**Jinja Template in dbt for snowflake**

In dbt, a **Jinja template** is a tool that allows you to write dynamic, reusable, and more manageable SQL code. Jinja, a templating language, is integrated into dbt to add logic, loops, and variables directly within SQL statements. When working with **Snowflake** as a data warehouse, Jinja templates can help generate complex SQL, filter data, and conditionally add or remove SQL code based on different environments or configurations.

**Why Use Jinja in dbt?**

Jinja templates make it easy to:

1. **Parameterize SQL queries**: Instead of hardcoding values, you can set parameters like table names or filters to change based on the environment.
2. **Reuse code**: Common SQL snippets or logic can be reused with Jinja functions, variables, and loops, reducing repetitive code.
3. **Add logic to SQL**: You can use if statements, loops, and other control structures to make your SQL adaptable and versatile.

**Basic Jinja Features in dbt (with Snowflake examples)**

Here’s how Jinja templates are commonly used in dbt models with Snowflake.

**1. Variables**

Jinja allows you to define variables within SQL models, which can either be set directly in the model file or passed in from dbt\_project.yml.

**Example**: Setting a year variable and using it to filter data.

-- models/filter\_by\_year.sql

{% set year = 2024 %}

SELECT \*

FROM {{ ref('my\_table') }}

WHERE YEAR(created\_at) = {{ year }}

In this example, year is defined as 2024 within the model file and used to filter data in the WHERE clause.

**2. Using Variables from dbt\_project.yml**

Instead of defining variables in the model, you can set them in the dbt\_project.yml file, making them globally available to all models.

**dbt\_project.yml**:

vars:

target\_year: 2024

**Model file**:

-- models/filter\_by\_target\_year.sql

SELECT \*

FROM {{ ref('my\_table') }}

WHERE YEAR(created\_at) = {{ var('target\_year') }}

Using {{ var('target\_year') }}, the model file fetches the target\_year variable from dbt\_project.yml.

**3. If Statements**

Jinja allows you to use if statements in SQL, which is useful for adding conditions dynamically.

**Example**: Conditionally adding a column if a certain condition is met.

-- models/conditional\_column.sql

{% set include\_status = true %}

SELECT

customer\_id,

first\_name,

last\_name

{% if include\_status %}

, status -- Only include this column if `include\_status` is true

{% endif %}

FROM {{ ref('customers') }}

If include\_status is true, the status column will be included in the query output; otherwise, it will be omitted.

**4. Loops**

Jinja loops can generate repetitive SQL code, which is especially useful for aggregations or transformations across multiple columns.

**Example**: Summing multiple month columns dynamically.

-- models/monthly\_sales\_summary.sql

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

SELECT

store\_id,

{% for month in months %}

SUM(sales\_{{ month }}) AS total\_sales\_{{ month }}

{% if not loop.last %}, {% endif %}

{% endfor %}

FROM {{ ref('sales') }}

GROUP BY store\_id

This loop generates SQL to sum sales for each month without manually writing each line.

**5. Ref Function**

ref is a Jinja function used in dbt to refer to other models within a project, creating dependencies and ensuring models run in the correct order.

**Example**: Using ref to reference another model.

SELECT \*

FROM {{ ref('another\_model') }}

ref will substitute another\_model with the appropriate schema and table name for the model in Snowflake.

**6. Built-In Jinja Filters**

dbt includes several Jinja filters that help format or modify variables in SQL statements.

**Example**: Using the | upper filter to uppercase text.

-- models/uppercased\_data.sql

SELECT

UPPER(first\_name) AS first\_name\_upper,

{{ 'some\_value' | upper }} AS upper\_static\_value

FROM {{ ref('customers') }}

**Putting It All Together: A Complete Example**

Here’s a full example that combines multiple Jinja features. Assume we want to create a summary model that:

1. Filters data based on a year.
2. Aggregates monthly sales.
3. Conditionally adds columns for specific metrics.

-- models/sales\_summary.sql

{% set year = 2024 %}

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

WITH filtered\_sales AS (

SELECT \*

FROM {{ ref('sales') }}

WHERE YEAR(sales\_date) = {{ year }}

)

SELECT

store\_id,

{% for month in months %}

SUM(sales\_{{ month }}) AS total\_sales\_{{ month }},

AVG(sales\_{{ month }}) AS avg\_sales\_{{ month }}

{% if not loop.last %}, {% endif %}

{% endfor %}

FROM filtered\_sales

GROUP BY store\_id

This model demonstrates how Jinja templates in dbt make Snowflake SQL more dynamic, adaptable, and manageable. With Jinja, you can tailor SQL code to specific needs and simplify the code structure, saving time and reducing errors.