**Lab Exercise 6 – Macro with Jinja Template**

**Lab Exercise: Jinja Macros in dbt with Snowflake**

In this lab exercise, we will explore how to use **Jinja macros** in dbt to simplify and modularize repetitive SQL code in your models. Macros are reusable snippets of SQL code that can be invoked in multiple dbt models, reducing duplication and enhancing maintainability. We will create a macro that handles a common aggregation pattern and use it in different models.

**Scenario**

Imagine you have sales data in Snowflake for multiple stores and months. You often need to calculate total sales and average sales for various months. Instead of writing the same aggregation logic repeatedly, we will create a macro that performs the aggregation and use it across different dbt models.

**Step 1: Create the Macro**

Macros are stored in the macros/ directory in your dbt project. We'll create a macro named aggregate\_sales that:

1. Accepts a list of months (e.g., ['jan', 'feb', 'mar']).
2. Calculates both total sales and average sales for each month.

**1.1 Create the Macro File**

Create a file macros/aggregate\_sales.sql in your dbt project:

-- macros/aggregate\_sales.sql

{% macro aggregate\_sales(months) %}

{% set sql\_parts = [] %}

{% for month in months %}

{% set total\_column = 'SUM(sales\_' ~ month ~ ') AS total\_sales\_' ~ month %}

{% set avg\_column = 'AVG(sales\_' ~ month ~ ') AS avg\_sales\_' ~ month %}

{% do sql\_parts.append(total\_column) %}

{% do sql\_parts.append(avg\_column) %}

{% endfor %}

-- Join the SQL parts into one string

{% set final\_sql = sql\_parts | join(', ') %}

-- Return the final SQL part

{{ return(final\_sql) }}

{% endmacro %}

**Explanation:**

* **Macro Definition**: The macro aggregate\_sales is defined with a parameter months, which is a list of month names (e.g., ['jan', 'feb', 'mar']).
* **Looping**: We use a Jinja loop to generate SQL for each month, calculating both the SUM and AVG of sales for that month.
* **Appending to List**: The do sql\_parts.append() statement appends each part of the SQL (SUM and AVG for each month) to the sql\_parts list.
* **Returning the Final SQL**: The join filter concatenates the list elements into a single string that contains the full SQL code for all months.

**Step 2: Use the Macro in a dbt Model**

Now that we have a macro for aggregating sales, we will use it in a dbt model. Let’s create a model named monthly\_sales\_summary.sql that uses the macro to dynamically generate the aggregation logic for multiple months.

**2.1 Create the Model File**

Create a file models/monthly\_sales\_summary.sql:

-- models/monthly\_sales\_summary.sql

{{ config(

materialized = 'table'

) }}

{% set months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec'] %}

WITH aggregated\_sales AS (

SELECT

store\_id,

sales\_month,

{{ aggregate\_sales(months) }} -- Use the macro to aggregate sales

FROM raw\_db.raw\_data.sales -- Reference the source table directly

GROUP BY store\_id, sales\_month

)

SELECT \*

FROM aggregated\_sales

**Explanation:**

* **Calling the Macro**: The line {{ aggregate\_sales(months) }} calls the aggregate\_sales macro and passes the list of months. The macro returns the SQL for summing and averaging the sales for each month, which is then inserted into the SELECT clause.
* **sales\_month Column**: This model assumes that the sales\_month column exists in your sales table, which helps identify the month for aggregation.
* **Grouping**: The data is grouped by store\_id and sales\_month, and the aggregated sales are calculated for each group.

**Step 3: Run the dbt Model**

1. **Run the Model**: In dbt Cloud, go to **Deploy** -> **Run** or, if running dbt locally, execute the following command:

dbt run --select monthly\_sales\_summary

1. **Check Results**: After the model finishes running, check the output table monthly\_sales\_summary in your Snowflake database. The table should contain the aggregated sales for each store and month, with columns like total\_sales\_jan, avg\_sales\_jan, total\_sales\_feb, avg\_sales\_feb, etc.

**Resulting Table Structure**

Your resulting table might look like this:

| **store\_id** | **sales\_month** | **total\_sales\_jan** | **avg\_sales\_jan** | **total\_sales\_feb** | **avg\_sales\_feb** | **...** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2024-01 | 1000 | 1000 | 1200 | 1200 | ... |
| 2 | 2024-01 | 900 | 900 | 1100 | 1100 | ... |

**Step 4: Extend the Macro Usage**

Now, you can reuse the aggregate\_sales macro in other models where similar sales aggregation is needed, without having to rewrite the same SQL code each time.

For example, you can create another model ***quarterly\_sales\_summary.sql*** that aggregates sales by quarter instead of by month.

Example: Quarterly Aggregation Model

-- models/quarterly\_sales\_summary.sql

{{ config(

    materialized = 'table'

) }}

{% set quarters = [

    ['jan', 'feb', 'mar'],

    ['apr', 'may', 'jun'],

    ['jul', 'aug', 'sep'],

    ['oct', 'nov', 'dec']

] %}

WITH quarterly\_sales AS (

    SELECT

        store\_id,

        -- Assuming sales\_month is in the format '2024-01', we'll extract the month

CASE

            WHEN TO\_CHAR(TO\_DATE(sales\_month, 'YYYY-MM'), 'mon') IN ('Jan', 'Feb', 'Mar') THEN 'Q1'

            WHEN TO\_CHAR(TO\_DATE(sales\_month, 'YYYY-MM'), 'mon') IN ('Apr', 'May', 'Jun') THEN 'Q2'

            WHEN TO\_CHAR(TO\_DATE(sales\_month, 'YYYY-MM'), 'mon') IN ('Jul', 'Aug', 'Sep') THEN 'Q3'

            WHEN TO\_CHAR(TO\_DATE(sales\_month, 'YYYY-MM'), 'mon') IN ('Oct', 'Nov', 'Dec') THEN 'Q4'

            ELSE 'Unknown'  -- For handling unexpected values

        END AS quarter,

        {{ aggregate\_sales(quarters[0]) }}  -- Aggregate for Q1

    FROM raw\_db.raw\_data.sales

    GROUP BY store\_id, quarter

)

SELECT \*

FROM quarterly\_sales

In this model, you could modify the aggregate\_sales macro or use the same one in a different context (e.g., aggregating for each quarter instead of each month).

**Summary**

In this lab exercise, we:

* Created a **Jinja macro** in dbt to encapsulate the logic for aggregating sales data.
* Used the macro in a dbt model to dynamically generate SQL for calculating total and average sales across multiple months.
* Learned how to reuse macros in multiple dbt models, improving the maintainability and flexibility of your dbt project.

Macros are a powerful feature in dbt that can help reduce redundancy, enhance code organization, and make your SQL more dynamic and maintainable.