Conditional Expressions and Operators

- Section Overview
  - CASE
  - COALESCE
  - NULLIF
  - CAST
  - Views
  - Import and Export Functionality

- These keywords and functions will allow us to add logic to our commands and workflows in SQL.

Let’s get started!

CASE

- We can use the **CASE** statement to only execute SQL code when certain conditions are met.
- This is very similar to **IF/ELSE** statements in other programming languages.

- There are two main ways to use a CASE statement, either a general CASE or a CASE expression.
- Both methods can lead to the same results.
- Let’s first show the syntax for a “general” CASE.

- General Syntax
  - ```
    CASE
            WHEN condition1 THEN result1

            WHEN condition2 THEN result2

    ELSE some_other_result

    END
    ```
    ```

- Simple Example
  - ```
    SELECT a,
    CASE WHEN a =  1 THEN ‘one’
         WHEN a = 2 THEN ‘two’
    ELSE ‘other’
    ```

a
1
2
    ```

- Simple Example
  - ```
    SELECT a,
    CASE WHEN a =  1 THEN ‘one’
         WHEN a = 2 THEN ‘two’
    ELSE ‘other’ AS label
    END
    FROM test;
    ```

a	label
1	one
2	two
    ```

The CASE expression syntax first evaluates an expression then compares the result with each value in the WHEN clauses sequentially.

- *CASE Expression Syntax*
  - ```
    CASE expression

        WHEN value1 THEN result1

        WHEN value2 THEN result2

        ELSE some_other_result

    END
    ```
    ```

- Rewriting our previous example:
  - ```
    SELECT a,
            CASE a
             WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
           END
    FROM test;
    ```

a	label
1	one
2	two
    ```

```sql
-- Simple CASE statement
SELECT customer_id, first_name, last_name,
       CASE
           WHEN age(date_of_birth) >= INTERVAL '65 years' THEN 'Senior'
           WHEN age(date_of_birth) >= INTERVAL '25 years' THEN 'Adult'
           ELSE 'Young Adult'
       END AS age_group
FROM customers;

-- CASE with aggregation
SELECT account_type,
       COUNT(*) AS total_accounts,
       SUM(CASE WHEN balance > 50000 THEN 1 ELSE 0 END) AS high_balance_count,
       SUM(CASE WHEN balance <= 1000 THEN 1 ELSE 0 END) AS low_balance_count,
       AVG(CASE WHEN balance > 0 THEN balance ELSE NULL END) AS avg_positive_balance
FROM accounts
GROUP BY account_type;
```

COALESCE

- The COALESCE function accepts an unlimited number of arguments. It returns the first argument that is not null. If all arguments are null, the COALESCE function will return null.
  - `COALESCE (arg_1,arg_2,...,arg_n)`

- Example
  - `SELECT COALESCE (1, 2)`
    - 1
  - `SELECT COALESCE(NULL, 2, 3)`
    - 2

The COALESCE function becomes useful when querying a table that contains null values and substituting it with another value.

Let’s see a simple example.

- Table of Products
  - What is the final price?

Item	Price	Discount
A	100	20
B	300	null
C	200	10

- `SELECT item,(price - discount) AS final`
  `FROM table`

Item	final
A	80
B	null
C	190

- `SELECT item,(price - discount) AS final FROM table`
- Doesn’t work for item B, should be 300.

Item	final
A	80
B	null
C	190

```
SELECT item,(price - COALESCE(discount,0)) AS final
FROM table
```

Item	final
A	80
B	300
C	190

Keep the COALESCE function in mind in case you encounter a table with null values that you want to perform operations on!

CAST

- The CAST operator let’s you convert from one data type into another.
- Keep in mind not every instance of a data type can be CAST to another data type, it must be reasonable to convert the data, for example ‘5’ to an integer will work, ‘five’ to an integer will not.

- Syntax for CAST function
  - `SELECT CAST('5' AS INTEGER)`

- PostgreSQL CAST operator
  - `SELECT '5'::INTEGER`

- Keep in mind you can then use this in a SELECT query with a column name instead of a single instance.
  - ```
    SELECT CAST(date AS TIMESTAMP)
    FROM table
    ```
    ```

NULLIF

- The NULLIF function takes in 2 inputs and returns NULL if both are equal, otherwise it returns the first argument passed.
  - `NULLIF(arg1,arg2)`
- Example
  - `NULLIF(10,12)`
    - Returns 10
  - `NULLIF(10,10)`
    - Returns NULL

VIEWS

- Often there are specific combinations of tables and conditions that you find yourself using quite often for a project.
- Instead of having to perform the same query over and over again as a starting point, you can create a VIEW to quickly see this query with a simple call.

- A view is a database object that is of a stored query.
- A view can be accessed as a virtual table in PostgreSQL.
- Notice that a view does not store data physically, it simply stores the query.
- You can also update and alter existing views.

```sql
-- Simple view for customer summary

CREATE VIEW customer_summary AS
SELECT c.customer_id, c.first_name, c.last_name, c.email,
       COUNT(a.account_id) AS account_count,
       COALESCE(SUM(a.balance), 0) AS total_balance,
       MAX(a.date_opened) AS latest_account_date
FROM customers c
LEFT JOIN accounts a ON c.customer_id = a.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.email;
```

```
-- Using views in queries

SELECT * FROM customer_summary WHERE total_balance > 50000;
SELECT customer_tier, COUNT(*) FROM high_value_customers GROUP BY customer_tier;

-- Updating views (if updatable)

UPDATE customer_public_info SET first_name = 'John' WHERE customer_id = 1;

-- Drop view

DROP VIEW IF EXISTS customer_public_info;
```

Challenge!

1. Challenge

Department: Customer Segmentation

Request: "Create a customer classification report using conditional logic. We need to categorize customers based on their annual income levels and credit scores for our marketing campaigns.

Expected Answer
Customer classification with income categories and credit score ratings using CASE statements.

Hints
- Use **customers** table
- Use CASE statement for income categorization
- Use COALESCE to handle NULL income values
- Use CASE for credit score ratings (Excellent: ≥750, Good: 650-749, Fair: 550-649, Poor: <550)
- Handle NULL credit scores appropriately

1. Challenge

Specific Requirements:

Income Categories: Create an 'income_category' column with these exact conditions:

- 'High Income' for customers with annual_income ≥ $100,000
- 'Middle Income' for customers with annual_income between $50,000 and $99,999
- 'Low Income' for customers with annual_income < $50,000
- 'Unknown' for customers where annual_income is NULL or missing

Credit Score Ratings: Create a 'credit_rating' column with these exact conditions:

- 'Excellent' for credit_score ≥ 750
- 'Good' for credit_score between 650 and 749
- 'Fair' for credit_score between 550 and 649
- 'Poor' for credit_score < 550
- 'Not Rated' for credit_score that is NULL or missing

1. Challenge

Specific Requirements:

- **Contact Information**:

  Create a 'primary_contact' column that shows:

  Email address if available Phone number if email is not available but phone exists

  'No Contact Info' if both email and phone are missing or NULL
- **Output Columns**: customer_number, first_name, last_name, annual_income, income_category, credit_score, credit_rating, primary_contact
- **Sort**: By annual_income in descending order, with NULL values appearing last"

2. Challenge

Department: Account Management

Request: "Create a comprehensive account status report with data cleansing and transformation. Our operations team needs clean data for their monthly account review process."

Expected Answer
Account report with data cleansing, type conversions, and risk categorization.

Hints
- Use **accounts** table
- Use COALESCE to handle NULL minimum_balance values
- Use CAST to convert balance to integer
- Use date functions with CASE for account age categories
- Create risk categories based on balance vs minimum balance ratio
- Use NULLIF to handle edge cases in calculations

2. Challenge

Specific Requirements:

- **Balance Conversion**: Convert the balance column to INTEGER (whole numbers) for reporting purposes - name it 'balance_rounded'
- **Minimum Balance Handling**: Create a 'min_balance_clean' column where:
  - Use the actual minimum_balance if it exists
  - Use 0 if minimum_balance is NULL or missing
- **Account Age Calculation**: Create an 'account_age_years' column showing the account age in whole years from date_opened to current date
- **Account Age Categories**: Create an 'account_age_category' column with these conditions:
  - 'New' for accounts less than 1 year old
  - 'Established' for accounts between 1 year and less than 3 years old
  - 'Mature' for accounts 3 years or older
- **Balance Risk Assessment**: Create a 'balance_risk_category' column with these conditions:
  - 'No Minimum' if the cleaned minimum balance equals 0
  - 'Well Above Minimum' if balance is 5 times or more than minimum balance
  - 'Above Minimum' if balance is 2 times or more than minimum balance (but less than 5 times)
  - 'Meets Minimum' if balance is equal to or greater than minimum balance (but less than 2 times)
  - 'Below Minimum' if balance is less than minimum balance
- **Status Data Cleaning**: Create a 'status_clean' column that removes any empty strings (convert empty strings to NULL)
- **Output Columns**: account_number, account_type, balance_rounded, min_balance_clean, account_age_years, account_age_category, balance_risk_category, status_clean
- **Sort**: By balance in descending order"

3. Challenge

Department: Transaction Analysis

Request: "Create a VIEW called 'transaction_summary_view' that provides basic transaction analysis with some conditional formatting. This view will be used by business users for simple transaction monitoring."

Expected Answer
A view creation with comprehensive transaction analysis and conditional logic.

Hints

- Create a VIEW using only the **transactions** table
- Use CASE for amount categories
- Use simple CASE for positive/negative amounts
- Use EXTRACT(HOUR) with simple CASE for business hours
- Use COALESCE for description handling

3. Challenge

Specific Requirements:

- **Base Data**: Include transaction_id, account_id, transaction_date, and amount from transactions table

- **Transaction Amount Categories**: Create an 'amount_category' column based on absolute value of amount:
  - 'Large' for amounts ≥ $10,000
  - 'Medium' for amounts ≥ $1,000 but < $10,000
  - 'Small' for amounts < $1,000

- **Transaction Type**: Create a 'transaction_flow' column:
  - 'Money In' for positive amounts (amount > 0)
  - 'Money Out' for negative amounts (amount < 0)
  - 'Zero' for zero amounts (amount = 0)
- **Business Hours Check**: Create a 'business_hours' column based on the hour of transaction_date:
  - 'Business Hours' for hours 9 AM to 5 PM (hours 9, 10, 11, 12, 13, 14, 15, 16, 17)
  - 'After Hours' for all other hours

- **Description Handling**: Create a 'description_clean' column:
  - Use the original description if it exists and is not NULL
  - Use 'No Description' if description is NULL