS rounded_amount_2_decimals,
    ROUND(sale_amount, 0) AS rounded_to_nearest_integer
FROM sales_data;
```

sale_id	sale_amount	rounded_amount_2_decimals	rounded_to_nearest_integer
1	234.567	234.57	235
2	-78.423	-78.42	-78
3	456.789	456.79	457
4	123.001	123.00	123
5	-65.999	-66.00	-66

- ROUND(sale_amount, 2): Rounds the number to 2 decimal places.
- ROUND(sale_amount, 0): Rounds the number to the nearest integer.

II. CEIL() Function

The CEIL() (Ceiling) function rounds a number **up** to the nearest integer, regardless of the decimal part.

```
SELECT
    sale_id,
    sale_amount,
    CEIL(sale_amount) AS ceiling_value
FROM sales_data;
```

sale_id	sale_amount	ceiling_value
1	234.567	235
2	-78.423	-78
3	456.789	457
4	123.001	124
5	-65.999	-65

- CEIL() rounds the numbers up to the nearest integer.

III. FLOOR() Function

The FLOOR() function rounds a number **down** to the nearest integer, ignoring the decimal part.

```
SELECT
    sale_id,
    sale_amount,
    FLOOR(sale_amount) AS floor_value
FROM sales_data;
```

sale_id	sale_amount	floor_value
1	234.567	234
2	-78.423	-79
3	456.789	456
4	123.001	123
5	-65.999	-66

- **FLOOR()** rounds the numbers **down** to the nearest integer.

IV. ABS() Function

The ABS() function returns the absolute (positive) value of a number.

```
SELECT
    sale_id,
    sale_amount,
    ABS(sale_amount) AS absolute_value
FROM sales_data;
```

sale_id	sale_amount	absolute_value
1	234.567	234.567
2	-78.423	78.423
3	456.789	456.789
4	123.001	123.001
5	-65.999	65.999

Summary

- **ROUND()**: Rounds the number to a specified number of decimal places.
- **CEIL()**: Rounds the number up to the nearest integer.
- **FLOOR()**: Rounds the number down to the nearest integer.
- **ABS()**: Returns the absolute (positive) value of a number.

These functions help you clean, format, and manipulate numerical data in SQL effectively.