**Snapshot Materialization in dbt for Snowflake**

**Snapshot Materialization in dbt**

Snapshot materialization in dbt is a powerful feature that allows you to capture historical changes to your data over time. This is particularly useful for situations like slowly changing dimensions (SCD), where records may change over time and you need to track these changes rather than just overwriting data.

When you use snapshot materialization in dbt, it stores each record as it was at a given point in time, creating a historical version of the data in the target table. Snapshots can help with auditing, change tracking, and maintaining historical records.

**How Snapshot Materialization Works:**

1. **Capture Historical Data**: When a snapshot model is run, dbt compares the source data with the previous version of the data (the existing snapshot). If there are changes (e.g., a new row or an update), dbt inserts new records into the snapshot table.
2. **Track Changes Over Time**: You can use the unique\_key to identify a record and use columns like updated\_at to determine if a record has changed.
3. **Expiration of Old Records**: If you're using a timestamp-based snapshot, when a record changes (for example, an email address is updated), the previous version is marked as "expired" (with a dbt\_valid\_to timestamp), and a new version of the record is inserted with the new values and a new dbt\_valid\_from timestamp.

**Snapshot Strategies in dbt:**

1. **Timestamp Strategy**:
   * Uses a column (like updated\_at) to track when a record was last modified. If the value in the updated\_at column has changed since the last snapshot, dbt considers the record as "changed" and inserts a new version.
   * This is typically used for **Slowly Changing Dimension Type 2 (SCD2)** scenarios.
2. **Check Strategy**:
   * This strategy checks all columns in the table for changes. If any column value changes, dbt will insert a new version of the record.
   * This approach may not be as efficient as the timestamp strategy because it checks all fields, not just the modified date.

**Steps to Create a Snapshot in dbt:**

Below is a simple example of how to set up snapshot materialization in dbt for Snowflake.

**Step 1: Set Up Your Source Data (In Snowflake)**

Before creating a snapshot, you should have a source table that will serve as the input for your snapshot. Here’s an example of creating a source table called customers in Snowflake:

CREATE OR REPLACE TABLE raw.customers (

customer\_id INT,

name STRING,

email STRING,

address STRING,

created\_at TIMESTAMP,

updated\_at TIMESTAMP

);

-- Insert some sample data

INSERT INTO raw.customers (customer\_id, name, email, address, created\_at, updated\_at) VALUES

(1, 'John Doe', 'john.doe@example.com', '123 Elm St', CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP),

(2, 'Jane Smith', 'jane.smith@example.com', '456 Oak St', CURRENT\_TIMESTAMP, CURRENT\_TIMESTAMP);

**Step 2: Create the Snapshot Model**

1. Create a new file in the snapshots/ folder of your dbt project (e.g., snapshots/customers\_snapshot.sql).
2. Define the snapshot in this file.

-- snapshots/customers\_snapshot.sql

{% snapshot customers\_snapshot %}

{{

config(

target\_schema='your\_schema', -- Replace with your schema name

unique\_key='customer\_id', -- Column that uniquely identifies each record

strategy='timestamp', -- Use timestamp strategy for tracking changes

updated\_at='updated\_at' -- The column that tracks when a record was updated

)

}}

-- Select the data from your source table

SELECT

customer\_id,

name,

email,

address,

created\_at,

updated\_at

FROM {{ source('raw', 'customers') }}

{% endsnapshot %}

**Explanation of Key Parts:**

* **unique\_key='customer\_id'**: This key uniquely identifies each record. dbt will track changes for each customer based on this key.
* **strategy='timestamp'**: This tells dbt to use the updated\_at column to determine if a record has changed.
* **updated\_at='updated\_at'**: The column that tracks the last updated time for each record. If this value has changed, dbt will create a new record.
* **{{ source('raw', 'customers') }}**: This references the raw.customers table as the source of your data.

**Step 3: Run the Snapshot**

Once you’ve created the snapshot model, you can run the snapshot with the following command:

dbt snapshot --select customers\_snapshot

This will create a historical record of the customers table based on the snapshot logic. dbt will insert the first set of records into the snapshot table.

**Step 4: Verify the Snapshot in Snowflake**

After running the snapshot, check the resulting snapshot table in Snowflake:

SELECT \* FROM your\_schema.customers\_snapshot;

The output should look something like this:

| **customer\_id** | **name** | **email** | **address** | **created\_at** | **updated\_at** | **dbt\_valid\_from** | **dbt\_valid\_to** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | John Doe | john.doe@example.com | 123 Elm St | 2024-11-01 12:00:00 | 2024-11-01 12:00:00 | 2024-11-01 12:00:00 | NULL |
| 2 | Jane Smith | jane.smith@example.com | 456 Oak St | 2024-11-01 12:00:00 | 2024-11-01 12:00:00 | 2024-11-01 12:00:00 | NULL |

* **dbt\_valid\_from**: The timestamp when the record was first valid.
* **dbt\_valid\_to**: If the record has changed, the previous version will have an end timestamp (when it was replaced by the new version). If the record is the latest version, this will be NULL.

**Step 5: Modify Data and Test Snapshot**

Let’s simulate a change in the customers table by updating one of the records. For example:

UPDATE raw.customers

SET email = 'john.doe@newdomain.com', updated\_at = CURRENT\_TIMESTAMP

WHERE customer\_id = 1;

Run the snapshot again:

dbt snapshot --select customers\_snapshot

After running the snapshot again, you should see that the previous record for customer\_id = 1 will have a dbt\_valid\_to timestamp, and a new record will be added with the updated email.

**Summary**

* **Snapshot Materialization** allows you to capture changes over time and is particularly useful for slowly changing dimensions (SCD Type 2).
* **timestamp strategy** uses the updated\_at column to track changes and inserts a new record if the data has been updated.
* **The snapshot table** will store the historical data, with dbt\_valid\_from and dbt\_valid\_to columns to track when the record was valid.

This approach ensures that your data is always historical, with each change being tracked over time, making it ideal for use cases such as customer tracking, order history, and auditing.