



Creating Custom Materializations and Incremental Strategies in dbt (part 1)



Izzy OKonek · Follow

Published in phData · 13 min read · 23 hours ago



phData

Creating Custom Materializations and Incremental Strategies in dbt (part 1)

By Bruno Souza De Lima

This blog was written by Bruno Souza De Lima.



Check out Bruno's [LinkedIn](#) profile today!

dbt official documentation has great guides for both [custom materializations](#) and [incremental strategies](#). However, this blog aims to discuss them in more detail, with examples, tips, and adapter-specific comments. It is divided into two parts: custom materializations and the second about incremental strategies.

At the end of this blog series, you will feel comfortable creating your custom materialization or incremental strategy.

Do you Need a Custom Materialization or Incremental Strategy?

Custom materializations/incremental strategies are advanced topics in dbt and require skilled people to create and maintain them.

dbt version updates might have breaking changes to your custom materializations/incremental strategies, which can cause problems with your models and pipelines.

So, before creating them, double-check if you have the team capable of doing it and if you can't do what you need using the built-in dbt features.

What is a dbt Materialization?

Models in dbt are chunks of SQL (or Python) code materialized in your data platform. The way they are materialized is defined by Materialization. Materialization contains the DML code used to create or modify objects in your platform.

dbt has five types of model [materializations](#):

- View
- Table
- Ephemeral
- Incremental
- Materialized view

dbt also materializes for snapshots, seeds, tests, clones, and unit tests. However, this blog focuses on the model's materializations.

Materializations are defined in dbt with Jinja blocks, like macros. Optionally, you can define a specific adapter for the materialization (Snowflake, for instance) or leave it as 'default.' You can see the source code for materializations [here](#).

```
{% materialization my_materialization_name, adapter='snowflake' %}  
-- materialization code goes here  
{% endmaterialization %}
```

You can also override materializations just as macros, with a similar precedence logic:

1. global project — default
2. global project — plugin specific
3. imported package — default
4. imported package — plugin specific

5. local project — default
6. local project — plugin specific

Where dbt will choose the highest number option if available.

6 Steps to Build a Materialization

In the [official docs](#), the materialization code is divided into 6 parts:

1. Prepare the database for the new model
2. Run pre-hooks
3. Execute any SQL required to implement the desired materialization
4. Run post-model hooks
5. Clean up the database as required
6. Update the Relation cache

Let's walk through each one, but I will change the order slightly.

Executing SQL Code

I will start with this step because this is the heart of materialization: the SQL code it will generate and run. You can't have an object without the SQL code to create it.

Let's create our first materialization, called 'my_first_materialization.'
Since we are not specifying an adapter, we can leave it as the default.

We need to use the ‘call statement’ command to execute an SQL code with the SQL we want to run. In this example, I am creating a transient table.

```
{% materialization my_first_materialization, default %}

    {% set target_relation = this.incorporate(type='table') %}

    -- build model
    {% call statement('main') -%}
        create or replace transient table {{this.database}}.{{this.schema}}.{{this.identifier}}
        {{ sql }}
    {% endcall %}

    {{ return({'relations': [target_relation]}) }}

{% endmaterialization %}
```

The `{{ SQL }}` variable stores the compiled code of your model, and dbt assigns your compiled code to this variable automatically. The variable `{{ this }}` is a relation, and a relation is a Python object that contains info about the database, schema, and identifier, among other things. More specifically, `{{ this }}` is a relation that contains information about the model being run.

At the end of the materialization block, we need to return the materialization’s target relation with its type. The target relation is not necessarily the same as the model’s relation. For instance, we can change the identifier of our model in our database.

Since we need to return a target relation with a type, we can use the ‘incorporate’ method. It adds an attribute to the relation, in this case, the type.

That's all, and we created a custom materialization!! Congrats!

Now, you can configure your model as:

```
{{ config(
    materialized='my_first_materialization'
) }}

select 1 as id
```

And the model is running fine!

...

We've built a materialization that creates a table, but dbt already has this capability, which has a lot more complexity and handles different possible use cases for a table. So, let's use dbt's built-in macros to get their improved functionality.

dbt has several macros to create objects.

- get_create_table_as_sql
- get_create_view_as_sql
- get_create_materialized_view_as_sql
- get_incremental_merge_sql
- get_incremental_append_sql
- get_incremental_delete_insert_sql

- And more!

Let's use the 'get_create_table_as_sql' macro. It requires 3 arguments:

- Temporary: A boolean indicating if the table we want to create is temporary.
- Relation: A relation containing the database, schema, and model identifier.
- SQL: The compiled code of the model.

```
{% materialization my_first_materialization, default %}  
  
  {%- set existing_relation = load_cached_relation(this) -%}  
  {%- set target_relation = this.incorporate(type='table') %}  
  
  -- build model  
  {% call statement('main') -%}  
    {{ get_create_table_as_sql(False, existing_relation, sql) }}  
  {%- endcall %}  
  
  {{ return({'relations': [target_relation]}) }}  
  
{% endmaterialization %}
```

We used the 'load_cached_relation' to get the relation of our model, which is needed for the 'get_create_table_as_sql' macro. The latter returns an SQL code to be run by the 'call statement' command.

You can create your own SQL logic for your materialization or overwrite the default ones. See how Snowflake [overwrites the table creation macro](#) to adapt to Snowflake specifications.

Taking one of these files is a good starting point for your SQL logic. In the end, your macro needs to return an SQL code. Additionally, dbt has several macros to help you with building your logic.

Running pre and post-hooks

This part is quite simple. Many times, people will want to add pre- and post-hooks to your model. If you don't know what they are, check out this doc page.

Adding them to your materialization is easy. You have to add the following macros before your SQL execution

- {{ run_hooks(pre_hooks, inside_transaction=False) }}
- {{ run_hooks(pre_hooks, inside_transaction=True) }}

And this macro after your SQL execution

- {{ run_hooks(post_hooks, inside_transaction=True) }}
- {{ run_hooks(post_hooks, inside_transaction=False) }}

The ones with `inside_transaction=True` will run inside the transaction. And the ones with `inside_transaction=False` will run before the transaction BEGIN or after the transaction COMMIT.

Note that transactions are only supported for Postgres and Redshift adapters. If you are using another adapter, you don't need to care about transaction processing, allowing you to use this instead

- {{ run_hooks(pre_hooks) }}
- {{ run_hooks(post_hooks) }}

See the [snowflake implementation of the table materialization](#) for an example.

For this post, we are considering a case where you might need to split hooks based on transactions. In this case, we also want to add the macro {{ adapter.commit() }}, to commit the transactions.

So, our materialization will look like this:

```
{% materialization my_first_materialization, default %}

{% set existing_relation = load_cached_relation(this) -%}
{% set target_relation = this.incorporate(type='table') %}

{{ run_hooks(pre_hooks, inside_transaction=False) }}

-- `BEGIN` happens here:
{{ run_hooks(pre_hooks, inside_transaction=True) }}

-- build model
{% call statement('main') -%}
    {{ get_create_table_as_sql(False, existing_relation, sql) }}
{% endcall %}

{{ run_hooks(post_hooks, inside_transaction=True) }}

-- `COMMIT` happens here
{{ adapter.commit() }}

{{ run_hooks(post_hooks, inside_transaction=False) }}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}
```

Preparing the Database

Preparing the database means knowing its state and avoiding conflicts or problems when our materialization runs.

Using intermediate and backup relations in materializations is common to avoid messing up the target relation, or we can create a backup. So, in the preparation phase, ensuring that you have the appropriate relations is good. If your adapter does not support transactions, this becomes less important, as you will see at the end of the blog with the snowflake implementation.

During the previous setup, you may have noticed:

```
{%- set existing_relation = load_cached_relation(this) -%}
```

```
{%- set target_relation = this.incorporate(type='table') %}
```

The macros from this step help us prepare the database and collect information about the existing relationship.

Let's add intermediate and backup relations as we already set our existing and target relations. For that, we will use the macros

- [make_intermediate_relation](#)
- [make_backup_relation](#)

And to drop relations if they exist (remembering that we don't want the intermediate and backup relations to already exist to avoid conflicts), we can use the macro

- drop_relation_if_exists

So now our materialization looks like this:

```
{% materialization my_first_materialization, default %}

{% set existing_relation = load_cached_relation(this) -%}
{% set target_relation = this.incorporate(type='table') %}
{% set intermediate_relation = make_intermediate_relation(target_relation) -%}
-- the intermediate_relation should not already exist in the database; get_relation
-- will return None in that case. Otherwise, we get a relation that we can drop
-- later, before we try to use this name for the current operation
{% set preexisting_intermediate_relation = load_cached_relation(intermediate_relation) -%}

{% set backup_relation_type = 'table' if existing_relation is none else existing_relation.type %}
{% set backup_relation = make_backup_relation(target_relation, backup_relation_type) %}
-- as above, the backup_relation should not already exist
{% set preexisting_backup_relation = load_cached_relation(backup_relation) -%}

-- drop the temp relations if they exist already in the database
{{ drop_relation_if_exists(preexisting_intermediate_relation) }}
{{ drop_relation_if_exists(preexisting_backup_relation) }}

{{ run_hooks(pre_hooks, inside_transaction=False) }}

-- `BEGIN` happens here:
{{ run_hooks(pre_hooks, inside_transaction=True) }}

-- build model
{% call statement('main') -%}
    {{ get_create_table_as_sql(False, intermediate_relation, sql) }}
{% endcall %}

{{ run_hooks(post_hooks, inside_transaction=True) }}

-- `COMMIT` happens here
{{ adapter.commit() }}

{{ run_hooks(post_hooks, inside_transaction=False) }}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}
```

Note that we now use the intermediate relation as the argument for the `get_create_table_as_sql` macro. This helps us avoid dropping an existing relation if rendering the table is an issue.

However, the intermediate relation has a suffix like `'__dbt_temp.'` So if my model is called `'my_first_model,'` the intermediate relation will be called `'my_first_model__dbt_temp.'`

We don't want to permanently create a model named `'my_first_model__dbt_temp'` in our database, so we can use the [`adapter.rename_relation`](#) method to rename it to our `target_relation` identifier if it runs successfully.

Additionally, we should create a backup relation before committing to this new table. If, for any reason, the transaction is not committed, we still have our table and relation. This is accomplished by using the same rename relation function.

Now, our materialization looks like:

```
{% materialization my_first_materialization, default %}

{%- set existing_relation = load_cached_relation(this) -%}
{%- set target_relation = this.incorporate(type='table') -%}
{%- set intermediate_relation = make_intermediate_relation(target_relation) -%}
-- the intermediate_relation should not already exist in the database; get_relation
-- will return None in that case. Otherwise, we get a relation that we can drop
-- later, before we try to use this name for the current operation
{%- set preexisting_intermediate_relation = load_cached_relation(intermediate_rel

{%- set backup_relation_type = 'table' if existing_relation is none else existing
{%- set backup_relation = make_backup_relation(target_relation, backup_relation_t
-- as above, the backup_relation should not already exist
{%- set preexisting_backup_relation = load_cached_relation(backup_relation) -%}
```

```

-- drop the temp relations if they exist already in the database
{{ drop_relation_if_exists(preexisting_intermediate_relation) }}
{{ drop_relation_if_exists(preexisting_backup_relation) }}

{{ run_hooks(pre_hooks, inside_transaction=False) }}

-- `BEGIN` happens here:
{{ run_hooks(pre_hooks, inside_transaction=True) }}

-- build model
{% call statement('main') -%}
    {{ get_create_table_as_sql(False, existing_relation, sql) }}
{% endcall %}

-- cleanup
{% if existing_relation is not none %}
    /* Do the equivalent of rename_if_exists. 'existing_relation' could have been
       since the variable was first set. */
    {% set existing_relation = load_cached_relation(existing_relation) %}
    {% if existing_relation is not none %}
        {{ adapter.rename_relation(existing_relation, backup_relation) }}
    {% endif %}
{% endif %}

{{ adapter.rename_relation(intermediate_relation, target_relation) }}

{{ run_hooks(post_hooks, inside_transaction=True) }}

-- `COMMIT` happens here
{{ adapter.commit() }}

{{ run_hooks(post_hooks, inside_transaction=False) }}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}

```

Two more things before the next step: grants and persistent docs.

If we have permissions on our table that we want to keep even if a new object is generated, then we need to get the configured grants of our models with:

```
{% set grant_config = config.get('grants') %}
```

Note that this `config.get()` function can be used to retrieve any model configuration.

Now, we can use the `should_revoke` and `apply_grants` functions. The first will check if any grants should be passed to the new object, and the latter will apply the grants, if any.

To persist our documentation in the database ([check this link](#) for more information), we can use the `persist_docs` macro.

Let's have another look at our materialization.

```
{% materialization my_first_materialization, default %}

{% set existing_relation = load_cached_relation(this) -%}
{% set target_relation = this.incorporate(type='table') %}
{% set intermediate_relation = make_intermediate_relation(target_relation) -%}
-- the intermediate_relation should not already exist in the database; get_relation
-- will return None in that case. Otherwise, we get a relation that we can drop
-- later, before we try to use this name for the current operation
{% set preexisting_intermediate_relation = load_cached_relation(intermediate_relation) %}

{% set backup_relation_type = 'table' if existing_relation is none else existing_relation.type %}
{% set backup_relation = make_backup_relation(target_relation, backup_relation_type) %}
-- as above, the backup_relation should not already exist
{% set preexisting_backup_relation = load_cached_relation(backup_relation) -%}
-- grab current tables grants config for comparison later on
{% set grant_config = config.get('grants') %}

-- drop the temp relations if they exist already in the database
{{ drop_relation_if_exists(preexisting_intermediate_relation) }}
{{ drop_relation_if_exists(preexisting_backup_relation) }}

{{ run_hooks(pre_hooks, inside_transaction=False) }}

-- `BEGIN` happens here:
```

```

{{ run_hooks(pre_hooks, inside_transaction=True) }}

-- build model
{% call statement('main') -%}
    {{ get_create_table_as_sql(False, existing_relation, sql) }}
{% endcall %}

-- cleanup
{% if existing_relation is not none %}
    /* Do the equivalent of rename_if_exists. 'existing_relation' could have been
       since the variable was first set. */
    {% set existing_relation = load_cached_relation(existing_relation) %}
    {% if existing_relation is not none %}
        {{ adapter.rename_relation(existing_relation, backup_relation) }}
    {% endif %}
{% endif %}

{{ adapter.rename_relation(intermediate_relation, target_relation) }}

{{ run_hooks(post_hooks, inside_transaction=True) }}

{% set should_revoke = should_revoke(existing_relation, full_refresh_mode=True) %}
{% do apply_grants(target_relation, grant_config, should_revoke=should_revoke) %}

{% do persist_docs(target_relation, model) %}

    -- `COMMIT` happens here
    {{ adapter.commit() }}

{{ run_hooks(post_hooks, inside_transaction=False) }}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}

```

Clean up Database

After our new model is in our database, we just need to clean it up, basically dropping everything we don't want anymore. In this case, we can drop the backup_relation (after the transaction COMMIT). We will use the same drop_relation_if_exists macro.

```

{% materialization my_first_materialization, default %}

{%- set existing_relation = load_cached_relation(this) -%}
{%- set target_relation = this.incorporate(type='table') %}
{%- set intermediate_relation = make_intermediate_relation(target_relation) -%}
-- the intermediate_relation should not already exist in the database; get_relati
-- will return None in that case. Otherwise, we get a relation that we can drop
-- later, before we try to use this name for the current operation
{%- set preexisting_intermediate_relation = load_cached_relation(intermediate_rel

{%- set backup_relation_type = 'table' if existing_relation is none else existing
{%- set backup_relation = make_backup_relation(target_relation, backup_relation_t
-- as above, the backup_relation should not already exist
{%- set preexisting_backup_relation = load_cached_relation(backup_relation) -%}
-- grab current tables grants config for comparision later on
{% set grant_config = config.get('grants') %}

-- drop the temp relations if they exist already in the database
{{ drop_relation_if_exists(preexisting_intermediate_relation) }}
{{ drop_relation_if_exists(preexisting_backup_relation) }}

{{ run_hooks(pre_hooks, inside_transaction=False) }}

-- `BEGIN` happens here:
{{ run_hooks(pre_hooks, inside_transaction=True) }}

-- build model
{% call statement('main') -%}
    {{ get_create_table_as_sql(False, existing_relation, sql) }}
{%- endcall %}

-- cleanup
{% if existing_relation is not none %}
    /* Do the equivalent of rename_if_exists. 'existing_relation' could have been
       since the variable was first set. */
    {% set existing_relation = load_cached_relation(existing_relation) %}
    {% if existing_relation is not none %}
        {{ adapter.rename_relation(existing_relation, backup_relation) }}
    {% endif %}
{% endif %}

{{ adapter.rename_relation(intermediate_relation, target_relation) }}

{{ run_hooks(post_hooks, inside_transaction=True) }}

{% set should_revoke = should_revoke(existing_relation, full_refresh_mode=True) %}
{% do apply_grants(target_relation, grant_config, should_revoke=should_revoke) %}

```



```

{% do persist_docs(target_relation, model) %}

-- `COMMIT` happens here
{{ adapter.commit() }}

-- finally, drop the existing/backup relation after the commit
{{ drop_relation_if_exists(backup_relation) }}

{{ run_hooks(post_hooks, inside_transaction=False) }}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}

```

Updating the Relation Cache

Here, we must tell dbt everything we modified in the relations cache. If your materialization returns one relation (our case), we can return it at the end.

We did this in the first step with the following line `{{ return({'relations': [target_relation]}) }}`.

But if you dropped or renamed some other relation, then there are some other steps we should consider:

- If you use `adapter.drop_relation` and `adapter.rename_relation` to drop and rename your intermediate relations. You are fine and don't need to do anything else.
- If you didn't use these methods, you must synchronize the cache by calling `adapter.cache_dropped` and `adapter.cache_renamed` methods.

We made it!

If you didn't notice, we just finished building the dbt's default table materialization. Congrats!!

You can use this as a starting point to build your custom materialization.

The biggest modification you will need to make will be in the executed SQL code. For inspiration, you can look at other implementations, such as the snowflake's table materialization and table creation macro, and use several macros dbt has to help you with the logic.

Materialization Configurations

Your custom materialization may require some extra configuration, like with incremental models. For example, when you use a merge incremental strategy, you need to configure a unique key.

```
{{ config(
    materialized='incremental'
    , incremental_strategy='merge'
    , unique_key='user_id'
) }}
```

These configurations are treated the same way as model configurations, and you can use them in your materialization logic.

To retrieve them, you can use the macro:

- `config.get('your_config')`

You can define a default value for the:

- `config.get('your_config', default='default_value')`

If you want to make a configuration mandatory, you can use the macro:

- `config.require('your_config')`

Analyzing an Adapter Specific Materialization

We have built the default implementation of table materialization. But let's examine the custom table materialization of a specific adapter to see how it differs.

This is the table materialization implementation for the snowflake adapter.

```
{% materialization table, adapter='snowflake', supported_languages=['sql', 'python']

  {% set original_query_tag = set_query_tag() %}

  {%- set identifier = model['alias'] -%}
  {%- set language = model['language'] -%}

  {% set grant_config = config.get('grants') %}

  {%- set old_relation = adapter.get_relation(database=database, schema=schema, ident
  {%- set target_relation = api.Relation.create(identifier=identifier,
                                                schema=schema,
                                                database=database, type='table') -%}

  {{ run_hooks(pre_hooks) }}

  {#-- Drop the relation if it was a view to "convert" it in a table. This may lead t
  -- downtime, but it should be a relatively infrequent occurrence #}
  {% if old_relation is not none and not old_relation.is_table %}
    {{ log("Dropping relation " ~ old_relation ~ " because it is of type " ~ old_rela
    {{ drop_relation_if_exists(old_relation) }}
  {% endif %}

  {% call statement('main', language=language) -%}
    {{ create_table_as(False, target_relation, compiled_code, language) }}
```

```

{% - endcall %}

{{ run_hooks(post_hooks) }}

{% set should_revoke = should_revoke(old_relation, full_refresh_mode=True) %}
{% do apply_grants(target_relation, grant_config, should_revoke=should_revoke) %}

{% do persist_docs(target_relation, model) %}

{% do unset_query_tag(original_query_tag) %}

{{ return({'relations': [target_relation]}) }}

{% endmaterialization %}

```

What we can note is:

- In the materialization block initialization, we have the definition of the adapter and supported languages. {% materialization table, adapter='snowflake', supported_languages=['sql', 'python'] %}
- As Snowflake does not support transactions, there is only one macro for pre_hooks and one for post_hooks. There is also no need to commit a transaction.
- Due to the lack of transaction support, it does not add an intermediate or backup relation, which means you don't have to drop them.
- There is logic for the query_tag, which is a specific snowflake configuration.
- To generate SQL code, It calls a macro called create_table_as, which calls this macro here.

In summary, the snowflake custom table materialization is the default implementation with some simplification (you can remove all the

transaction-related logic) and some additions (for instance, the logic for the query tag). It is even more readable than the default version.

The same will happen to other adapters and other materializations/macros.

Closing Thoughts

Thanks for reading this far! Custom materializations require skilled people to create and maintain them, and I hope you now have the skills to work on your materialization in dbt.

In part two, we will explore custom incremental strategies. If you feel comfortable with materializations, incremental strategies will be a walk in the park. See you there.

If your organization wants to succeed with dbt, phData would love to help! As dbts' 2023 Partner of the Year, our experts will ensure your dbt instance becomes a powerful organizational transformation tool.

[Explore phData's award-winning dbt Services](#)



Written by Izzy OKonek

0 Followers · Editor for phData

Follow



More from Izzy OKonek and phData



Claire Sasse in phData

How To Tame Apache Impala Users With Admission Control

In order to effectively manage resources for Apache Impala, we recommend using the...

Dec 6, 2019



Claire Sasse in phData

Log Aggregation, Search, And Alerting On CDH With Pulse

Pulse is an application log aggregation, search, and alert framework that runs on...

Dec 17, 2019



 Claire Sasse in phData

Implementing Metadata As Part Of Data Management

Data centralization without careful metadata implementation is like stocking a warehouse...

Dec 17, 2019



 Claire Sasse in phData

Data Science Enablement: The First Of Its Kind

Here at phData, we are proud to announce our newest service offering in the Big Data space...


Nov 26, 2019



See all from Izzy OKonek

See all from phData

Recommended from Medium


 Abhay Parashar in The Pythoneers

17 Mindblowing Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance

★ 5d ago 🖱️ 2.6K 💬 20



 Piotr in ITNEXT

10 Essential Kubernetes Tools You Didn't Know You Needed

Celebrating 10th Kubernetes Anniversary 🎉

Jul 2 🖱️ 273 💬 3



Lists



Staff Picks

689 stories · 1143 saves



Self-Improvement 101

20 stories · 2320 saves




Stories to Help You Level-Up at Work


19 stories · 694 saves



Productivity 101

20 stories · 2047 saves

 Alexander Nguyen in Level Up Coding

 Zach Qui... in Pipeline: Your Data Engineering Reso...

The resume that got a software engineer a \$300,000 job at Google.

1-page. Well-formatted.



Jun 1



13.1K



178



Go From Staging Table To Production Data With 1 Stupid...

A risk-averse approach to “flipping the switch” from test tables to production tables...



Jul 9



24



1



Tari Ibaba in Coding Beauty

10 essential tips to supercharge VS Code and code faster (0 to 100)

95% of developers are just wasting VS Code's potential.



Jul 8



456



5



Avi Siegel in Management Matters

Stop Working— You're a Manager Now

Escape your IC mindset



Jul 9



1K



18



See more recommendations