Developer guide of dbt

This comprehensive developer guide is tailored for beginners, providing an accessible resource that empowers data developers to quickly engage with and commence their work using dbt.

- Megha Goyate

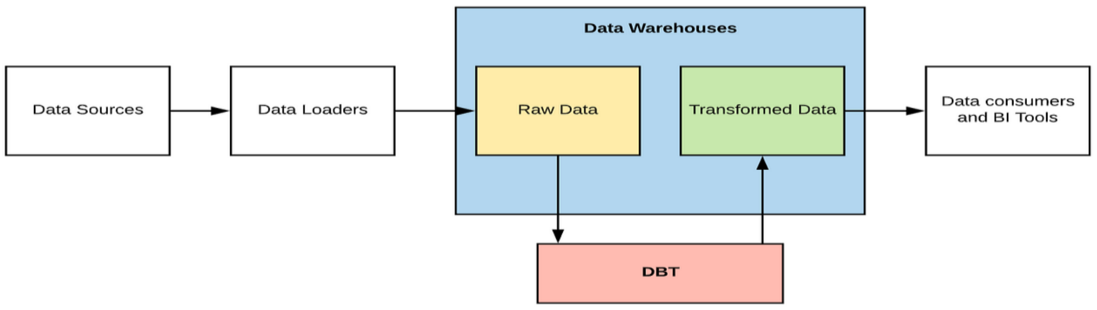
Introduction

What is dbt?

Dbt is a transformation workflow that helps you get more work done while producing higher quality results. You can use dbt to modularize and centralize your analytics code, while also providing your data team with guardrails typically found in software engineering workflows. Collaborate on data models, version them, and test and document your queries before safely deploying them to production, with monitoring and visibility. In addition to generating SQL code, dbt also provides several features that make it easier to work with data. These features include the ability to manage dependencies between data models, run tests to ensure data integrity, and track the lineage of data to understand how it has been transformed over time

Multiple databases are supported, including:

- Postgres
- Redshift
- Big Query
- Snowflake



With dbt, data analysts take ownership of the entire analytics engineering workflow from writing data transformation code all the way through to deployment and documentation—as well as to becoming better able to promote a data-driven culture within the organization. They can:

1. Quickly and easily provide clean, transformed data ready for analysis: dbt enables data analysts to custom-write transformations through SQL SELECT statements. There is no need to write boilerplate code. This makes data transformation accessible for analysts that don't have extensive experience in other programming languages.

2. Build reusable and modular code using Jinja: dbt (data build tool) allows you to establish macros and integrate other functions outside of SQL's capabilities for advanced use cases. Macros in Jinja are pieces of code that can be used multiple times. Instead of starting at the raw data with every analysis, analysts instead build up reusable data models that can be referenced in subsequent work.

3. Maintain data documentation and definitions within dbt as they build and develop lineage graphs: Data documentation is accessible, easily updated, and allows you to deliver trusted data across the organization. dbt (data build tool) automatically generates documentation around descriptions, models dependencies, model SQL, sources, and tests. dbt creates lineage graphs of the data pipeline, providing transparency and visibility into what the data is describing, how it was produced, as well as how it maps to business logic.

4.Apply software engineering practices—such as modular code, version control, testing, and continuous integration/continuous deployment (CI/CD)—to analytics code: Continuous integration means less time testing and quicker time to development, especially with dbt Cloud. You don't need to push an entire repository when there are necessary changes to deploy, but rather just the components that change. You can test all the changes that have been made before deploying your code into production. dbt Cloud also has integration with GitHub for automation of your continuous integration pipelines, so you won't need to manage your own orchestration, which simplifies the process.

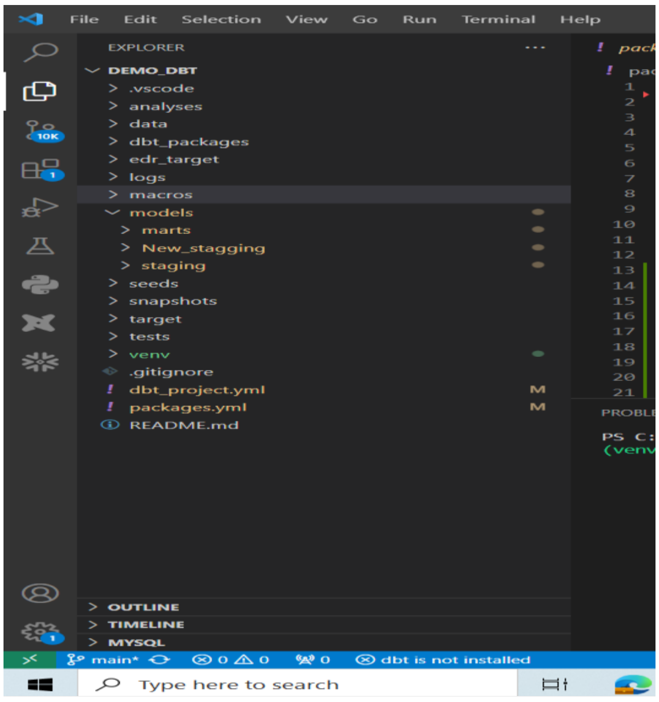
5.Perform automated testing: dbt (data build tool) comes prebuilt with unique, not null, referential integrity, and accepted value testing. Additionally, you can write your own custom tests using a combination of Jinja and SQL. To apply any test on a given column, you simply reference it under the same YAML file used for documentation for a given table or schema. This makes testing data integrity an almost effortless process.

Key Terminologies

- **Model: A** DBT model is a fundamental unit of transformation. It represents a SQL query that transforms raw data from a source into a structured, analytically useful format. Models define how data is transformed, cleaned, and organized
- **Source**: Sources are configurations in DBT that specify how to connect to external data, such as databases or files. They provide the raw data that you work with in your models. Sources define where your data comes from.
- **Seeds**: Seeds are DBT models that represent raw or reference data. They serve as a starting point for your modelling process and can contain static or slowly changing data that doesn't require frequent updates.
- **Jinja templates**: Jinja templates are a powerful feature used for parameterization and dynamic SQL generation within your SQL queries and DBT models. Jinja is a templating engine that allows you to embed dynamic expressions, variables, and logic directly into your SQL code. This makes it easier to create flexible and reusable transformations.
- **Snapshot**: Snapshots in DBT are a way to create historical versions of your models. They capture the state of a model at a specific point in time, allowing you to track changes and analyse historical data.
- **Test**: Tests in DBT are SQL-based assertions that you define to validate the quality and accuracy of your transformed data. Common tests include checking for null values, uniqueness, or specific data ranges.
- **Documentation**: DBT automatically generates documentation for your models, columns, and tests. This documentation is a valuable resource for users, providing descriptions and explanations of the data and transformations.
- **Ref**: The **ref** function in DBT allows you to reference other models within your SQL queries. It creates a dependency relationship, enabling you to build complex transformations by combining different models.

- **Hook**: Hooks in DBT are custom SQL code snippets that can be executed before or after a DBT model is built. They are often used for data preprocessing, running data quality checks, or other custom tasks.
- **Package**: DBT packages are reusable collections of models, tests, macros, and other DBT assets. They allow you to package and share code across projects and organizations, promoting code reuse and best practices.
- **Materialization**: Materialization determines how the results of a DBT model are stored in the target data warehouse. Common materialization types include **table** (creating physical tables), **view** (creating database views), **ephemeral** (temporary results) and **incremental.**
- **Snapshot Strategy**: Snapshot strategies in DBT define how historical snapshots are generated and managed for your models. Strategies can be time-based (e.g., daily snapshots) or version-based (e.g., snapshots triggered by data changes).
- **Selectors**: Selectors are used to filter which models DBT should build during a run. They help control the scope of transformations, allowing you to build specific models based on criteria like tags or names.
- **Run**: Running DBT involves executing the transformations defined in your project. You can use the **dbt run** command to build or update your models, applying the defined SQL transformations to your data.
- **Warehouse**: The data warehouse is the target database where DBT models are created and stored. DBT supports various data warehouses like Snowflake, Big Query, Redshift, and more, making it versatile for different environments.
- **Profiles.yml** : profiles.yml is a configuration file that contains information about your data warehouse connection. It includes database credentials, connection details, and other settings necessary for DBT to interact with your data warehouse.
- **Target**: In DBT, a target refers to a specific database schema where DBT models will be created and stored. It allows you to isolate your DBT project's work from other databases and schemas in your data warehouse.

Introductory screenshot:



Installation

System requirements

To install dbt on Windows, you will need:

- Python 3.8 or higher
- Git for Windows
- Visual Studio Code version 1.54 or higher

To install dbt on Ubuntu, you will need:

- Python 3.8 or higher
- Visual Studio Code version 1.54 or higher

Step-by-step installation guide

Python installation guide :

Step 1: Check if Python is Installed

- First, check if Python is already installed on your system. Open a terminal or command prompt and run the following command:

```
python --version
```

- If Python is installed, it will display the version number (e.g., Python 3.8.5). If it's not installed or you have an older version, proceed to the next steps.

Step 2: Download Python

- Visit the official Python website to download the latest version of Python:
- Python Downloads: https://www.python.org/downloads/

Step 3: Download the Installer

- Choose the appropriate installer for your operating system (Windows, macOS, or Linux).
- Ensure you download the latest version of Python 3.x.

Step 4: Run the Installer

For Windows:

- Double-click the downloaded executable file (e.g., python-3.x.x.exe) to run the installer.
- Check the box that says "Add Python x.x to PATH" during installation. This is important to make Python accessible from the command line.

Vs code installation guide:

- Visit the Visual Studio Code website: Go to the official Visual Studio Code website at https://code.visualstudio.com/.
- Download the installer: Click on the "Download for Windows" button to download the Windows installer.
- Run the installer: Locate the downloaded installer file (usually in your "Downloads" folder) and double-click it to run the installation.
- Installation wizard: Follow the installation wizard's instructions. You can choose the installation location and options during the installation process.
- Complete the installation: After the installation is complete, you can launch Visual Studio Code by finding it in your Start menu or using the desktop shortcut

Pre-requisites

- Snowflake account URL
- Database created in snowflake
- Data warehouse created in snowflake
- Schema created in above database
- User, roles created in Snowflake with appropriate privileges across above database (refer to appendix for sample script)
- Familiarity with SQL queries
- Familiarity with advanced SQL functionalities such as common table expression (CTE)
- Raw dataset available in Snowflake (refer to appendix)
- Expected fact and dimension table structure (refer to appendix)

Getting Started

Project creation

Concepts

To understand the concept of creating a dbt (Data Build Tool) project is to break it down into key components and steps:

1. **project Initialization**: Start by creating a new directory that will serve as your dbt project's root. This directory will house all the configuration and SQL files required for your data transformation project.
2. **Configuration File (dbt_project.yml)**: The dbt_project.yml file is the heart of your dbt project. It contains essential project-level configurations, including:
   - **Name and Version**: Give your project a name and version.
   - **Data Warehouse Connection**: Specify the target data warehouse, including the database name and credentials.
   - **Seeds**: Define seed files or tables used as raw data sources.
   - **Models Directory**: Set the path to the directory where your model files are located.
   - **Profiles.yml**: Reference your data warehouse connection details stored in a **profiles.yml** file.
3. **Models Directory**: This is where you define your individual model files. Each model is a SQL query that transforms data from your source (often raw data) into a desired format. Model files usually have a .sql extension.
4. **Seeds**: If you have static data that doesn't change frequently, you can include seed files in the **data** directory. Seed files contain the initial data that will be used in your transformations.
5. **Macros Directory**: Macros are reusable pieces of SQL code that can be used in your model files. You can create and store macros in a macros directory within your project.
6. **Analysis Directory**: The analysis directory is where you can store ad-hoc SQL queries, analytics scripts, or exploratory data analysis related to your project.
7. **Dependency Management (packages.yml)**: If you rely on external packages, custom macros, or SQL functions, you can specify them in the packages.yml file. dbt will manage these dependencies.
8. **Defining Models**: In your model files, write SQL code to define how raw data should be transformed. You can create relationships between models to specify their execution order. For example, you might create a model for customers and another for orders, with the customer model being a dependency for the order model.
9. **Testing**: dbt encourages the practice of writing tests to ensure data quality. You can define assertions and expectations for your models in separate .sql files within a tests directory.
10. **Documentation**: You can add descriptions, metadata, and tags to your models and columns. This documentation makes it easier for team members to understand the purpose and usage of your data transformation code.
11. **Version Control**: To collaborate effectively and track changes in your dbt project, use a version control system like Git. This enables multiple team members to work on the project simultaneously, manage branches, and merge changes.
12. **Execution and Deployment**: You can execute your dbt project using the dbt run command. It will execute your models in the specified order and load the transformed data into the target data warehouse. For deployment to production or other environments, you can use the dbt deploy command.

Execution

1. **Install DBT**

   Once Python is installed, you can use Pip to install DBT:

   a. Open a terminal or command prompt.

   b. Run the following command to install DBT:

   ```
   Pip install dbt
   ```

c. Install dbt Core and the dbt Snowflake adapter:

```
- pip install  dbt-snowflake
```

d. Verify the installation by running:

```
- dbt  --version
```

2. **Initialize a DBT Project**
   a. Open VS Code and create a new directory for your DBT project or navigate to an existing project directory.
   b. In the terminal within VS Code, navigate to your project directory.
   c. Initialize a new DBT project:

```
-  dbt init my_project
```

   d. Replace **my_project** with your preferred project name.

3. **Configure Snowflake Connection**
   a. Inside your project directory, locate the **profiles.yml** file that was generated during the project initialization.
   b. Open **profiles.yml** with a text editor within VS Code. Add your Snowflake connection details under the appropriate profile. Here's an example of a Snowflake profile configuration.

```
my-snowflake-db:
 target: dev
 outputs:
  dev:
   type: snowflake
   account: [account id]
   # User/password auth
   user: [username]
   password: [password]
   authenticator: username_password_mfa
   role: [user role]
   database: [database name]
   warehouse: [warehouse name]
   schema: [dbt schema]
```

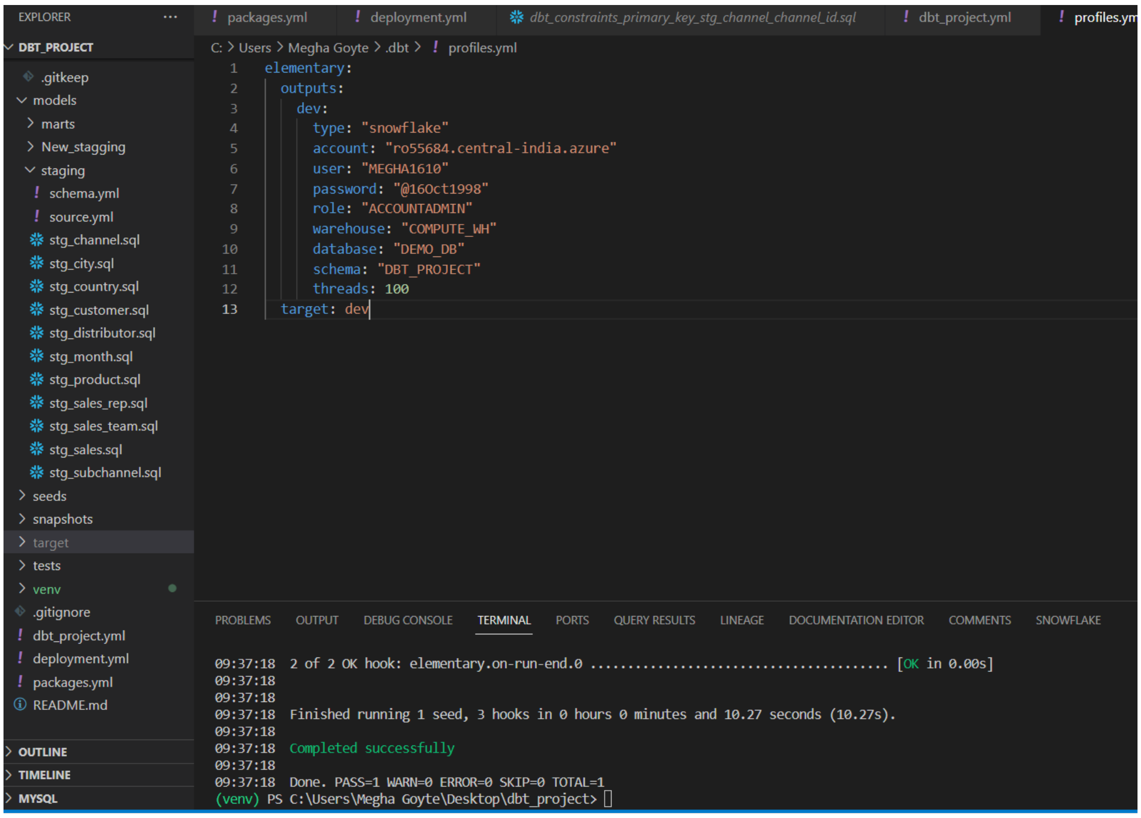
Make sure to replace your_account, your_warehouse, your_database, your_schema, your_username, and your_password with your Snowflake account details

4. **Create DBT Models**
    a. In VS Code, navigate to the **models** directory in your DBT project.
    b. Create your DBT models by creating SQL files within this directory. For example, you can create a file named **my_model.sql** and define your model's SQL transformations in that file

5. **Run DBT commands**
    a. You've now created a DBT project in VS Code. You can start developing your DBT models by adding SQL files to the **models** directory within your project.
    b. You can run DBT commands (e.g., **dbt run**, **dbt test**) from the VS Code terminal to build and test your models.
    c. Monitor the output in the terminal for any errors or issues. DBT will create tables or views in Snowflake based on your model definitions.

Refer this link
https://github.com/DBT-Snowflake-project/dbt_project.yml

Outcome:
A folder structure will be created as shown in the below screenshot.



```yaml
elementary:
  outputs:
    dev:
      type: "snowflake"
      account: "ro55684.central-india.azure"
      user: "MEGHA1610"
      password: "@16Oct1998"
      role: "ACCOUNTADMIN"
      warehouse: "COMPUTE_WH"
      database: "DEMO_DB"
      schema: "DBT_PROJECT"
      threads: 100
  target: dev
```

```
09:37:18  2 of 2 OK hook: elementary.on-run-end.0 ...................................... [OK in 0.00s]
09:37:18
09:37:18
09:37:18  Finished running 1 seed, 3 hooks in 0 hours 0 minutes and 10.27 seconds (10.27s).
09:37:18
09:37:18  Completed successfully
09:37:18
09:37:18  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
(venv) PS C:\Users\Megha Goyte\Desktop\dbt_project>
```

dbt pipeline

Concepts

Model Configurations in dbt_project.yml:

The dbt_project.yml file allows you to configure model settings for your entire project. Here are some common configurations:

- +materialized: Specifies the materialization type (e.g., table, view, incremental, ephemeral) for a model.
- description: Provides a description or documentation for the model.
- tags: Assigns one or more tags to the model for categorization.
- column_types: Specifies data types for columns in a model.
- unique_key: Defines a unique key for a model.

Example:

```
# dbt_project.yml

models:

  my_model:

    +materialized: table

    description: This is a table model.

    tags: [t1]
```

You can refer to project.yml file from the link https://github.com/meghagoyate/DBT-Snowflake-project

Schema.yml file :

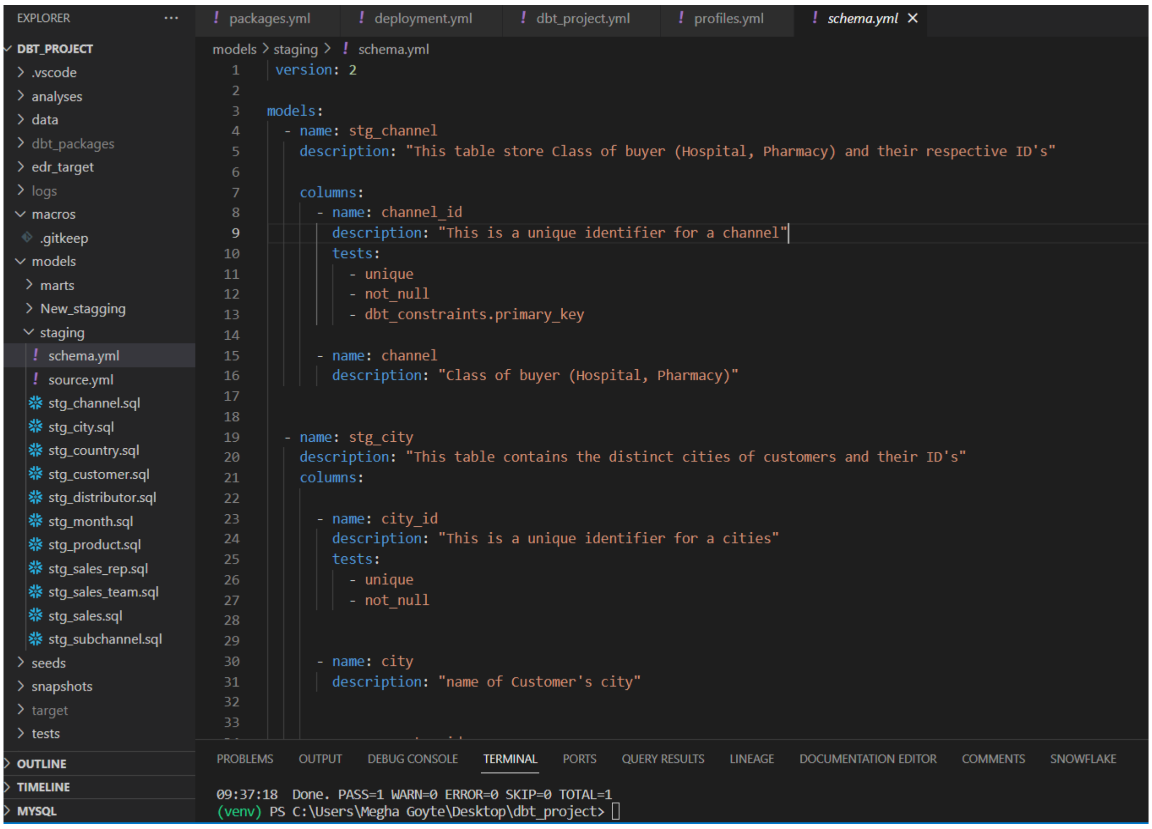
The schema.yml file in a dbt project is a YAML file that defines the schema of the data models in the project. It contains information about the columns in each model, their data types, and any constraints on the data. The schema.yml file is used by dbt to generate the SQL code that creates and maintains the data models in the project.

schema.yml file should include the following information:

- The name of the data model that the schema.yml file defines.
- A brief description of the data model, including what kind of data it contains and how it is used.
- A list of the columns in the data model, along with their data types and any constraints on the data.
- Any other relevant information about the data model, such as foreign key relationships or indexes.

You can refer schema.yml file https://github.com/meghagoyate/models/staging/schema.yml

Example: Here's an example that defines both sources and models for the project



```yaml
version: 2

models:
  - name: stg_channel
    description: "This table store Class of buyer (Hospital, Pharmacy) and their respective ID's"

    columns:
      - name: channel_id
        description: "This is a unique identifier for a channel"
        tests:
          - unique
          - not_null
          - dbt_constraints.primary_key

      - name: channel
        description: "Class of buyer (Hospital, Pharmacy)"


  - name: stg_city
    description: "This table contains the distinct cities of customers and their ID's"
    columns:

      - name: city_id
        description: "This is a unique identifier for a cities"
        tests:
          - unique
          - not_null


      - name: city
        description: "name of Customer's city"

```

```
09:37:18  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
(venv) PS C:\Users\Megha Goyte\Desktop\dbt_project>
```

What is dbt macro?

A dbt macro, within the dbt framework, works like a function in traditional programming. It allows users to encapsulate specific, often repetitive, SQL logic into a named, callable unit. Just as you'd use a function in programming to avoid writing the same code repeatedly, a dbt macro lets you define a particular SQL operation once and then call upon that operation wherever you need it in your dbt project. By leveraging macros, analysts and data engineers can maintain cleaner, more modular, and consistent codebases in their data transformation workflows. Think of it to give SQL supercharged, function-like capabilities within the realm of dbt!

Jinja placeholders in dbt macros:

Syntax: {{...}}

Anything enclosed between double curly braces is evaluated and replaced with its value. In dbt, the most common use of this syntax is for the functions ref() and source(). In the example below, the users model is referenced. (The complete list of such Jinja functions is available at https://docs.getdbt.com/reference/dbt-jinja-functions).

Staging : stg_channel.sql

```
with channel as (
   select * from {{ ref('pharma_with_id') }}
),
renamed as (
   select
         distinct channel_id,
          channel

   from channel
)
select * from renamed
order by channel_id asc
```

The ref('pharma_with_id') is a dbt function that gets replaced with the actual reference to the view or table that is defined in the users model of your dbt project.

Compiled code: When the above code gets compiled, it looks like this (placeholder {{ ref('pharma_with_id') }} replaced by the actual table name DEMO_DB.DBT_PROJECT.pharma_with_id:

```
with channel as (
    select * from
DEMO_DB.DBT_PROJECT.pharma_with_id
),
renamed as (
    select
            distinct channel_id,
             channel
        from channel
)
select * from renamed
```

Staging in model folder:

The staging area is where you ingest, clean, and prepare raw data from source systems before it undergoes more complex transformations. The primary objective is to ensure that the incoming data is of good quality and conforms to a consistent structure. Staging acts as a buffer between raw source data and the data mart to ensure data quality and integrity.

It is used for:

- Ingesting data from source databases, APIs, or files.
- Basic data cleaning, validation, and standardization (e.g., handling null values, data type conversions).
- Retaining an untouched copy of the source data for reference and audit purposes.

You can refer staging folder DBT-Snowflake-project/models/staging at main · meghagoyate/DBT-Snowflake-project (github.com)

marts in model folder:

Data marts are where you store structured, transformed, and business-ready data for analytics and reporting. These tables are optimized for efficient querying and are often denormalized or aggregated to make them more user-friendly and performant. Data marts represent a clean, organized view of the data that's well-suited for data analysts and business users. We can create models in marts folder with taking the staging as source in marts and create fact and dimension tables

You can refer staging folder https://github.com/meghagoyate/DBT-Snowflake-project/tree/main/models/marts

It is used for:

- Aggregating data for reporting (e.g., monthly sales totals by region).
- Applying business-specific calculations (e.g., profitability metrics).
- Joining data from multiple sources for comprehensive analysis.
- Creating denormalized or star-schema data structures to facilitate queries and analytics.

Execution

Dbt run and compile commands :

dbt run is a simple command used to run SQL SELECT statements in the models using a desired materialization strategy. **dbt run** command will execute all the models in the project. You can even use the **dbt run** command with flags like **–full-refresh** to treat incremental models as table models. It is used when the schema has changed for the incremental model, and you want to recreate the transformation.

```
dbt run --full-refresh
```

dbt run –m

The **–m** flag is used to specify which models to run. It is used to avoid running all the models.

```
dbt run --m stg_distributor
```

With the help of above command we can run specific models instead of all the models in the dbt project.

dbt run-tags

Tags are set to describe or add metadata to models. These tags can be later used as a part of resource selection syntax to filter models while using the **dbt run** command. We can mention tags in our dbt_project.yml file

```
dbt run --select tag:t1
```

This command will run all the models that have been tagged t1.

Dbt compile command:

dbt compile generates executable SQL from source model, test, and analysis files. You can find these compiled SQL files in the target/ directory of your dbt project.

The compile command is useful for:

- Visually inspecting the compiled output of model files. This is useful for validating complex jinja logic or macro usage.
- Manually running compiled SQL. While debugging a model or schema test, it's often useful to execute the underlying select statement to find the source of the bug.

Interactive compile:
Starting in dbt v1.5, compile can be "interactive" in the CLI, by displaying the compiled code of a node or arbitrary dbt-SQL query:

- --select a specific node *by name*
- --inline an arbitrary dbt-SQL query

This will log the compiled SQL to the terminal, in addition to writing to the target/ directory.

For example:

```
dbt compile --select stg_city
```

```
dbt compile --inline "select * from {{ ref('pharma_with_id') }}"
```

The command accesses the data platform to cache-related metadata, and to run introspective queries

Important commands:

dbt docs command helps you generate your project documentation. It has two subcommands—generate and serve.

```
dbt docs
```

dbt docs generate
This command allows you to generate your project's documentation website. It is done in three steps:

- Copy the website **index.html** file into the **target/** directory.
- Compile the project to **target/manifest.json**.
- Produce the **target/catalog.json** file. The file contains metadata about the views and tables produced by models.

You can use the **–no-compile flag** with **dbt docs generate** to skip the second step from the above steps.

```
dbt docs generate
```

dbt docs serve
The command will create a documentation website locally. You can access the web server on port 8080. But you can specify different ports using the **–port** flag. The web server will be located in the **target/** directory. However, run **dbt docs generate** before **dbt docs serve** since the **docs server** relies on catalog metadata artifact that the **server command** depends upon.

```
dbt docs serve
```

dbt seed

dbt seeds are CSV files referenced in the downstream models using the **ref** function. However, you must first add the data to the data warehouse from your locally hosted project. This is where the **dbt seed** command helps you with loading data. Once the data is loaded, it can be referenced while building models. Consider a CSV file named country_codes is located at **seeds/country_codes.csv**. You can load your CSV file into the data warehouse by executing the dbt seed command.

Outcome

Models in a dbt project are SQL scripts that define the data models that will be created in your data warehouse. They can be used to transform and clean data from your sources, create new tables and views, and define materialized views and incremental models they are typically written in SQL, but they can also include Jinja templates. Jinja templates allow you to use variables and logic in your models, which can be useful for generating dynamic SQL code or for reusing common code snippets.
Example:

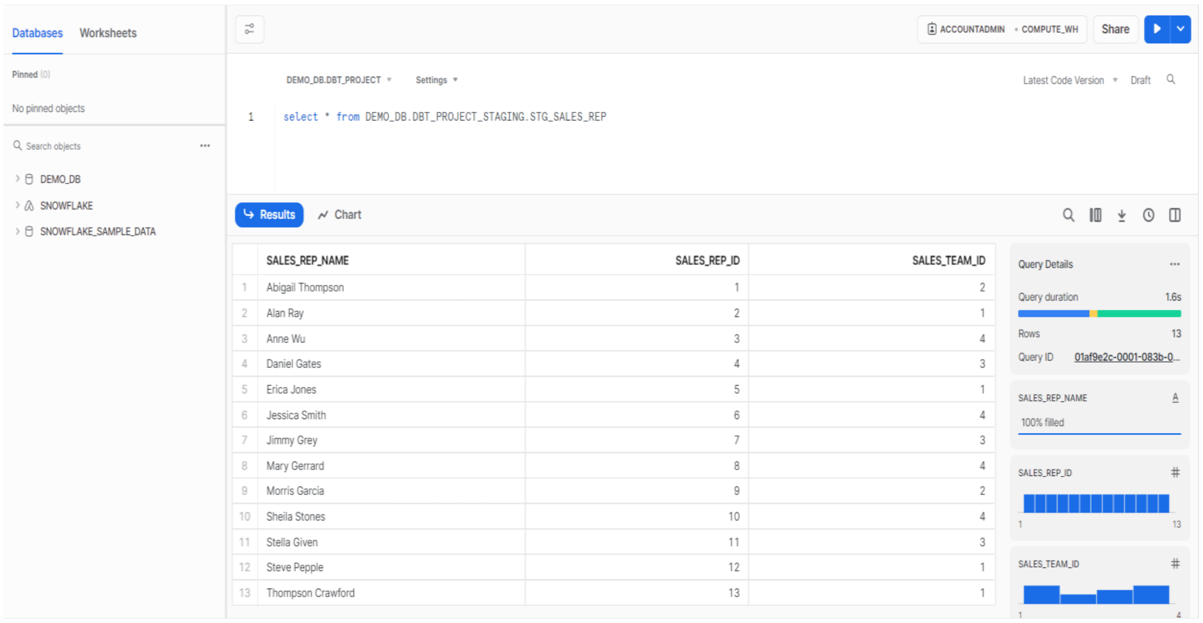
Marts : Dim_sales_rep.sql

```
select

    stg_sales_rep.sales_rep_id,

    stg_sales_rep.sales_rep_name,

    stg_sales_team.sales_team_id

from {{ ref('stg_sales_rep') }}

left join {{ ref('stg_sales_team') }} on stg_sales_rep.sales_team_id = stg_sales_team.sales_team_id
```

And the compile code in target folder of above code will look like :

```
select

    stg_sales_rep.sales_rep_id,

    stg_sales_rep.sales_rep_name,

    stg_sales_team.sales_team_id

from DEMO_DB.DBT_PROJECT_staging.stg_sales_rep

left join DEMO_DB.DBT_PROJECT_staging.stg_sales_team on stg_sales_rep.sales_team_id =
stg_sales_team.sales_team_id
```

After the dbt run command we can see the table stg_sales_rep in our data warehouse it will be look like



```
select * from DEMO_DB.DBT_PROJECT_STAGING.STG_SALES_REP
```

	SALES_REP_NAME	SALES_REP_ID	SALES_TEAM_ID
1	Abigail Thompson	1	2
2	Alan Ray	2	1
3	Anne Wu	3	4
4	Daniel Gates	4	3
5	Erica Jones	5	1
6	Jessica Smith	6	4
7	Jimmy Grey	7	3
8	Mary Gerrard	8	4
9	Morris Garcia	9	2
10	Sheila Stones	10	4
11	Stella Given	11	3
12	Steve Pepple	12	1
13	Thompson Crawford	13	1

Test & constraints

Concepts

Tests and constraints in dbt projects are used to ensure the quality and integrity of your data.Tests are used to validate the data in your models. They can be used to check for things like null values, duplicate values, and values that are outside of a certain range.Constraints are used to enforce rules on the data in your models. They can be used to prevent things like null values from being inserted into a column or to ensure that a column contains unique values.

Tests and constraints can be defined in the schema.yml file for each model. They can also be defined in separate files and referenced in the schema.yml file. Tests and constraints are an important part of any dbt project. By using tests and constraints, you can help to ensure that your data is accurate, complete, and consistent. List of the different types of tests and constraints that are available in dbt:

Tests:

- not_null: Asserts that a column is not null.
- unique: Asserts that a column is unique.
- referential_integrity: Asserts that the values in a column have a valid relationship with the values in another column.
- accepted_values: Asserts that the values in a column are within a set of accepted values.
- relationship: Asserts that the values in a column have a valid relationship with the values in another table.
- expression: Asserts that a SQL expression evaluates to true.

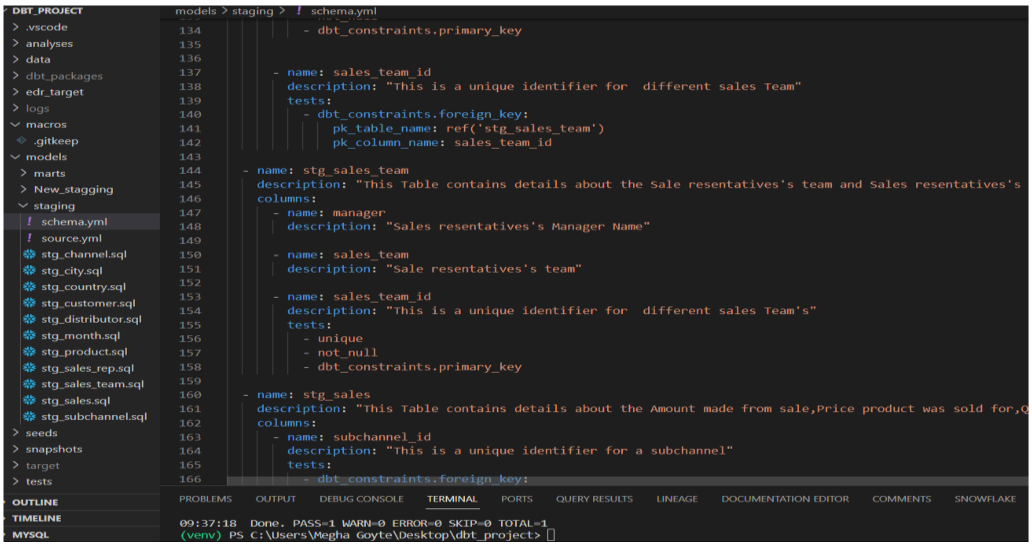
Constraints:

- primary_key: Ensures that a column contains unique values and that the column is not null.
- unique: Ensures that a column contains unique values.
- foreign_key: Enforces a relationship between two columns in different tables.
- not_null: Ensures that a column is not null
- DBT-Snowflake-project/models/staging/schema.yml /DBT-Snowflake-project

Execution

You can provide constraint and test lists in the constraints and tests sections of the schema.yml file for each model in your dbt project.

In the following schema.yml file we can see the tests and constraints are mentioned



```yaml
          - dbt_constraints.primary_key


      - name: sales_team_id
        description: "This is a unique identifier for  different sales Team"
        tests:
          - dbt_constraints.foreign_key:
              pk_table_name: ref('stg_sales_team')
              pk_column_name: sales_team_id

  - name: stg_sales_team
    description: "This Table contains details about the Sale resentatives's team and Sales resentatives's
    columns:
      - name: manager
        description: "Sales resentatives's Manager Name"

      - name: sales_team
        description: "Sale resentatives's team"

      - name: sales_team_id
        description: "This is a unique identifier for  different sales Team's"
        tests:
          - unique
          - not_null
          - dbt_constraints.primary_key

  - name: stg_sales
    description: "This Table contains details about the Amount made from sale,Price product was sold for,Q
    columns:
      - name: subchannel_id
        description: "This is a unique identifier for a subchannel"
        tests:
          - dbt_constraints.foreign_key:
```

```
09:37:18  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
(venv) PS C:\Users\Megha Goyte\Desktop\dbt_project>
```

command to test constraints :

To test constraints and tests in a dbt project, you can use the dbt test command.The dbt test command runs all of the tests and constraints that are defined for your models. If any of the tests or constraints fail, the command will print an error message and the test or constraint will not succeed.To run the dbt test command, simply navigate to the root directory of your dbt project and run the following command:

```
dbt test
```

This will run all of the tests and constraints for all of the models in your project.If you only want to run the tests and constraints for a specific model, you can use the --select flag. For example, to run the tests and constraints for the stg_city model, you would run the following command:

```
dbt test --select stg_city
```

Outcome

When you define constraints in the schema.yml file of a dbt project, they will be translated to SQL queries when dbt runs the models. The specific SQL queries that are generated will depend on the type of constraints that you have defined.For example, if you define a unique constraint on a column, dbt will generate a SQL query that creates a unique index on the column. If you define a foreign key constraint, dbt will generate a SQL query that creates a foreign key relationship between the column and the referenced column.Here are some examples of how constraints are translated to SQL queries in schema.yml file :

```
version: 2

models:

 - name: stg_channel

   description: "This table store Class of buyer (Hospital, Pharmacy) and their respective ID's"columns:

     - name: channel_id

       description: "This is a unique identifier for a channel"

       tests:

         - unique

         - not_null

         - dbt_constraints.primary_key
```

This will generate the following compiled code in target :

Below code generate for unique channel_id

```
select

    channel_id as unique_field,

    count(*) as n_records

from DEMO_DB.DBT_PROJECT_staging.stg_channel

where channel_id is not null

group by channel_id

having count(*) > 1
```

Below code generate for not_null channel_id

```
select channel_id

from
DEMO_DB.DBT_PROJECT_staging.stg_channel

where channel_id is null
```

We have mentioned primary key constarint in the schema.yml file so the compiled code for primary key will look like below:

```
with validation_errors as (

    select

        channel_id, count(*) as row_count

    from DEMO_DB.DBT_PROJECT_staging.stg_channel

    group by channel_id

    having count(*) > 1

        or channel_id is null


)

select *

from validation_errors
```

Incremental

Concepts

Incremental & Full Refresh:

Incremental refresh in dbt allows you to update your data models with new data without having to rebuild them from scratch. This can be much faster and more efficient than a full refresh, especially for large data models. To use incremental refresh, you need to define your models in a way that allows dbt to track changes to the underlying data. This can be done by using the keyword "incremental" in the schema.yml file or we can define it in models also.

When you run a dbt model with incremental refresh enabled, dbt will first check to see if there are any new rows in the underlying data. If there are new rows, dbt will insert them into the target table. If there are any existing rows that have changed, dbt will update them in the target table.

Full refresh in dbt rebuilds the target table from scratch. This can be necessary if the underlying data has changed significantly, or if the structure of the target table has changed.

To run a full refresh, you can use the --full-refresh flag with the dbt run command. For example, to run a full refresh of the stg_city model, you would run the following command:

```
dbt run --full-refresh stg_city
```

When to use incremental refresh vs full refresh

Incremental refresh is generally the preferred way to refresh your data models. It is faster and more efficient than a full refresh, and it can help to reduce the risk of data loss. However, there are some cases where you may need to run a full refresh. For example, you may need to run a full refresh if the underlying data has changed significantly, or if the structure of the target table has changed.

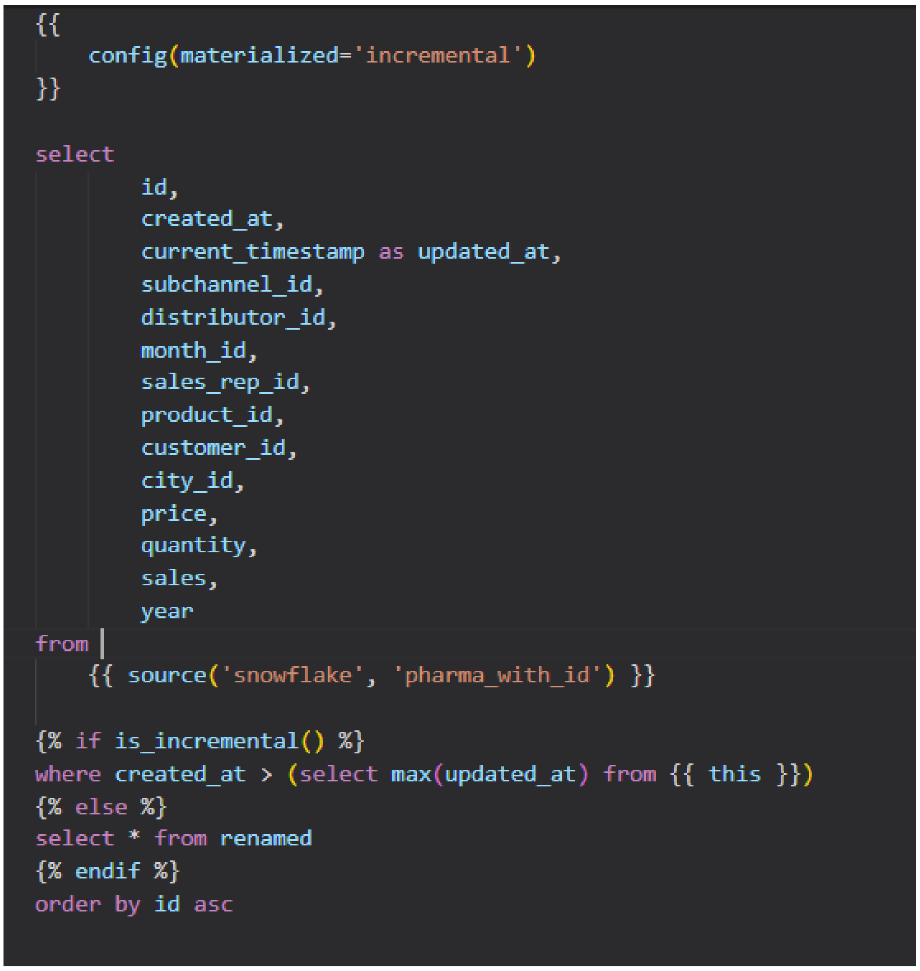
You may also want to run a full refresh periodically to ensure that the target table is consistent with the underlying data.

Execution

To do execution as incremental in dbt, you need to define your models in a way that allows dbt to track changes to the underlying data. This can be done by using the incremental keyword in the models or schema.yml file.

GitHub link for incremental materialization
https://github.com/meghagoyate/DBT-Snowflake-project/tree/main/models/marts

For example, the following model definition uses the materialized as incremental to enable incremental refresh:



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

select
        id,
        created_at,
        current_timestamp as updated_at,
        subchannel_id,
        distributor_id,
        month_id,
        sales_rep_id,
        product_id,
        customer_id,
        city_id,
        price,
        quantity,
        sales,
        year
from
    {{ source('snowflake', 'pharma_with_id') }}

{% if is_incremental() %}
where created_at > (select max(updated_at) from {{ this }})
{% else %}
select * from renamed
{% endif %}
order by id asc
```

This model definition tells dbt that the stg_sales model can be incrementally refreshed by tracking changes to the updated_at column.

Once you have defined your models in a way that allows for incremental refresh, you can then run them using the dbt run command. When you run a model with incremental refresh enabled, dbt will first check to see if there are any new rows in the underlying data. If there are new rows, dbt will insert them into the target table. If there are any existing rows that have changed, dbt will update them in the target table.

For incremental refresh in dbt, you can use the following commands:

```
# Run all incremental models in your project

dbt run

# Run a specific incremental model

dbt run <model_name>

# Run all incremental models in a specific directory

dbt run --select <directory_path>

# Run all incremental models with a specific tag

dbt run --filter <tag>
```

You can also use the --full-refresh flag to run a full refresh of a model. This will rebuild the target table from scratch.

Outcome

When you use incremental materialized views in dbt, the following changes may occur in the resulting fact and dimension tables:

- New rows may be inserted into the fact and dimension tables when new data is available in the source tables.
- Existing rows in the fact and dimension tables may be updated when the data in the source tables changes.
- Rows may be deleted from the fact and dimension tables if they are no longer valid.

The specific changes that occur will depend on the type of incremental materialized view that you are using and the data that is being processed.

Materializations

Concepts

Materializations are strategies for persisting dbt models in a warehouse. There are five types of materializations built into dbt. They are:

- table
- view
- incremental
- Ephemeral

Materialization types

View

When using the view materialization, your model is rebuilt as a view on each run, via a "create view as" statement.

- **Pros:** No additional data is stored, views on top of source data will always have the latest records in them.
- **Cons:** Views that perform a significant transformation, or are stacked on top of other views, are slow to query.
- **Advice:**
  Generally start with views for your models, and only change to another materialization when you're noticing performance problems.
  Views are best suited for models that do not do significant transformation, e.g. renaming, recasting columns.

Table

When using the table materialization, your model is rebuilt as a table on each run, via a create table as statement.

- **Pros:** Tables are fast to query
- **Cons:**
  Tables can take a long time to rebuild, especially for complex transformations
  New records in underlying source data are not automatically added to the table
- **Advice:**
  Use the table materialization for any models being queried by BI tools, to give your end user a faster experience
  Also use the table materialization for any slower transformations that are used by many downstream models

Incremental

incremental models allow dbt to insert or update records into a table since the last time that model was run.

- **Pros:** You can significantly reduce the build time by just transforming new records
- **Cons:** Incremental models require extra configuration and are an advanced usage of dbt. Read more about using incremental models here.
- **Advice:**
  Incremental models are best for event-style data
  Use incremental models when your dbt runs are becoming too slow (i.e. don't start with incremental models)

Ephemeral

ephemeral models are not directly built into the database. Instead, dbt will interpolate the code from this model into dependent models as a common table expression.

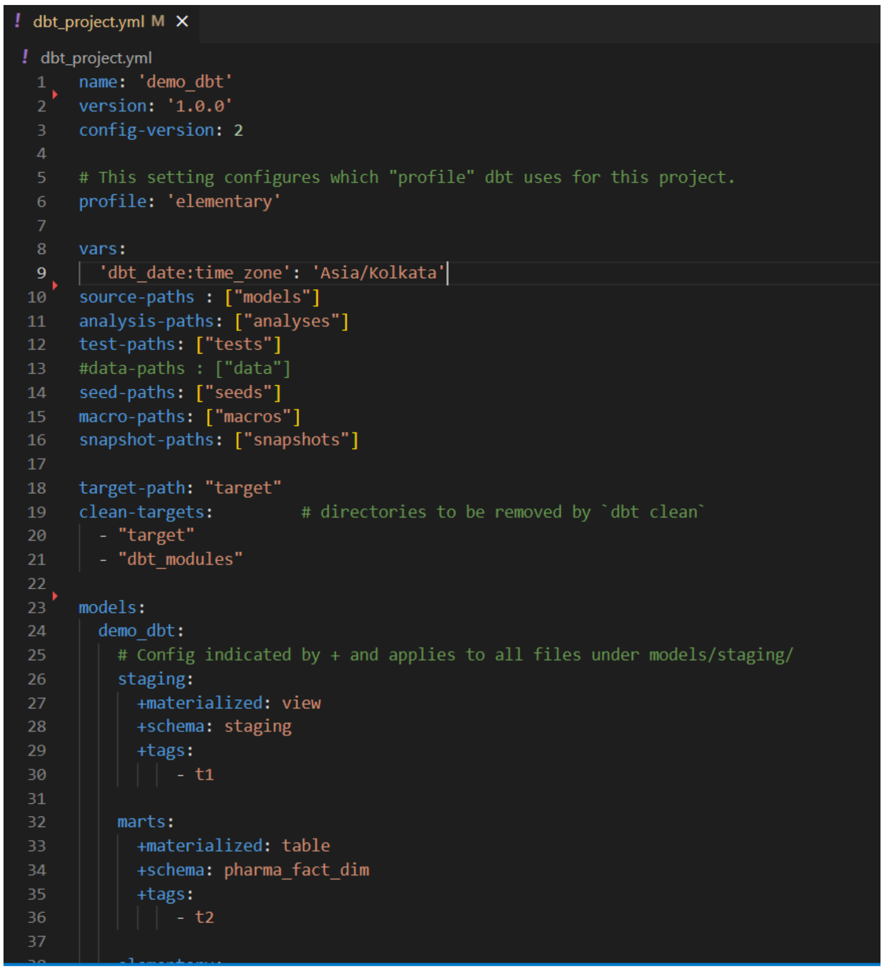
- **Pros:**
  You can still write reusable logic

Ephemeral models can help keep your data warehouse clean by reducing clutter (also consider splitting your models across multiple schemas by using custom schemas).

- **Cons:**
  You cannot select directly from this model.
  Operations (e.g. macros called via dbt run-operation cannot ref() ephemeral nodes)
  Overuse of ephemeral materialization can also make queries harder to debug.
- **Advice:** Use the ephemeral materialization for:
  very light-weight transformations that are early on in your DAG
  are only used in one or two downstream models, and
  do not need to be queried directly
- Check ephemeral materialization https://github.com/meghagoyate/DBT-Snowflake-project/blob/main/models/New_stagging/pharma_with_id.sql

Configuring materializations

By default, dbt models are materialized as "views". Models can be configured with a different materialization by supplying the configuration parameter “materialized” as shown below. (The following example shows all the models under “staging” being materialized as “view” and all models under “marts” being materialized as “table”



```yaml
name: 'demo_dbt'
version: '1.0.0'
config-version: 2

# This setting configures which "profile" dbt uses for this project.
profile: 'elementary'

vars:
  'dbt_date:time_zone': 'Asia/Kolkata'
source-paths : ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
#data-paths : ["data"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

target-path: "target"
clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_modules"

models:
  demo_dbt:
    # Config indicated by + and applies to all files under models/staging/
    staging:
      +materialized: view
      +schema: staging
      +tags:
        - t1

    marts:
      +materialized: table
      +schema: pharma_fact_dim
      +tags:
        - t2
```

Execution

Table Materialization:

- **Purpose**: The most common materialization type, this stores the model's output as a table in your target database. It's suitable for creating structured, readily tables for reporting and analysis.
- **Execution**:
- In your DBT model file, set the **materialized** parameter to **'table'**.
- Example in a DBT model file:

```
{{ config(
    materialized='table'
) }}
SELECT
  ...
```

View Materialization:

- **Purpose**: This creates a database view for the model instead of materializing it as a physical table. It's useful for creating virtual models without consuming additional storage space.
- **Execution**:
- In your DBT model file, set the **materialized** parameter to **'view'**.
- Example in a DBT model file:

```
{{ config(
    materialized='view'
) }}
SELECT
  ...
```

Incremental Materialization:

- **Purpose**: This materialization type is used for incremental updates. It stores a partial representation of the model's data and updates it incrementally with new data, reducing processing time and resource usage.
- **Execution**:
- In your DBT model file, set the **materialized** parameter to **'incremental'**.
- Specify the **unique_key** and **target_schema** parameters to identify how new data is added to the existing materialization.
- Example in a DBT model file:

```
{{ config(
    materialized='incremental',
    unique_key='id',
    target_schema='staging'
) }}
SELECT
```

Ephemeral Materialization:

- **Purpose**: This is a temporary materialization type that doesn't store data in the target database. It's useful for creating temporary results within a DBT run and can be referenced in subsequent queries.
- **Execution**:
- In your DBT model file, set the **materialized** parameter to **'ephemeral'**.
- Example in a DBT model file:

```
{{ config(
    materialized='ephemeral'
) }}
SELECT
  ...
```

To execute these materializations in your DBT project, you would typically run the **dbt run** command, which will execute the SQL statements in your models based on their configured materialization types. You can also use other DBT commands, like **dbt seed**, **dbt snapshot**, and **dbt test**, to perform various tasks related to data transformation and testing within your project.

Outcome

- If we use **materialized as Table** physical table is created in the target database, and the results of the model are stored as records in this table. This table can be queried like any other table in the database, making it suitable for reporting and analytics.
- The **view** is a virtual representation of the model, which does not store the data itself but instead references the source data. Views are useful for creating virtual models without consuming additional storage space
- Similar to the Table Materialization, an **incremental materialization** creates a physical table in the target database. However, it stores data in such a way that allows for incremental updates. New data is appended to the existing data based on the specified **unique_key**. This type is useful for efficiently updating data without reprocessing the entire dataset.
- **Ephemeral materializations** do not create any physical storage. They are temporary results that exist only for the duration of the current DBT run. Ephemeral models are not stored in

the database but can be referenced in subsequent queries within the same DBT run. These are useful for intermediate computations or one-off transformations.

Observability

Introduction

Data observability is the ability to understand, diagnose, and manage data health across multiple IT tools throughout the data lifecycle. A data observability platform helps organizations to discover, triage, and resolve real-time data issues using telemetry data like logs, metrics, and traces. Observability goes beyond monitoring by allowing organizations to improve security by tracking data movement across disparate applications, servers, and tools. With data observability, companies can streamline business data monitoring and manage the internal health of their IT systems by reviewing outputs.

The five pillars of data observability are:

1. Freshness
2. Distribution
3. Volume
4. Schema
5. Lineage

Freshness:

Concept

Freshness is how recently the data is updated and how often it should be updated. For dbt, the source freshness can be checked as part of the data test.

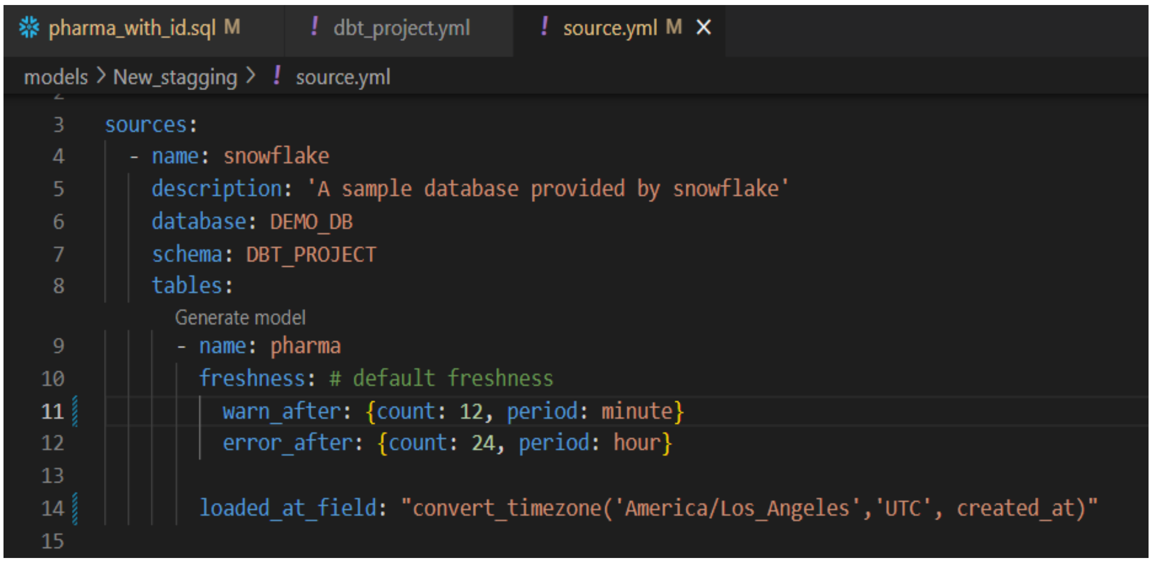
Approach 1: using dbt source freshness

The freshness can be specified to a particular column as *loaded_at_field* in the schema.yml. The threshold of the number of periods and time can be specified along with any filters to limit the data for performance.

In the freshness block, one or both of warn_after and error_after can be provided. If neither is provided, then dbt will not calculate freshness snapshots for the tables in this source.

- **warn_after** specifies the number of minutes/hours/days that can pass before you will get a warning when running your dbt test. This warning won't error out but will let you know to investigate the freshness of the table before it errors out.
- **error_after** specifies the number of days that can pass before you get a failure.
- the **loaded_at_field** is required to calculate freshness for a table. If a loaded_at_field is not provided, then dbt will not calculate freshness for the table.
- Example: If any source table hasn't been updated as per the specified freshness configuration, dbt will raise a warning or error. In our example, if the data is more than 12 hours old, we'll get a warning, since the warn_after threshold was set to 12 hours.

We can define freshness in source.yml file as follows.

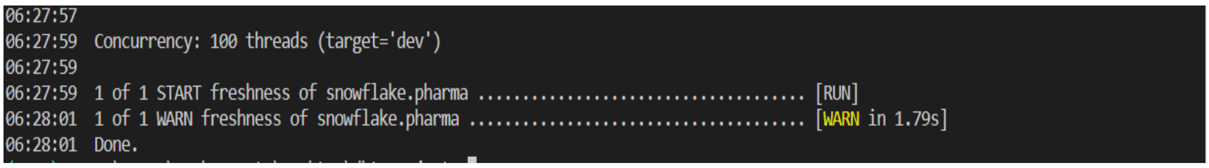


```yaml
sources:
  - name: snowflake
    description: 'A sample database provided by snowflake'
    database: DEMO_DB
    schema: DBT_PROJECT
    tables:
      - name: pharma
        freshness: # default freshness
          warn_after: {count: 12, period: minute}
          error_after: {count: 24, period: hour}

        loaded_at_field: "convert_timezone('America/Los_Angeles','UTC', created_at)"
```

The command to be run after this is

```
dbt source freshness
```

After running the above command it will give warning or error if the freshness condition is not satisfied it will gives result as follows

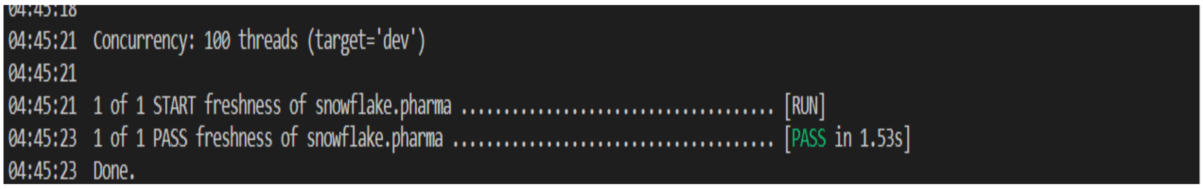


```
06:27:57
06:27:59  Concurrency: 100 threads (target='dev')
06:27:59
06:27:59  1 of 1 START freshness of snowflake.pharma .................................... [RUN]
06:28:01  1 of 1 WARN freshness of snowflake.pharma ..................................... [WARN in 1.79s]
06:28:01  Done.
```

Above screen-shot shows the freshness results as "WARN". Considering the definition "freshness" block in source.yml, the value of loaded_at_field is evaluated and since it is more than 12 minutes old, the result is considered as WARN. Had this value been more than 24 hours old, we would have got the results as 'ERROR".

The corrective action will be to ingest the fresh data.

(In this example, only for the illustration purpose, let's change the freshness block values. Once the freshness condition is satisfied, we will get result as pass instead of error or warning as follows.)



```
04:45:18
04:45:21  Concurrency: 100 threads (target='dev')
04:45:21
04:45:21  1 of 1 START freshness of snowflake.pharma ................................... [RUN]
04:45:23  1 of 1 PASS freshness of snowflake.pharma ..................................... [PASS in 1.53s]
04:45:23  Done.
```

Approach 2: dbt_expectations

Installation

dbt version required: >=1.2.0, <2.0.0

Include the following in your packages.yml file:

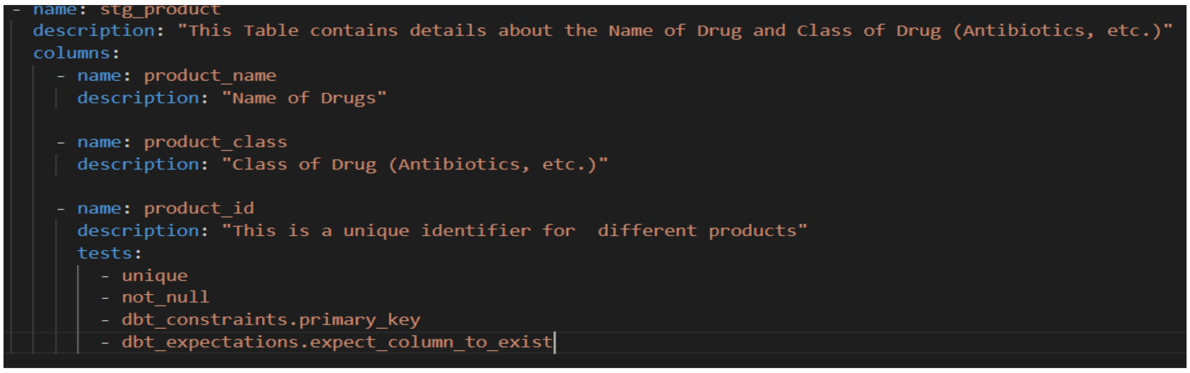
```
packages:

  - package: calogica/dbt_expectations

    version: 0.10.0
```

To install latest version of package please refer package Hub

Run **dbt deps** to install the package.

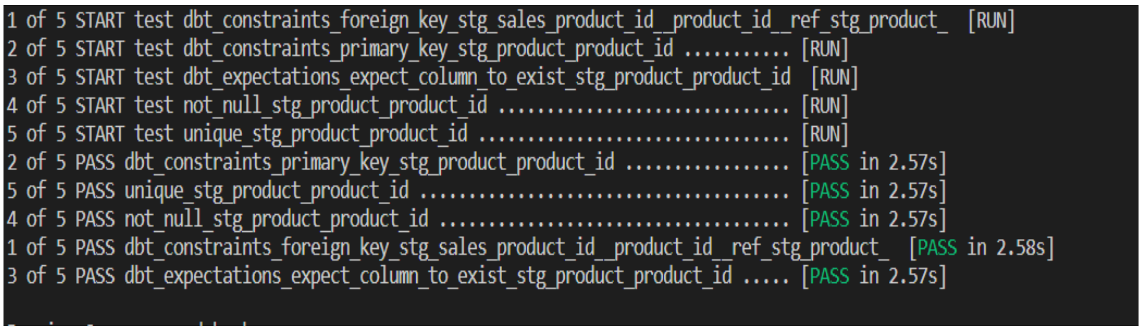
"dbt-expectations" is an extension package for dbt, whose intent is to allow dbt users to deploy GE-like tests in their data warehouse directly from dbt, vs having to add another integration with their data warehouse. For further executions refer dbt_expectations

Dbt expectations provide so many tests and it can be defined in the schema.yml file and we can define appropriate tests according to our need it can be applied to columns where we generally define the generic tests and after that if we run command **dbt run** the test will be check here in test block dbt expectation test id defined as **expect_column_to_exist**



```
- name: stg_product
  description: "This Table contains details about the Name of Drug and Class of Drug (Antibiotics, etc.)"
  columns:
    - name: product_name
      description: "Name of Drugs"

    - name: product_class
      description: "Class of Drug (Antibiotics, etc.)"

    - name: product_id
      description: "This is a unique identifier for  different products"
      tests:
        - unique
        - not_null
        - dbt_constraints.primary_key
        - dbt_expectations.expect_column_to_exist
```

After dbt run we can check whether the test id pass or fail and we will get it as follows



```
1 of 5 START test dbt_constraints_foreign_key_stg_sales_product_id__product_id__ref_stg_product_  [RUN]
2 of 5 START test dbt_constraints_primary_key_stg_product_product_id .......... [RUN]
3 of 5 START test dbt_expectations_expect_column_to_exist_stg_product_product_id  [RUN]
4 of 5 START test not_null_stg_product_product_id ............................ [RUN]
5 of 5 START test unique_stg_product_product_id ............................... [RUN]
2 of 5 PASS dbt_constraints_primary_key_stg_product_product_id ................ [PASS in 2.57s]
5 of 5 PASS unique_stg_product_product_id ..................................... [PASS in 2.57s]
4 of 5 PASS not_null_stg_product_product_id ................................... [PASS in 2.57s]
1 of 5 PASS dbt_constraints_foreign_key_stg_sales_product_id__product_id__ref_stg_product_  [PASS in 2.58s]
3 of 5 PASS dbt_expectations_expect_column_to_exist_stg_product_product_id ..... [PASS in 2.57s]
```

Distribution:

Distribution specifies if the data is within the expected range. For dbt, it can be achieved using the dbt_profiler package. This helps to capture the information of the table into docs that can be updated at every run.

Installation

dbt version required: >=1.1.0

Include the following in your packages.yml file:

```
packages:
  - package: data-mie/dbt_profiler
    version: 0.8.1
```

To install latest version of package please refer package Hub

Run **dbt deps** to install the package.

dbt-profiler implements dbt macros for profiling database relations and creating doc blocks and table schemas (schema.yml) containing said profiles. For further executions refer dbt_profiler

A calculated profile contains the following measures for each column in a relation:

- column_name: Name of the column
- data_type: Data type of the column
- not_null_proportion^: Proportion of column values that are not NULL (e.g., 0.62 means that 62% of the values are populated while 38% are NULL)
- distinct_proportion^: Proportion of unique column values (e.g., 1 means that 100% of the values are unique)
- distinct_count^: Count of unique column values
- is_unique^: True if all column values are unique
- min*^: Minimum column value
- max*^: Maximum column value
- avg**^: Average column value
- median**^: Median column value
- std_dev_population**^: Population standard deviation
- std_dev_sample**^: Sample standard deviation
- profiled_at: Profile calculation date and time

There are different types of micros available to get profile docs I have use the following micro
https://github.com/data-mie/dbt-profiler/blob/0.8.1/macros/print_profile_docs.sql
After adding the micro into your DBT project’s micro folder call the micro as an operation for that we need to run command

```
dbt run-operation print_profile_docs --args '{"relation_name": "customers"}'
```

Example output

```
{% docs dbt_profiler__customers  %}

| column_name       | data_type | not_null_proportion | distinct_proportion | distinct_count | is_unique | min        | max        | avg |
std_dev_population |      std_dev_sample | profiled_at                  |

| ---------------------- | --------- | ------------------ | ------------------ | ------------- | --------- | ---------- | ---------- | ------------------ | --------------
----- | ------------------ | ---------------------------- |

| customer_id          | int64     |               1.00 |               1.00 |           100 |         1 | 1          | 100        | 50.5000000000000000 |
28.8660700477221200 | 29.0114919758820200 | 2022-01-13 10:14:48.300040+00 |

| first_order          | date      |               0.62 |               0.46 |            46 |         0 | 2018-01-01 | 2018-04-07 |                    |                    |
| 2022-01-13 10:14:48.300040+00 |

| most_recent_order      | date      |               0.62 |               0.52 |            52 |         0 | 2018-01-09 | 2018-04-09 |                    |
|                  | 2022-01-13 10:14:48.300040+00 |

| number_of_orders      | int64     |               0.62 |               0.04 |             4 |         0 | 1          | 5          |  1.5967741935483863 |
0.7716692718648833 |  0.7779687173818426 | 2022-01-13 10:14:48.300040+00 |

| customer_lifetime_value | float64   |               0.62 |               0.35 |            35 |         0 | 1          | 99         | 26.9677419354838830 |
18.6599171435558730 | 18.8122455252636630 | 2022-01-13 10:14:48.300040+00 |
```

Volume:

Volume refers to the completeness of the data and data source health. For volume, the Bespoke dbt test (data test) can be run as part of the data pipeline validating the count on the data or the audit_helper package can be used to easily compare column values across tables.

Installation

dbt version required: >=1.2.0, <2.0.0

Include the following in your packages.yml file:

```
packages:
  - package: dbt-labs/audit_helper
    version: 0.9.0
```

To install latest version of package please refer package Hub

Run **dbt deps** to install the package.

audit_helper is a package for dbt whose main purpose is to audit data by comparing two tables (the original one versus a refactored model). It uses a simple and intuitive query structure that enables quickly comparing tables based on the column values, row amount, and even column types (for example, to make sure that a given column is numeric in both your table and the original one). Figure 1 graphically displays the workflow and where audit_helper is positioned in the refactoring process.

For data audits there are different types of macros available you can refer audit_helper and according to your need you can add macro into . I have use compare_relations macro

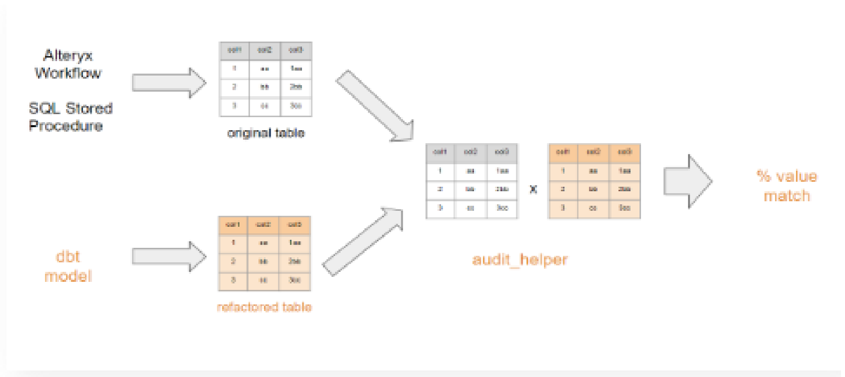


Figure 1 — Workflow of auditing process using audit_helper

Now that it is clear where the audit_helper package is positioned in the refactoring process, it is important to highlight the benefits of using audit_helper (and ultimately, of auditing refactored models). Among the benefits, we can mention:

- **Quality assurance**: Assert that a refactored model is reaching the same output as the original model that is being refactored.
- **Easy and intuitive code**: Because audit_helper relies on dbt macros, it was designed to be an intuitive tool that runs on simple SQL queries.
- **Clear output**: Audit_helper provides clear output showing how much of your refactored table matches the original one.
- **Visibility to a project**: Audit_helper gives visibility to a refactoring process by showing how your code is delivering the same results in both row-wise and column-wise comparisons.
- **Flexibility to compare rows and columns**: It is simple to quickly compare the results in rows or columns through pre-made templates that just require you to place your columns' names and the original model's ones.

Audit rows (compare_queries)

According to the audit_helper package documentation, this macro comes in handy when:

You need to filter out records from one of the relations,
Some columns must be renamed or recast in order to match,
But you only want to compare a few columns, since it's simpler to write the columns you want to compare rather than the columns you want to exclude.

How it works

When you run the dbt audit model, it will compare all columns, row by row. To count for the match, every column in a row from one source must exactly match a row from another source, as illustrated in the example in Figure 2 below:

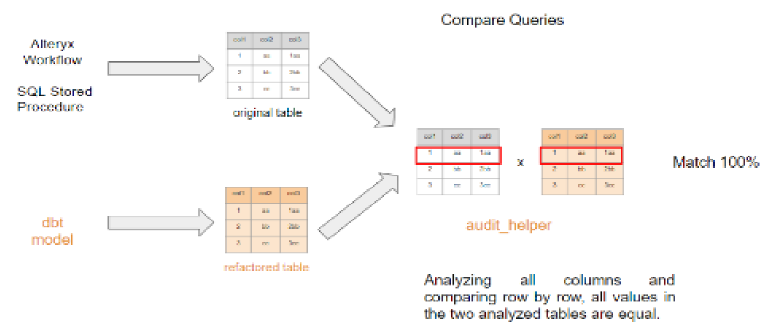


Figure 2 — Workflow of auditing rows (compare_queries) using audit_helper

As shown in the example, the model is compared line by line, and in this case, all lines in both models are equivalent and the result should be 100%. Figure 3 below depicts a row in which two of the three columns are equal and only the last column of row 1 has divergent values. In this case, despite the fact that most of row 1 is identical, that row will not be counted towards the final result. In this example, only row 2 and row 3 are valid, yielding a 66.6% match in the total of analyzed rows.

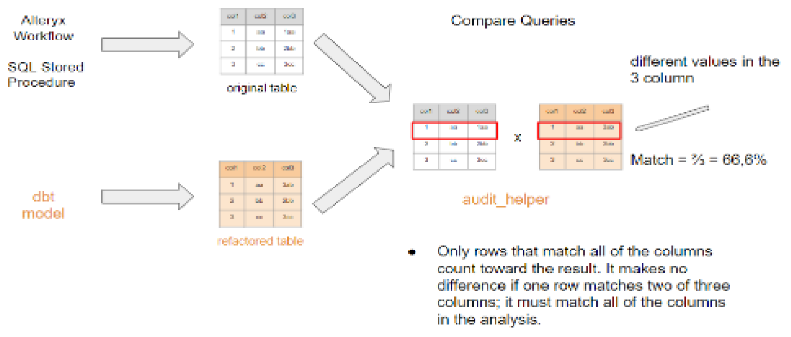
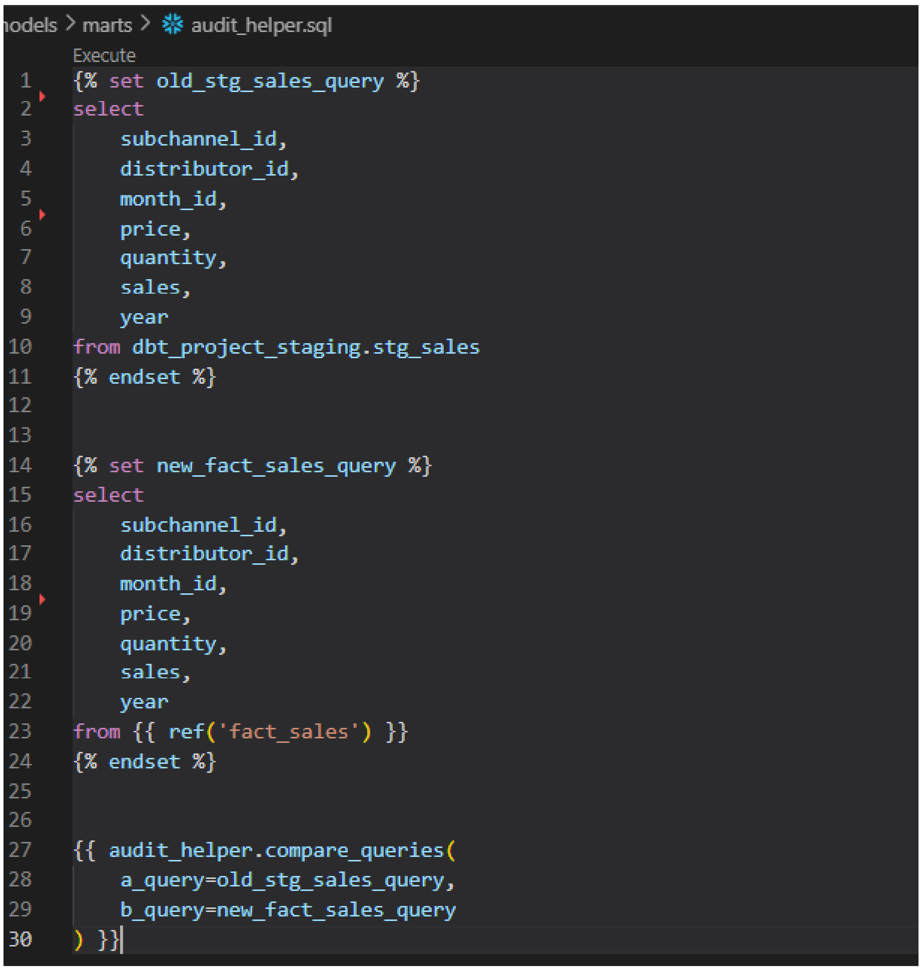

Alteryx Workflow
SQL Stored Procedure
original table
dbt model
refactored table
Compare Queries
different values in the 3 column
x
Match = ⅔ = 66,6%
audit_helper
Only rows that match all of the columns count toward the result. It makes no difference if one row matches two of three columns; it must match all of the columns in the analysis.

Figure 3 — Example of different values

As previously stated, for the match to be valid, all column values of a model's row must be equal to the other model. This is why we sometimes need to exclude columns from the comparison (such as date columns, which can have a time zone difference from the original model to the refactored

The code for the compare_queries macro: Step-by-step

- Create a new .sql model in the folder of your choice
- Copy and paste the following example below in the model created:



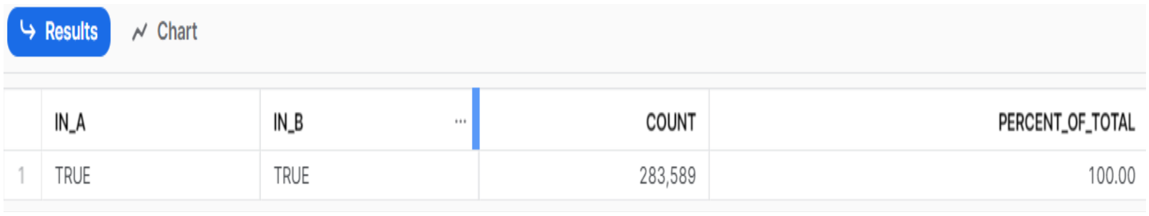
```
{% set old_stg_sales_query %}
select
    subchannel_id,
    distributor_id,
    month_id,
    price,
    quantity,
    sales,
    year
from dbt_project_staging.stg_sales
{% endset %}


{% set new_fact_sales_query %}
select
    subchannel_id,
    distributor_id,
    month_id,
    price,
    quantity,
    sales,
    year
from {{ ref('fact_sales') }}
{% endset %}


{{ audit_helper.compare_queries(
    a_query=old_stg_sales_query,
    b_query=new_fact_sales_query
) }}
```

- Run the audit model as you would run any other dbt model using the command below:
- **dbt run --select <name of your audit model>**

Check the result by copying and pasting the code below into your development environment(snowflake/postgres/bigquery) :

```
select * from <name of your audit model>
```

The output will be the similar to the one shown in Figure



	IN_A	IN_B	COUNT	PERCENT_OF_TOTAL
1	TRUE	TRUE	283,589	100.00

Schema:

Schema check refers to the changes in the organization of the data in the table. dbt is well known for its capabilities of schema testing. It has four generic tests (unique, not null, accepted values, and relationships) built-in that are defined in the *schema.yml*. Custom generic tests and the package dbt_utils can be used to easily implement other common schema tests.

Installation
dbt version required: >=1.2.0, <2.0.0

Include the following in your packages.yml file:

```
packages:

  - package: fishtown-analytics/dbt_utils

    version: 0.7.0
```

To install latest version of package please refer package Hub

Run **dbt deps** to install the package.

For further executions refer dbt_utils
For generic tests refer generic_tests

If you've set up any dbt tests, you've probably used some of the generic ones like not_null, of which dbt natively supports only 4. dbt utils gives you, well, more of them. We'll cover two examples but there are tons more.

equal_rowcount
Although it's quite basic, the equal_rowcount macro is probably one of the most important tests that can be applied to any data set. This macro simply asserts that two relations—tables, views, or models—have the same number of rows. It returns true if the row count in both relations is the same and false if it is not. Super useful when you're making a model change and want to verify that the table in dev matches the one in prod.

In this example, we're testing the orders model against the stg_orders model (which could be a staging table for the orders data). The test will pass if both models have the same number of rows, and it will fail otherwise.

```
version: 2

models:

  - name: orders

    columns:

      - name: id

        tests:

          - dbt_utils.equal_rowcount:

              compare_model: ref('stg_sales')
```

After defining the test, you can run it at the command line with dbt test. dbt outputs the results of the test (among your other tests), showing you whether it passed or failed.

not_accepted_values

Another macro that is perfect for testing is not_accepted_values, allowing you to assert that certain columns of a table never take on values from among a specified set. This macro helps to prevent incorrect or inconsistent data from being saved in the database. Note that dbt ships with the accepted_values generic test, and this is just the inverse of that.

Let's assume you have a users table, and you want to ensure that the status column never has the value inactive. You would define this as follows:

```
version: 2

models:

  - name: users

    columns:

      - name: status

        tests:

          - dbt_utils.not_accepted_values:

              values: ['inactive']
```

The values parameter is a list of unacceptable values.

Lineage:

Data lineage is the process of understanding, recording, and visualizing data as it flows from data sources to consumption. Data lineage is part of the dbt docs and it makes use of the *sources.yml* and *schema.yml* to provide a consumable HTML based report and a data dependency graph
You can generate a documentation site for your project (with or without descriptions) using the CLI.

First, run **dbt docs generate** — this command tells dbt to compile relevant information about your dbt project and warehouse into manifest.json and catalog.json files respectively. To see documentation for all columns and not just columns described in your project, ensure that you have created the models with dbt run beforehand. Then, run **dbt docs serve** to use these .json files to populate a local website.

Good documentation for your dbt models will help downstream consumers discover and understand the datasets which you curate for them.

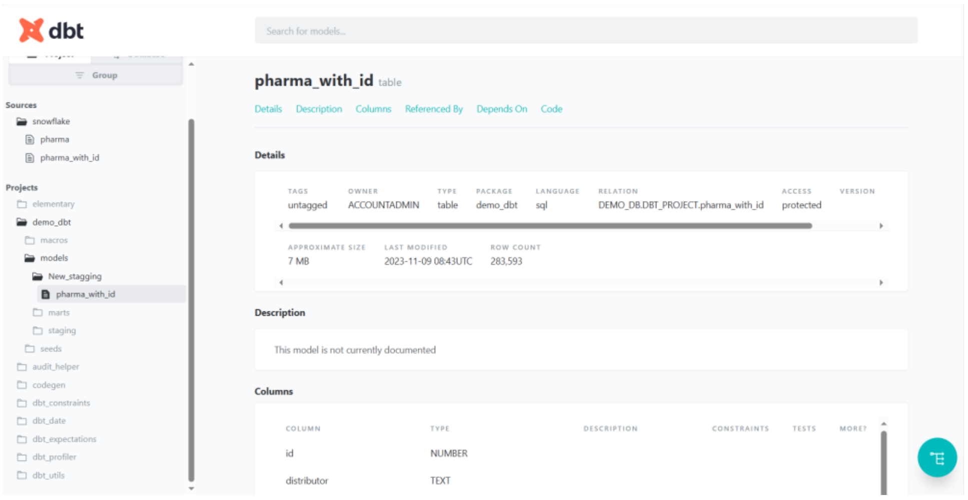
dbt provides a way to generate documentation for your dbt project and render it as a website. The documentation for your project includes:

- **Information about your project**: including model code, a DAG of your project, any tests you've added to a column, and more.
- **Information about your data warehouse**: including column data types, and table sizes. This information is generated by running queries against the information schema.

Importantly, dbt also provides a way to add **descriptions** to models, columns, sources, and more, to further enhance your documentation.

Navigating the documentation site

Using the docs interface, you can navigate to the documentation for a specific model. That might look something like this:


dbt

Search for models...

Group

Sources
- snowflake
  - pharma
  - pharma_with_id

Projects
- elementary
- demo_dbt
  - macros
  - models
    - New_stagging
      - pharma_with_id
    - marts
    - staging
  - seeds
- audit_helper
- codegen
- dbt_constraints
- dbt_date
- dbt_expectations
- dbt_profiler
- dbt_utils

pharma_with_id table

Details Description Columns Referenced By Depends On Code

Details

TAGS	OWNER	TYPE	PACKAGE	LANGUAGE	RELATION	ACCESS	VERSION
untagged	ACCOUNTADMIN	table	demo_dbt	sql	DEMO_DB.DBT_PROJECT.pharma_with_id	protected	

APPROXIMATE SIZE	LAST MODIFIED	ROW COUNT
7 MB	2023-11-09 08:43UTC	283,593

Description

This model is not currently documented

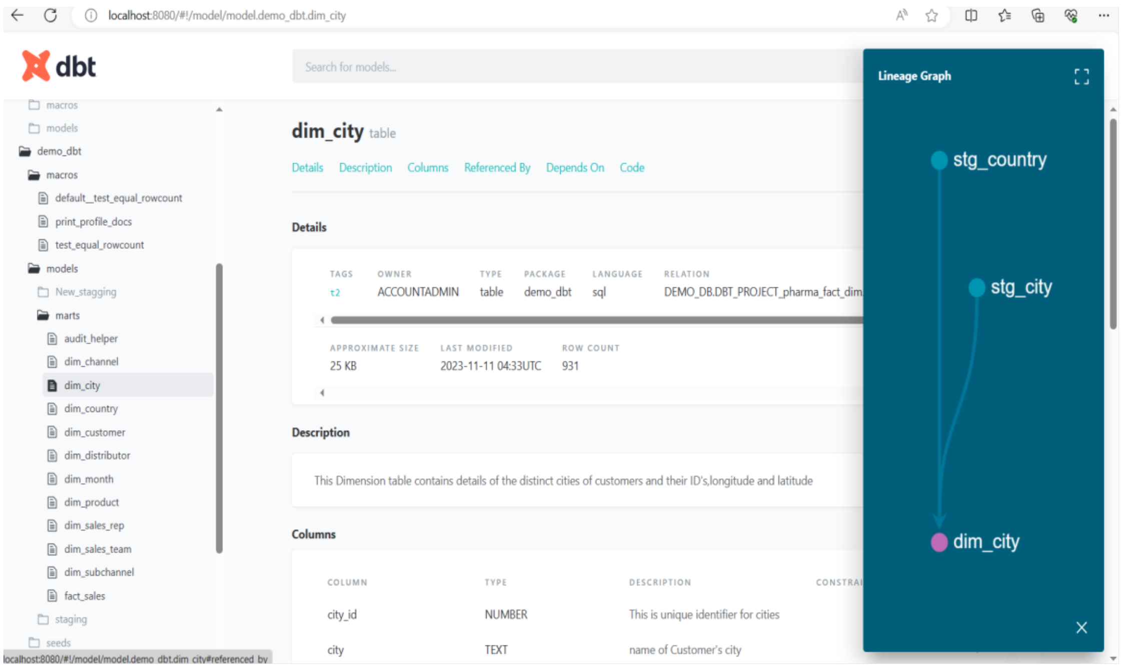
Columns

COLUMN	TYPE	DESCRIPTION	CONSTRAINTS	TESTS	MORE?
id	NUMBER				
distributor	TEXT				

Here, you can see a representation of the project structure, a markdown description for a model, and a list of all of the columns (with documentation) in the model.

From a docs page, you can click the green button in the bottom-right corner of the webpage to expand a "mini-map" of your DAG. This pane (shown below) will display the immediate parents and children of the model that you're exploring.

Here's an example docs site:


localhost:8080/#!/model/model.demo_dbt.dim_city

dbt

Search for models...

- macros
- models
- demo_dbt
  - macros
    - default__test_equal_rowcount
    - print_profile_docs
    - test_equal_rowcount
  - models
    - New_stagging
    - marts
      - audit_helper
      - dim_channel
      - dim_city
      - dim_country
      - dim_customer
      - dim_distributor
      - dim_month
      - dim_product
      - dim_sales_rep
      - dim_sales_team
      - dim_subchannel
      - fact_sales
    - staging
  - seeds

localhost:8080/#!/model/model.demo_dbt.dim_city#referenced_by

dim_city table

Details Description Columns Referenced By Depends On Code

Details

TAGS	OWNER	TYPE	PACKAGE	LANGUAGE	RELATION
t2	ACCOUNTADMIN	table	demo_dbt	sql	DEMO_DB.DBT_PROJECT_pharma_fact_dim

APPROXIMATE SIZE	LAST MODIFIED	ROW COUNT
25 KB	2023-11-11 04:33UTC	931

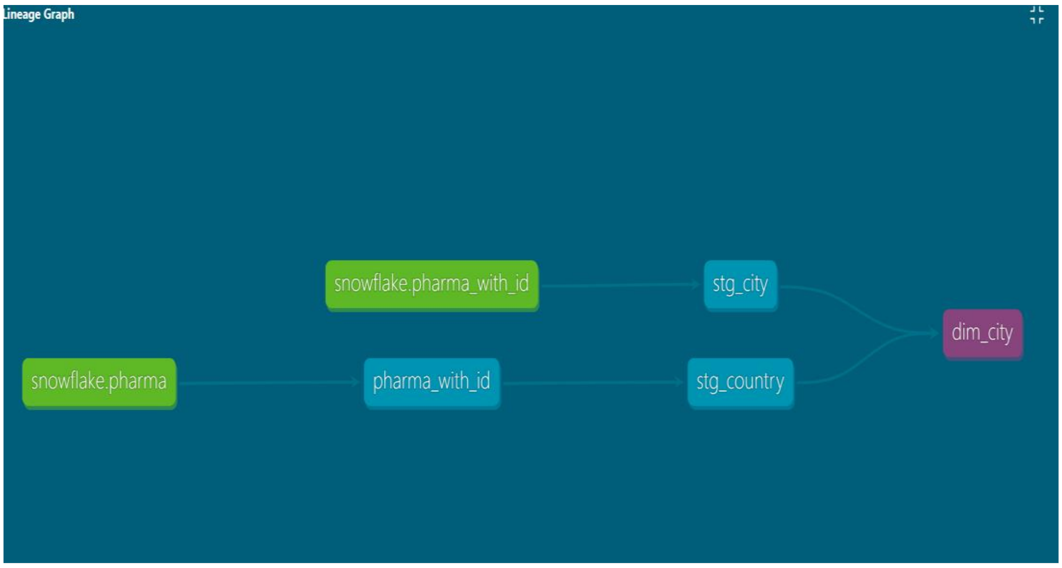
Description

This Dimension table contains details of the distinct cities of customers and their ID's,longitude and latitude

Columns

COLUMN	TYPE	DESCRIPTION	CONSTRAI
city_id	NUMBER	This is unique identifier for cities	
city	TEXT	name of Customer's city	

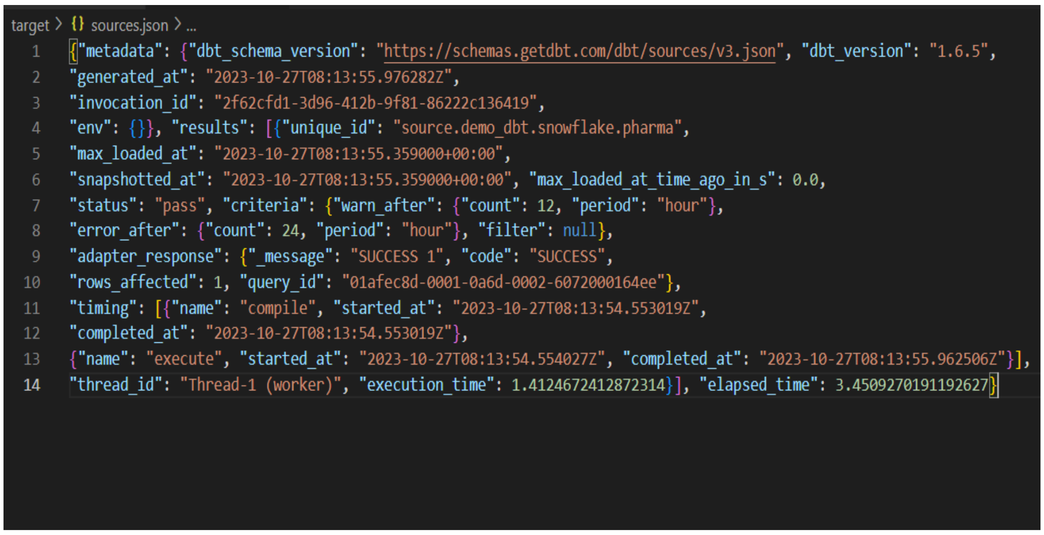
In this example, the dim_city model By clicking the "Expand" button in the top-right corner of the window, we can pivot the graph horizontally and view the full lineage for our model. This lineage is filterable using the --select and --exclude flags, which are consistent with the semantics of model selection syntax. Further, you can right-click to interact with the DAG, jump to documentation, or share links to your graph visualization with your coworkers.



Like this we can check lineage of all models and lineage of full project also

Outcome

- In **DBT freshness** If any source table hasn't been updated as per the specified freshness configuration, dbt will raise a warning or error. After running the above command in the target folder the **sources. Json** file will create to check the freshness it can be as follows



```
{"metadata": {"dbt_schema_version": "https://schemas.getdbt.com/dbt/sources/v3.json", "dbt_version": "1.6.5",
"generated_at": "2023-10-27T08:13:55.976282Z",
"invocation_id": "2f62cfd1-3d96-412b-9f81-86222c136419",
"env": {}}, "results": [{"unique_id": "source.demo_dbt.snowflake.pharma",
"max_loaded_at": "2023-10-27T08:13:55.359000+00:00",
"snapshotted_at": "2023-10-27T08:13:55.359000+00:00", "max_loaded_at_time_ago_in_s": 0.0,
"status": "pass", "criteria": {"warn_after": {"count": 12, "period": "hour"},
"error_after": {"count": 24, "period": "hour"}, "filter": null},
"adapter_response": {"_message": "SUCCESS 1", "code": "SUCCESS",
"rows_affected": 1, "query_id": "01afec8d-0001-0a6d-0002-6072000164ee"},
"timing": [{"name": "compile", "started_at": "2023-10-27T08:13:54.553019Z",
"completed_at": "2023-10-27T08:13:54.553019Z"},
{"name": "execute", "started_at": "2023-10-27T08:13:54.554027Z", "completed_at": "2023-10-27T08:13:55.962506Z"}],
"thread_id": "Thread-1 (worker)", "execution_time": 1.4124672412872314}], "elapsed_time": 3.4509270191192627}
```

- **Dbt expectations** provide so many tests and it can be defined in the schema.yml file and we can define appropriate tests according to our need it can be applied to columns where we generally define the generic tests and after that if we run command **dbt run** the test will be check for example if we need to test **expect_row_values_to_have_recent_data(**Expect the model to have rows that are at least as recent as the defined interval prior to the current timestamp. Optionally gives the possibility to apply filters on the results**)** so we will define it as

```
tests:
  - dbt_expectations.expect_row_values_to_have_recent_data:
      datepart: day
      interval: 1
      row_condition: 'id is not null' #optional
```

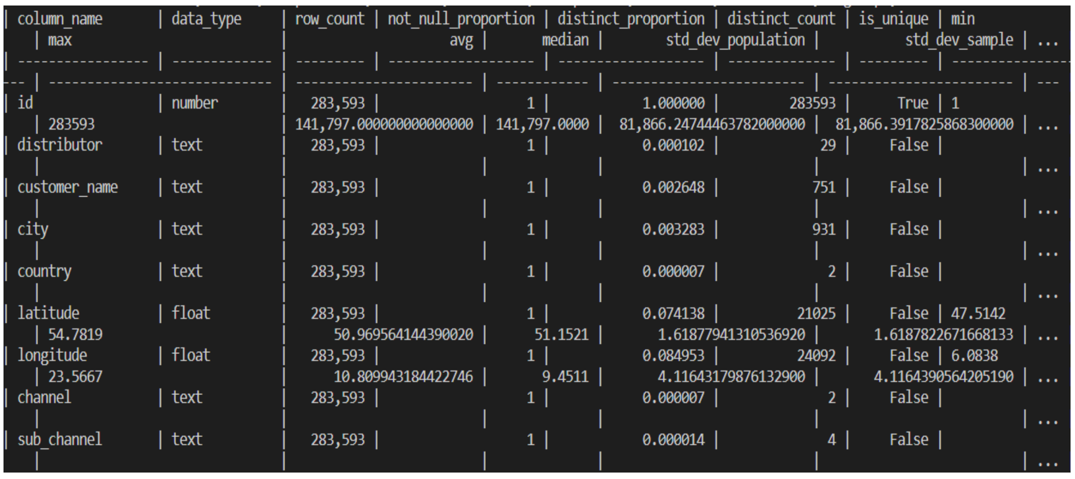
If the test condition satisfied after dbt test the result will be pass and if the condition will not satisfy the result will be failed.

- **dbt-profiler** implements dbt macros for profiling database relations and creating doc blocks after adding the print_profile_doc macro in our deb project you need to run the following command to create the profile.

```
dbt run-operation print_profile_docs --args '{"relation_name": "pharma_with_id"}'
```

choose model name according to your project.

The sample output:



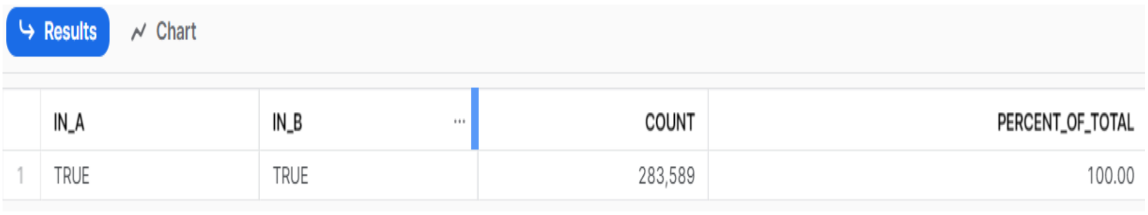
```
| column_name      | data_type    | row_count | not_null_proportion | distinct_proportion | distinct_count | is_unique | min
   | max                          |                   avg |      median |         std_dev_population |          std_dev_sample | ...
| ---------------- | ------------ | --------- | ------------------- | ------------------- | -------------- | --------- | ---------------
--- | --------------------------- | --------------------- | ----------- | ---------------------- | ---------------------- | ---
| id               | number       |   283,593 |                   1 |            1.000000 |         283593 |      True | 1
   | 283593                       | 141,797.0000000000000000 | 141,797.0000 | 81,866.24744463782000000 | 81,866.39178258683000000 | ...
| distributor      | text         |   283,593 |                   1 |            0.000102 |             29 |     False |
   |                              |                       |             |                        |                        | ...
| customer_name    | text         |   283,593 |                   1 |            0.002648 |            751 |     False |
   |                              |                       |             |                        |                        | ...
| city             | text         |   283,593 |                   1 |            0.003283 |            931 |     False |
   |                              |                       |             |                        |                        | ...
| country          | text         |   283,593 |                   1 |            0.000007 |              2 |     False |
   |                              |                       |             |                        |                        | ...
| latitude         | float        |   283,593 |                   1 |            0.074138 |          21025 |     False | 47.5142
   | 54.7819                      |    50.969564144390020 |     51.1521 |    1.61877941310536920 |     1.6187822671668133 | ...
| longitude        | float        |   283,593 |                   1 |            0.084953 |          24092 |     False | 6.0838
   | 23.5667                      |    10.809943184422746 |      9.4511 |    4.11643179876132900 |     4.1164390564205190 | ...
| channel          | text         |   283,593 |                   1 |            0.000007 |              2 |     False |
   |                              |                       |             |                        |                        | ...
| sub_channel      | text         |   283,593 |                   1 |            0.000014 |              4 |     False |
   |                              |                       |             |                        |                        | ...
```

- **Audit helper** is Useful macros when performing data audits I have use audit helper to compare queries it is

1. Super similar to compare_relations, except it takes two select statements. This macro is useful when:
2. You need to filter out records from one of the relations.
3. You need to rename or recast some columns to get them to match up.

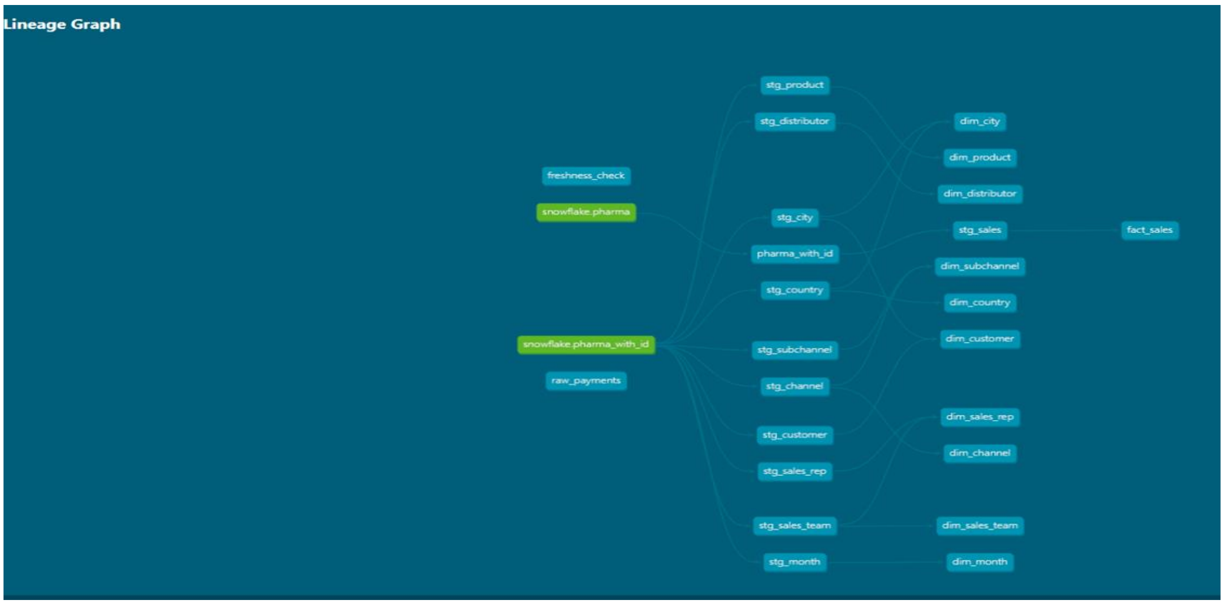
4. You only want to compare a small number of columns, so it's easier write the columns you want to compare, rather than the columns you want to exclude.

The sample result is as follolws.



	IN_A	IN_B	COUNT	PERCENT_OF_TOTAL
1	TRUE	TRUE	283,589	100.00

- lineage graph



Elementary

Concepts

What is Elementary?

Elementary is an open-source data observability solution for data & analytics engineers. Monitor your dbt project and data in minutes, and be the first to know of data issues. Gain immediate visibility, detect data issues, send actionable alerts, and understand the impact and root cause.

Key features

- **Data observability report**
  Generate a data observability report, host it or share with your team.
- **Data anomaly detection as dbt tests**
  Monitoring of data quality metrics, freshness, volume and schema changes, including anomaly detection. Elementary data monitors are configured and executed like native tests in dbt your project.
- **Models performance**
  Visibility of execution times, easy detection of degradation and bottlenecks.
- **dbt artifacts and run results**
  Uploading and modelling of dbt artifacts, run and test results to tables as part of your runs.

- **Slack alerts**
  Get informative notifications on data issues, schema changes, models and tests failures.
- **Data lineage**
  Inspect upstream and downstream dependencies to understand impact and root cause of data issues.

Execution

Install dbt package

1. Add elementary to packages.yml
Add the following to your packages.yml file

```
packages:
  - package: elementary-data/elementary
    version: 0.11.0
    ## Docs: https://docs.elementary-data.com
```

2. Add to your dbt_project.yml

```
models:
  ## see docs: https://docs.elementary-data.com/
  elementary:
    ## elementary models will be created in the schema '<your_schema>_elementary'
    +schema: "elementary"
    ## To disable elementary for dev, uncomment this:
    # enabled: "{{ target.name in ['prod','analytics'] }}"
```

3. Import the package

```
dbt deps
```

4. Run to create the package models

```
dbt run --select elementary
```

5. Run tests

```
dbt test
```

After you ran your tests, we recommend that you ensure that the results were loaded to elementary_test_results table.

Install Elementary CLI

To install the monitor module run:

```
pip install elementary-data
```

Run one of the following commands based on your platform (no need to run all):

```
pip install 'elementary-data[snowflake]'

pip install 'elementary-data[bigquery]'

pip install 'elementary-data[redshift]'

pip install 'elementary-data[databricks]'

## Postgres doesn't require this step
```

Generate Tests Report UI

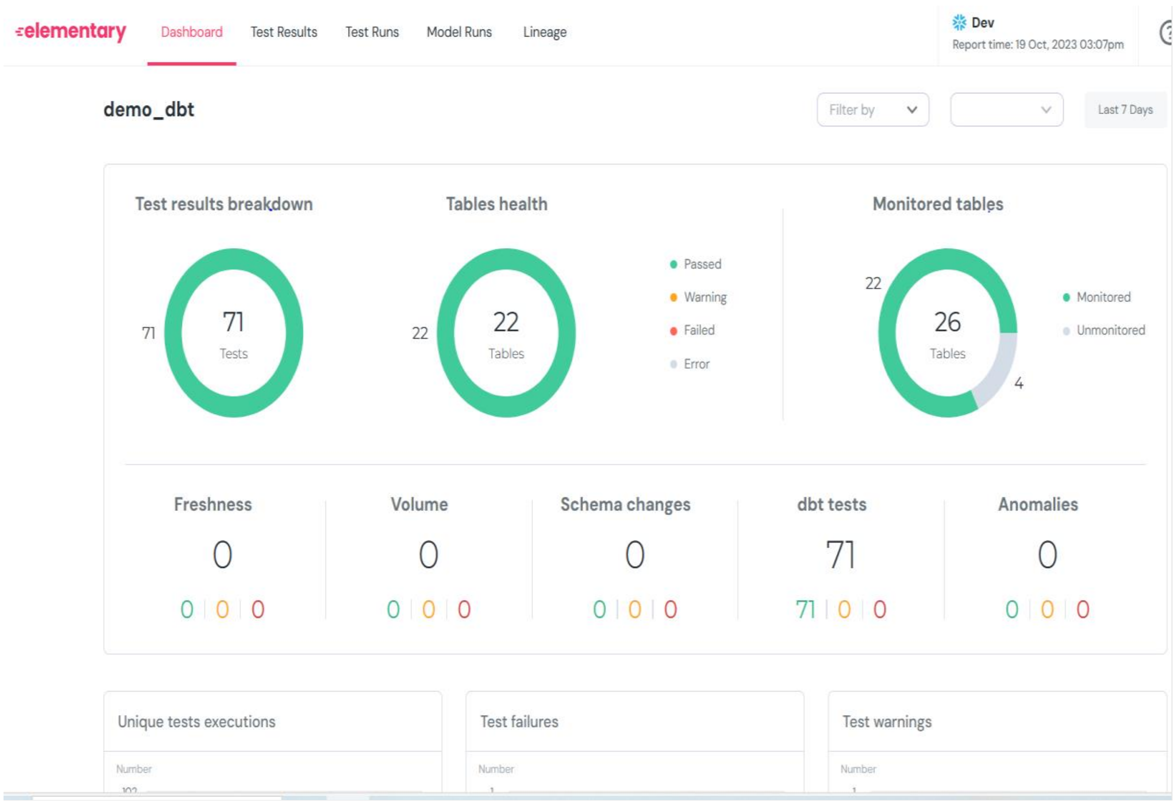
Execute edr report **in your terminal**
After installing and configuring the CLI, execute the command:

```
edr report
```

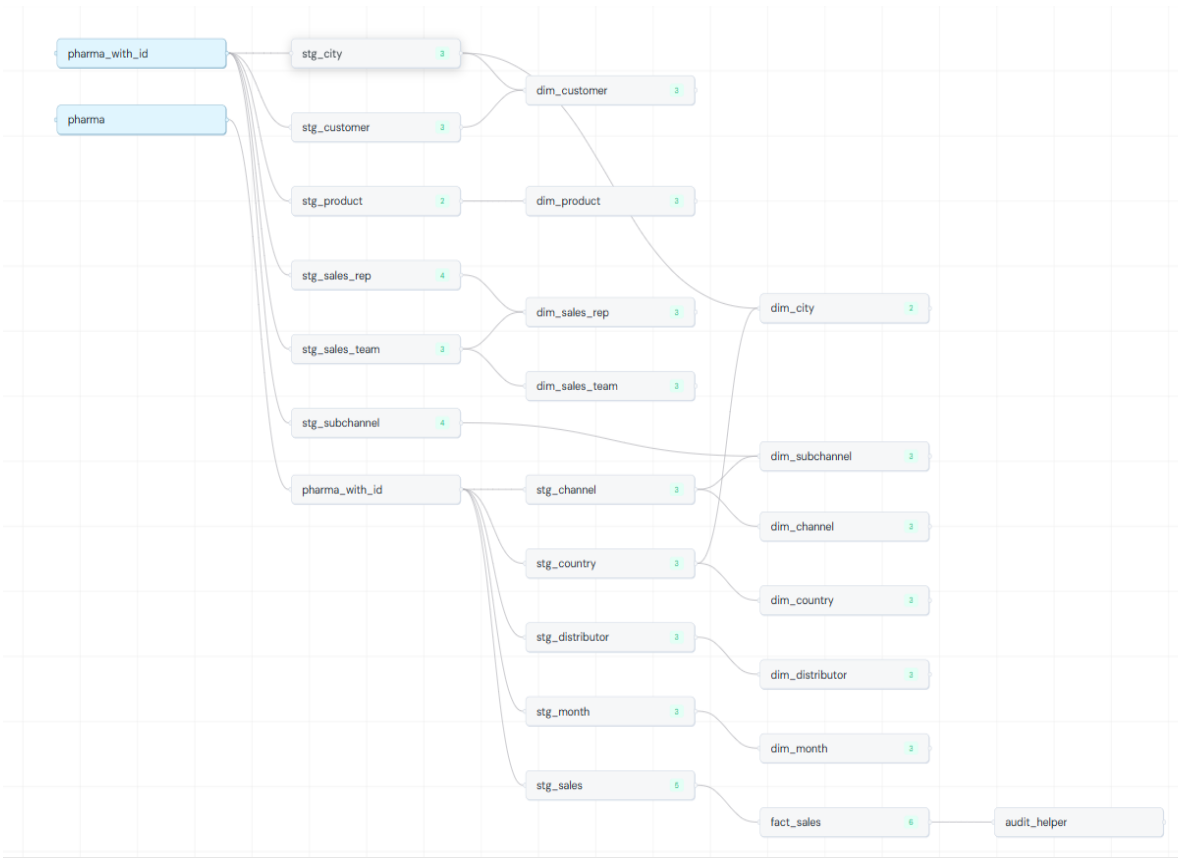
The command will use the provided connection profile to access the data warehouse, read from the Elementary tables, and generate the report as an HTML file.

Outcome

Elementary dashboard



Lineage graph:


pharma_with_id
pharma
stg_city
stg_customer
stg_product
stg_sales_rep
stg_sales_team
stg_subchannel
pharma_with_id
dim_customer
dim_product
dim_sales_rep
dim_sales_team
stg_channel
stg_country
stg_distributor
stg_month
stg_sales
dim_city
dim_subchannel
dim_channel
dim_country
dim_distributor
dim_month
fact_sales
audit_helper

Unleashing dbt's potential

Alternative to stored procedure

The Drawbacks of Stored Procedures

Stored procedures are widely used throughout the data warehousing world. They're great for encapsulating complex transformations into units that can be scheduled and respond to conditional logic via parameters. However, as teams continue building their transformation logic using the stored procedure approach, we see more data downtime, increased data warehouse costs, and incorrect / unavailable data in production. All of this leads to more stressed and unhappy developers, and consumers who have a hard time trusting their data.

If your team works heavily with stored procedures, and you ever find yourself with the following or related issues:

- dashboards that aren't refreshed on time
- It feels too slow and risky to modify pipeline code based on requests from your data consumers
- It's hard to trace the origins of data in your production reporting

It's worth considering if an alternative approach with dbt might help.
dbt is a wonderful tool that helps in transforming data in real time by just using SELECT statements. Dbt takes these input statements and turns them into views and tables. Based on this principle, dbt solves major problems in the data industry, and it works as an effective alternative to stored procedures.

At first, stored procedures make complete sense but their disadvantages become apparent later on when we expect data pipelines to assist in sophisticated but necessary processes such as testability, transparent documentation, and code reusability. Stored procedures aren't testable since they don't function well in the documentation data flow, which affects documentation negatively as well.

Stored procedures could turn into a source of stress for development teams as their intermediate steps aren't obvious and most of the time you'll find the same code repeated in more than one stored procedure. This lowers the efficiency of your teams massively. To be more practical, stored procedures could be the reason why it's difficult to trace the primary origins of data in production reports and why your dashboards don't refresh in a timely manner.

Why consider dbt as an alternative?

dbt offers an approach that is self-documenting, testable, and encourages code reuse during development. One of the most important elements of working in dbt is embracing modularity when approaching data pipelines. In dbt, each business object managed by a data pipeline is defined in a separate model (think: orders data). These models are flexibly grouped into layers to reflect the progression from raw to consumption ready data. Working in this way, we create reusable components which helps avoid duplicating data and confusion among development teams.

Modularity is key when working with dbt. Business objects are managed in separate models and organized into layers which make the data easier to be consumed and namely more testable and self-documenting. Also, this approach minimizes data duplication and simplifies code reuse.
Furthermore, dbt supports version control integration which helps data workers implement and test changes with git-based tools in their transformation pipelines in just a few clicks. We can say that the two most straightforward benefits of this migration are:

1. Solutions to New Use Cases
Dbt uses a real-life scenario of a dbt Cloud use to illustrate the potential of this migration. As a result, after moving from using stored procedures to using dbt, the team could work with new use cases such as real-time data reporting with lambda views.

2. Uptime Enhancements
The refreshing time can slow down whole processes in an organization if it doesn't match the standards of other systems and clients' expectations. After migrating from using stored procedures to dbt, the team we mentioned was able to spend less time on pipeline refreshes, improving the uptime to 99.99% (from 65%).

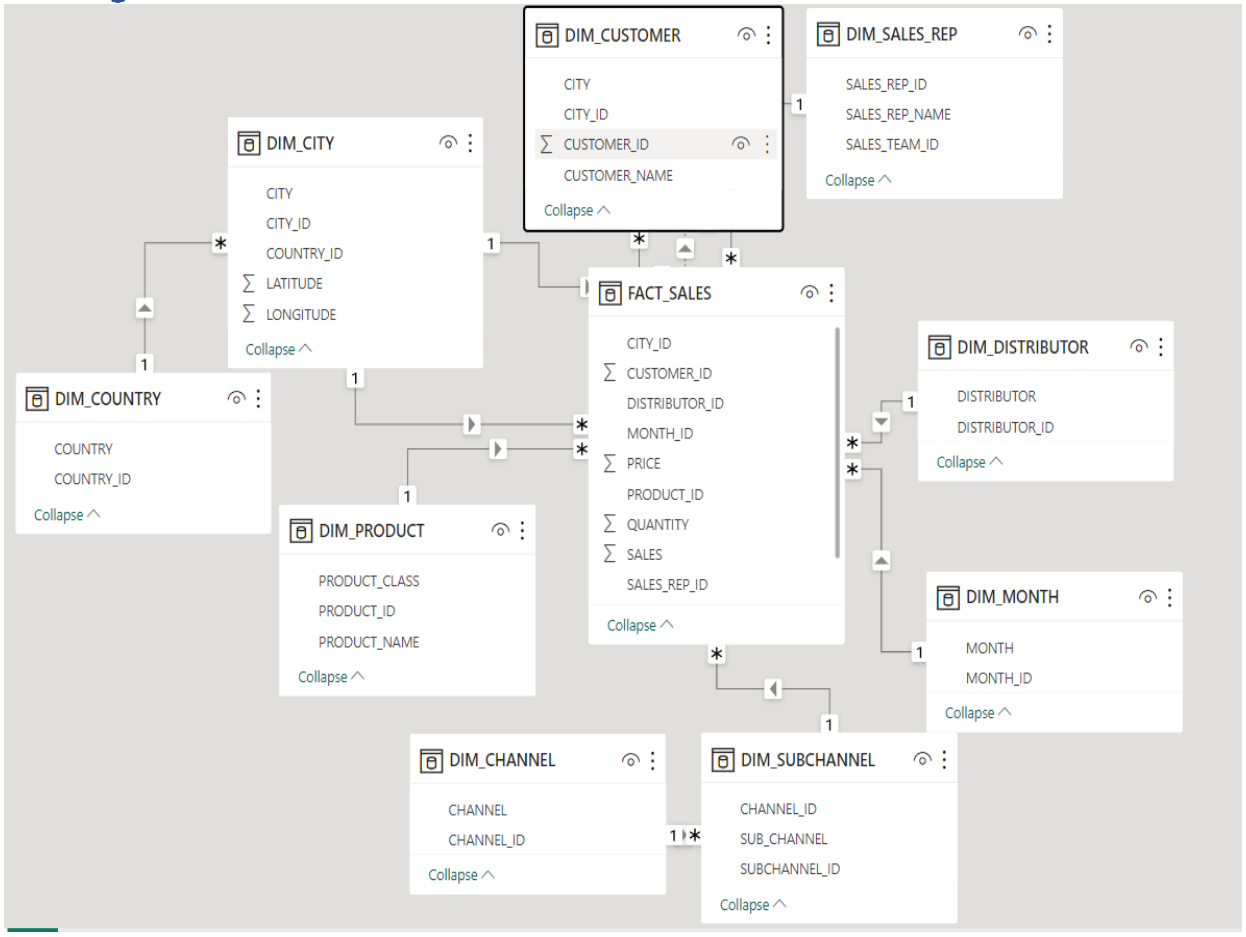
Appendix

User and role creation script

Sample raw data

pharma_sample_data

ER diagram of facts and dimension tables



Links

- Model configurations

  https://docs.getdbt.com/reference/model-configs

- Github URL of the sample project

  Github Repository

- Snowflake setup

  snowflake setup

- Data Observability

  https://medium.com/snowflake/implementing-data-observability-in-dbt

- Data Observability using elementary

  https://docs.elementary-data.com/understand-elementary/cli-install

- Semantic Layer

  https://www.getdbt.com/blog/dbt-semantic-layer-whats-next

  https://docs.getdbt.com/docs/build/semantic-models

- Jinja Templates

  https://docs.getdbt.com/guides/advanced/using-jinja

- Dbt Commands cheat sheet

  https://datacaffee.com/dbt-data-built-tool-commands-cheat-sheet/

- Materialization

  https://docs.getdbt.com/docs/build/materializations

  incremental-models