2021-03-31T17:00:00+02:00 | Andrew Bone

What Is CASE in SQL?

SQL CASE is a very useful expression that provides if-else logic to your SQL queries. It's a slightly more advanced topic, but you'll need it when preparing reports – it will deliver massive value to your personal and professional projects.

The SQL `CASE` statement is a control flow tool that allows you to add if-else logic to a query. Generally speaking, you can use the `CASE` statement anywhere that allows a valid expression – e.g. with the `SELECT`, `WHERE`, and `GROUP BY` clauses.

The `CASE` expression goes through each condition and returns a value when the first condition is met. Once a condition is true, `CASE` will return the stated result. If no conditions are true, it will return the value in the `ELSE` clause. If there is no ELSE and no conditions are true, it returns NULL.

Simple SQL CASE Example

Here is the syntax for the SQL `CASE` expression:

```
CASE
        WHEN condition_1 THEN result_1
        WHEN condition_2 THEN result_2
        ELSE else_result
END
```

In this syntax, SQL `CASE` matches the value with either `condition_1` or `condition_2`. If a match is found, the statement will return the corresponding result (`result_1` if the value matches `condition_1` or `result_2` if it matches `condition_2`). If the value does not match either condition, the `else_result` is returned. The `ELSE` statement is optional and provides a way to capture values not specified in the `WHEN..THEN`

statements. Finally, every `CASE` statement must end with the `END` keyword.

Do you already know SQL but don't know what to do with it? Check out our interactive SQL Practice Set course!

The data type of the SQL `CASE` statement result depends on the context where it is used. For example, if the `CASE` expression is used with CHAR strings, it returns the result as a CHAR string. If the `CASE` expression is used in a numerical context, it returns the result as an integer, a decimal, or a real value. Gaining mastery of this powerful control flow tool creates many new opportunities to retrieve and display data in creative ways, as shown in this article about adding logic to your SQL query with CASE.

Let's apply the SQL `CASE` statement to a practical example. Imagine we have a small grocery store and we use a simple database table to track our stock. The **`stock`** table contains the item, the price of the item, and the quantity of that item currently in stock.

Item	Price	Quantity
Bread	1.59	23
Milk	2.00	3
Coffee	3.29	87
Sugar	0.79	0
Eggs	2.20	53
Apples	1.99	17

What if we wanted a simple description to accompany our data and provide more context for our reports? This is easily accomplished with `CASE WHEN`:

```
SELECT Item, Price,
       CASE
   WHEN Price < 1.00 THEN 'Below $1.00'
       WHEN Price >= 1.00 THEN 'Greater or Equal to $1.00'
       END AS 'Price Description'
  FROM stock
```

First, our `SELECT` states that we want to retrieve data from our `Item` and `Price` columns. Next is our `CASE` statement. When the `Price` is below 1.00, we return the string 'Below $1.00'. When the `Price` is greater or equal to 1.00, we want to return the string, 'Greater or Equal to $1.00'. This is applied to every `Price` value in our table.

We also specify that the values returned by the `CASE WHEN` statement should be in a column called *Price Description*:

Item	Price	Price Description
Brea	1.59	Greater or Equal to $1.00
Milk	2.00	Greater or Equal to $1.00
Coffee	3.29	Greater or Equal to $1.00
Sugar	0.79	Below $1.00
Eggs	2.20	Greater or Equal to $1.00
Apples	1.99	Greater or Equal to $1.00

There we have it! For each row where the `Price` is below 1.00, the string 'Below $1.00' is returned. For `Price` values greater than or equal to 1.00, the string 'Greater or Equal to $1.00' is returned. The results are shown in the *Price Description* column.

SQL CASE WHEN with ELSE

If you're using `ELSE`, this statement must come after each `CASE WHEN` condition you have specified. Suppose we now want to categorize the different prices in our table into 3 different categories:

Items below $1.00.
Items between $1.00 and $3.00.
Items above $3.00.

We will use the `ELSE` statement to handle `Price` values above 3.00:

```
SELECT Item, Price,
       CASE WHEN Price < 1.00 THEN 'Below $1.00'
       WHEN Price >= 1.00 AND Price <= 3.00 THEN 'Between $1.00 and $3.00'
       ELSE 'Above $3.00'
       END AS 'Price Description'
  FROM stock
```

The `Price` of each row is checked to see if it is equal to or below 1.00 or between 1.00 and 3.00. If it falls into one of these categories, the corresponding string is returned. If `Price` is not below 3.00, the `ELSE` statement is reached. Our `ELSE` statement returns the string, 'Above $3.00'.

This is why the ordering of your statements is important. SQL evaluates each `CASE` in order, finally reaching the `ELSE` if no conditions were met.

Item	Price	Price Description
Bread	1.59	Between $1.00 and $3.00
Milk	2.00	Between $1.00 and $3.00
Coffee	3.29	Above $3.00
Sugar	0.79	Below $1.00
Eggs	2.20	Between $1.00 and $3.00
Apples	1.99	Between $1.00 and $3.00

Using Multiple CASES

The main reason someone might choose to use the SQL `CASE` statement is that they want to evaluate multiple conditions. They want to perform a series of checks and turn the results into meaningful data, usually in the form of a report.

Let's say we want to generate a simple report for our **`stock`** table. It will tell us whether the stock level is high, medium, low, or out of stock altogether! This can be easily achieved using `CASE`:

```
SELECT Item,
      CASE WHEN Quantity > 0 AND Quantity <= 20 THEN 'Low'
           WHEN Quantity > 20 AND Quantity <= 50 THEN 'Medium'
           WHEN Quantity > 50 THEN 'High'
           ELSE 'Out Of Stock'
       END AS 'Stock Level'
    FROM stock
```

This is our most complex example so far. Let's break down this SQL query.

Our result will have two columns. The first column is `Item`, which we are explicitly selecting with:

```
SELECT Item
```

The second column is the results column generated by our SQL `CASE WHEN` expressions, which we are calling *Stock Level*:

```
END AS 'Stock Level'
```

Now let's breakdown each condition, in the order SQL would evaluate them.

First, SQL checks whether the `Quantity` is greater than zero and less than or equal to 20.

```
CASE WHEN Quantity > 0 AND Quantity <= 20 THEN 'Low'
```

If this is true, 'Low' is returned and the next row begins to be evaluated.

If the result is false, the evaluator looks at the next `CASE` statement:

```
WHEN Quantity > 20 AND Quantity <= 50 THEN 'Medium'
```

`Quantity` is checked again to see whether the value is greater than 20 and less than or equal to 50, returning the string 'Medium' if this is the case. If this condition is not met, the next condition is checked:

```
WHEN Quantity > 50 THEN 'High'
```

The final `CASE` statement checks if the `Quantity` is greater than 50, returning the string 'High' if it is.

There is one other situation not covered by our different `CASE` statements. What if the `Quantity` of a particular `Item` is 0? Look at our `CASE` statements again, particularly:

```
CASE WHEN Quantity > 0 AND Quantity <= 20 THEN 'Low'
```

We check that `Quantity` is greater than 0, meaning if it is equal to 0, this condition would evaluate as false and the database would continue to check the other `CASE` statements. We have included the `ELSE` statement in our SQL query for this reason:

```
ELSE 'Out Of Stock'
```

This caters to this exact scenario. If the `Quantity` of an `Item` is 0, the SQL evaluator will reach our `ELSE` statement and return 'Out of Stock'.

Executing this query yields the following result:

Item	Stock Level
Bread	Medium
Milk	Low
Coffee	High
Sugar	Out Of Stock
Eggs	High
Apples	Low

We can see that sugar has a `Quantity` of 0, which results in it showing as 'Out of Stock'. Compare the other `Quantity` values in our **`stock`** table with the `Stock Level` shown to make sure you understand how our `CASE` statements work.

Imagine how useful this report would be if there were hundreds of items. A report like this could be sent to purchasing managers on a daily basis, allowing them to maintain stock levels of popular items.

CASE with NULL Values

When using `CASE`, you may notice unwanted NULL values in your result set. Why do these values appear and what actions can you take to remove them? NULL values appear when a value does not match any of the `CASE` or `ELSE` statements you declare. Let's look at a practical example that shows how NULL can be returned.

Imagine that we excluded the `ELSE` statement from our previous example. How would it impact our results? Let's look at the previous query, this time without the `ELSE` statement:

```
SELECT Item,
      CASE WHEN Quantity > 0 AND Quantity <= 20 THEN 'Low'
           WHEN Quantity > 20 AND Quantity <= 50 THEN 'Medium'
           WHEN Quantity > 50 THEN 'High'
       END AS 'Stock Level'
    FROM stock
```

The results would look like this. Pay special attention to the *Stock Level* for sugar:

Item	Stock Level
Bread	Medium
Milk	Low
Coffee	High
Sugar	NULL
Eggs	High
Apples	Low

Without `ELSE` to handle the situation of `Quantity` being zero, our query returns a NULL.

If you have an unwanted NULL value in your `CASE WHEN` results, you may have a scenario that is not covered by your `CASE WHEN` and `ELSE` conditions.

GROUP BY with CASE

As mentioned before, you can use the SQL `CASE` expression with `GROUP BY`. Let's examine a practical example of this.

Imagine we wanted to group items based on their price while also displaying the minimum and maximum price for the low-cost and high-cost groups. This requires the use of the aggregate functions `MIN()` and `MAX()`. The `GROUP BY` statement is often used to group resulting data by one or more columns, and often specifically with aggregate functions. Here is an example of how GROUP BY is used alongside aggregate functions that you can read for more information. Let's break down the SQL query below to show how our desired result set can be achieved:

```
SELECT
CASE WHEN Price >= 2.00 THEN 'High Price Item'
WHEN Price > 0 AND Price < 2.00 THEN 'Low Price Item'
END AS PriceLevel,
Min(Price) as MinimumPrice,
Max(Price) as MaximumPrice

FROM stock
GROUP BY
```

```
CASE WHEN Price >= 2.00 THEN 'High Price Item'
WHEN Price > 0 AND Price < 2.00 THEN 'Low Price Item'

END
```

First, let's analyze the `CASE` statement. It is similar to the previous example.

```
CASE WHEN Price >= 2.00 THEN 'High Price Item'
WHEN Price > 0 AND Price < 2.00 THEN 'Low Price Item'
END AS PriceLevel
```

If the `Price` is greater than or equal to 2.00, the item is classified as a high-price item. If the `Price` is greater than 0 but less than 2.00, the item is a low-price item. These string values are then stored and displayed in the column `PriceLevel`, as specified by the `END AS` alias.

We use the `MIN()` and `MAX()` aggregate functions on our `Price` column. This gets the lowest and highest `Price` of the items in our table.

We use our `GROUP BY` clause to apply these aggregate functions to our two categories of high and low price levels. (Don't worry if this seems complicated; mastering `GROUP BY` requires lots of practice. Check out our SQL Practice track for interactive exercises that hone your GROUP BY technique and other SQL skills.)

Executing this SQL query returns the following result set:

PriceLevel	MinimumPrice	MaximumPrice
High Price Item	2.00	3.29
Low Price Item	0.79	1.99

These are exactly the results we wanted! We can now clearly see the minimum and maximum price of each of the item categories defined in our SQL `CASE WHEN` statement. Refer to our **`stock`** table and note which individual items link to the values shown for `MinimumPrice` and `MaximumPrice`. If we were to add a new item to our **`stock`** table that costs $4.00, you would see the `MaximumPrice` of the "High Price Item" increase to 4.00.

Want to practice basic SQL using hands-on exercises? Try our SQL Practice Set: 88 interactive SQL exercises.

SQL CASE and Reusable Complex Queries

Using SQL `CASE` will allow you to write complex queries that perform a range of actions. We have just shown you how to use `CASE WHEN` with `SELECT`. As shown by this article on using CASE with data modifying statements, `CASE WHEN` case also be used with `INSERT`, `UPDATE`, and `DELETE`. This results in highly reusable queries that can be implemented for reports or applications. If you are interested in learning how to build high-value customized reports, I recommend LearnSQL.com's comprehensive course on creating SQL reports.
