**Getting Started with dbt (Data Build Tool): A Beginner’s Guide to Building Data Transformations**

[[](https://medium.com/@suffyan.asad1?source=post_page-----28e335be5f7e--------------------------------)](https://medium.com/@suffyan.asad1?source=post_page-----28e335be5f7e--------------------------------)

[Suffyan Asad](https://medium.com/@suffyan.asad1?source=post_page-----28e335be5f7e--------------------------------)

·

Follow

19 min read

·

Jul 6, 2023

172

1

**Introduction**

Data Build Tool, commonly known as dbt, has gained recently significant popularity in the realm of data pipelines. Intrigued by its popularity, I have been exploring dbt lately. This article has been written based on my exploration and self-learning, and is intended to serve as a tutorial for beginners who are just embarking on learning dbt and intend to use it in their data pipelines. dbt is a data transformation tool. Its purpose is to allow development of data transformation steps that are well documented, reusable, and testable.

A room with a large white floor

Description automatically generated

Photo by [Taylor Vick](https://unsplash.com/@tvick?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

**What exactly is dbt, and why does it exist?**

Before we dive into the details, let’s address a fundamental question: What exactly is dbt, and what purpose does it serve? Moreover, how is it superior to executing SQL scripts within, for instance, Python code? Python code can be version controlled, and it can be made somewhat generic by using code to generate or manipulate SQL.

While searching for the answer to these questions, I stumbled upon an excellent article that covers the utility and advantages of dbt in detail:

**[What is dbt?](https://www.getdbt.com/product/what-is-dbt/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[dbt is a data transformation tool that enables data analysts and engineers to transform data in a cloud analytics…](https://www.getdbt.com/product/what-is-dbt/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[www.getdbt.com](https://www.getdbt.com/product/what-is-dbt/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Although I highly recommend reading the entire article, I will provide a summary to give you a glimpse of dbt’s essence.

Data Build Tool or dbt is built to transform data, and is therefore, the T in an ELT pipeline. I mentioned ELT because it is designed to work after data has been loaded, and is ready for transformation. Additionally, out of the box, it cannot connect with multiple databases, and depends on data that has been loaded, or otherwise accessible to the target database executing the dbt steps. Additionally, how dbt runs is that it executes SQL on the target database.

The utility of dbt that makes it superior to other approaches originates from the combination of Jinja templates with SQL, and re-usable components or models. For example, consider an online customer model, which is defined as a customer whose 50% or more sales are online:

An example of dbt model: SQL with Jinja

**Building first dbt project**

Starting simple, the first dbt project is a simple selection from the sales.salesorderheader table in the AdventureWorks 2014 example database. It performs the following tasks:

* Select a few columns, and all rows from sales.salesorderheader table to a new staging\_warehouse.sales\_order\_header table.
* Rename the rowguid column to row\_id. Ensure uniqueness and not null using tests.
* Test that due date (duedate column) is always after the order date (orderdate column) using a custom test.

This tutorial uses PostgreSQL database instance running Adventure Works 2014 database.

**Setting up PostgreSQL database instance with Adventure Works 2014**

The first step is to set-up the database. Adventure Works 2014 is an example database by Microsoft, and is used as example in their tutorials and documentation:

**[AdventureWorks sample databases - SQL Server](https://learn.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver16&tabs=ssms&source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Follow these instructions to download and install AdventureWorks sample databases to SQL Server using Transact-SQL…](https://learn.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver16&tabs=ssms&source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[learn.microsoft.com](https://learn.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver16&tabs=ssms&source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

I usually use an open-source project that creates a docker container running Adventure Works 2014 on a PostgreSQL instance. The code can be obtained from the following repository:

**[GitHub - lorint/AdventureWorks-for-Postgres: Set up the AdventureWorks sample database for use with…](https://github.com/lorint/AdventureWorks-for-Postgres?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Set up the AdventureWorks sample database for use with Postgres - GitHub - lorint/AdventureWorks-for-Postgres: Set up…](https://github.com/lorint/AdventureWorks-for-Postgres?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[github.com](https://github.com/lorint/AdventureWorks-for-Postgres?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

After checking out the repository, starting up the instance is as simple as running docker-compose up. And then the instance can be queried by connecting to it over http://[localhost](<http://localhost>):5432 using postgres as username and password.

A screenshot of a computer

Description automatically generated

Setting up connection with the PostgreSQL database

**Completed project code**

Completed project code is present at the following GitHub repository:

**[dbt-tutorial/dbt\_tutorial at main · SA01/dbt-tutorial](https://github.com/SA01/dbt-tutorial/tree/main/dbt_tutorial?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Contribute to SA01/dbt-tutorial development by creating an account on GitHub.](https://github.com/SA01/dbt-tutorial/tree/main/dbt_tutorial?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[github.com](https://github.com/SA01/dbt-tutorial/tree/main/dbt_tutorial?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

**Setting up dbt project**

I’ll use the Microsoft Visual Studio Code for this demonstration. It can be downloaded from:

**[Visual Studio Code - Code Editing. Redefined](https://code.visualstudio.com/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Visual Studio Code is a code editor redefined and optimized for building and debugging modern web and cloud…](https://code.visualstudio.com/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[code.visualstudio.com](https://code.visualstudio.com/?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Next, create a folder with name of your choice, and open it in VSCode. Then, set-up a Python Virtual Environment. Oh, and if Python 3.10 is not present on the system, please install it. In VSCode, press Command + Shift + P on a mac (or Control + Shift + P on other systems) to open the Command Palette, and search Create Environment:

A blue and white rectangle

Description automatically generated

Command Palette — search for Create Environment

Next, select Venv:

A screenshot of a computer

Description automatically generated

select Venv

Next, select the installed version of Python 3.10.

A screenshot of a computer

Description automatically generated

Select Python 3.10

After this, it’ll take a few seconds to set-up the virtual environment, and once done, it will create the virtual environment folder called .venv:

A close-up of a logo

Description automatically generated

virtual Environment created

Next, add a requirements.txt file:

requirements.txt file

We will use two pip packages, dbt-core and dbt-postgres. The dbt-postgres is the package to connect to and work with PostgreSQL instance. Next, open the terminal in VSCode, and run pip install -r requirements.txt

Before running the above command, make sure it is connected to the virtual environment. If not, run the command source venv/bin/activate.

A screenshot of a computer

Description automatically generated

Open new terminal in VS Code

Next, run pip install -r requirements.txt to install the packages. Then, initialize the dbt project by running dbt init. And then, run through the initialization options:

A screenshot of a computer

Description automatically generated

Initializing dbt project

Project name is dbt\_tutorial, and we have selected 1 to use the PostgreSQL database.

This process creates the profile.yml file in the .dbt folder in the home directory, which is used to configure the connection properties. The process of creating a new dbt project creates a template that can be filled with connection properties to connect to the database. Also, it has connections for various environments, we will only provide credentials for the development (dev) environment:

A screenshot of a computer program

Description automatically generated

Auto-generated profile.yml file

Next, fill the parameters to connect to the Adventure Works db that was initiated earlier:

A screenshot of a computer program

Description automatically generated

profile.ymp file with parameters to connect with PostgreSQL

The schema defined here, called warehouse is important because all the artifacts created by running this project (in the dev environment) will be created in the warehouse schema in the database.

After this, we are ready to start defining models!

**First Model — Sales Order**

The aforementioned steps also creates the folder structure inside the directory with the name given to the project, dbt\_tutorial in our case. Additionally, the dbt\_project.yml file gets created which defines the project. To create the model, the default path (defined inside the dbt\_project.yml file and can be changed if needed) is dbt\_tutorial/models folder. Inside the models folder, create a folder called warehouse, and inside it, create a file called source\_adventureworks.yml. This defines the source, and the tables which will be referred to in this tutorial:

Defining data sources

Please note that any table which needs to be referred to using Jinja constructs need to be defined here.

Next, lets define the model warehouse.sales\_order\_header. Create a new file inside the models/warehouse folder with name sales\_order\_header.sql, and define the columns that will be included in the model. Please note that we are defining a simple model that only selects a few fields from one table and create it at a different schema. It is possible to create complex models using multiple tables, CTEs etc. The model built to define sales\_order\_header is:

Sales order header model SQL

Here, some columns are being selected from the sales.salesorderheader table, and one column, rowguid is being renamed as row\_id. The Jinja syntax source('sales', 'salesorderheader') refers to the source previously defined. The documentation states that it

Creates dependencies between a source and the current model, which is useful for documentation and model selection

And takes the following arguments:

*source\_name: The name: defined under a sources: key*

*table\_name: The name: defined under a tables: key*

Link to documentation:

**[About source function | dbt Developer Hub](https://docs.getdbt.com/reference/dbt-jinja-functions/source?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Read this guide to understand the source Jinja function in dbt.](https://docs.getdbt.com/reference/dbt-jinja-functions/source?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/reference/dbt-jinja-functions/source?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

When compiled, this will be replaced with the source and table name. Next, lets run the model. In the terminal, type

running the first model

to run the model.

A close-up of a computer screen

Description automatically generated

Console output after running the first model

A successful run, but the logs show that a view has been created. Also, there is a new schema called warehouse in the Adventure Works database, but there is no table, only a view has been created:

A screenshot of a computer

Description automatically generated

Model is materialized as view after execution

By default, the models are set to materialize as views, and this has been defined in the dbt\_project.yml file that was generated by dbt when the project was created. To materialize as tables, we need to change the section in the dbt\_project.yml file from:

default materialization

to:

Changing materialization to table

This will materialize all models as tables. Run with the command dbt run --full-refresh:

After the run, we can see that the view has been dropped and the table has been created:

A screenshot of a computer

Description automatically generated

Sales Order Header model materialized as table

A screenshot of a computer

Description automatically generated

Data in warehouse.sales\_order\_header table

If the requirement is to materialize some models as tables, and some as views, it can be implemented by specifying the materialized configuration for each folder like:

Changing materialization of models

The above configuration will materialize all models inside the folder staging as views, and warehouse as tables. Additionally, this configuration can be defined for each model in the SQL file to set it for individual model. In addition to table and view, the models can be materialized as ephemeral and incremental. Please refer to the documentation:

**[Materializations | dbt Developer Hub](https://docs.getdbt.com/docs/build/materializations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Read this tutorial to learn how to use materializations when building in dbt.](https://docs.getdbt.com/docs/build/materializations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/docs/build/materializations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

The dbt run first compiles the project to generate the SQL that it will execute on the target database. The generated files are present inside the target folder (dbt\_tutorial/target). Lets examine the SQL for creating and populating the sales\_order\_header table:

A screenshot of a computer

Description automatically generated

SQL generated by dbt to create the sales\_order\_header table

This is the SQL that gets executed on the database. As mentioned before, the Jinja templating referring to the source is replaced with the actual source table reference.

**Defining tests**

One of the major capabilities that dbt provides is to define tests and validations on columns and models. There is a large number of built-in tests, and more can be added by using external packages, or by writing custom tests. This requires creating a model yml file for the model. Inside the models/warehouse folder, add a new file with the name sales\_order\_header.yml and define each column for the sales\_order\_header model as:

Sales order header model configuration YAML file

Here, we have added definitions for all the columns of the model, and have also addd descriptions for some columns. This will add documentation to the models of the column, and will serve as a nice reference. If needed, this documentation can be carried over to the target database, and this can be done by adding the persist\_docs configuration to dbt\_project.yml:

Persisting documentation to database

The persisted table and column descriptions are inserted into the database, and in PostgreSQL they are added as comments:

A screenshot of a computer

Description automatically generated

Persisted documentation as comments in the database

Link to documentation:

**[persist\_docs | dbt Developer Hub](https://docs.getdbt.com/reference/resource-configs/persist_docs?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Persist\_docs - Read this in-depth guide to learn about configurations in dbt.](https://docs.getdbt.com/reference/resource-configs/persist_docs?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/reference/resource-configs/persist_docs?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Next, there are some tests defined on the row\_id column in the yml file:

Defining not\_null and unique tests

The not\_null and unique tests are among the tests that have been built-into dbt. They have been added to the row\_id column because we need to make it the key, and therefore, enforce uniqueness and not null.

Additionally, the accepted\_values test has been added to the status column, with 1, 2, 3 and 4 as accepted values for the column:

Accepted values test

This test takes a values parameter containing the accepted values for the column. If they are strings, they need to be put in quotes. Additionally, please note that this test will fail, and this is intentional for demonstration. Next, lets run the tests. Execute dbt test.

We get 2 passed tests and 1 failed test in the run logs:

A close-up of a computer screen

Description automatically generated

Results of tests

The logs mention the generated SQL file for the test, which is present together with other tests at the path target/compiled/dbt\_tutorial/models/warehouse/sales\_order\_header.yml/ :

Lets open the not null test at target/compiled/dbt\_tutorial/models/warehouse/sales\_order\_header.yml/not\_null\_sales\_order\_header\_row\_id.sql:

Test compiled as SQL

And now the failed test:

accepted\_values test SQL

The SQL is directly executable on the table, and all the tests fail if they return a row. If the first test is executed, there is no row returned. But for the failed test, rows are returned:

A screenshot of a computer

Description automatically generated

Executing the failed accepted\_values test SQL to examine the cause of failure

This is how column level tests can be used in dbt. Lastly, lets remove the accepted\_values test, and re-build the data with the command dbt run — full-refresh.

A few things to remember:

* In addition to column level tests, tests can be run at the model level. This can be done by adding the test under model in the .yml file, and is covered in the next section.
* The model .yml file can have any name, and can also contain multiple models in a single file. The important this is that the mode name for each model should match with the name of the model SQL file.

For documentation about testing in dbt, please visit the link:

**[Add tests to your DAG | dbt Developer Hub](https://docs.getdbt.com/docs/build/tests?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Read this tutorial to learn how to use tests when building in dbt.](https://docs.getdbt.com/docs/build/tests?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/docs/build/tests?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Additionally, dbt-expectations is a package inspired by Great Expectations, and contains many additional tests that can be added to the the dbt project and used for validation and to ensure data quality. Please visit

**[GitHub - calogica/dbt-expectations: Port(ish) of Great Expectations to dbt test macros](https://github.com/calogica/dbt-expectations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Port(ish) of Great Expectations to dbt test macros - GitHub - calogica/dbt-expectations: Port(ish) of Great…](https://github.com/calogica/dbt-expectations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[github.com](https://github.com/calogica/dbt-expectations?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

for more details.

**Advanced Testing — Adding custom tests**

Creating custom tests is supported in dbt. The way tests work is that they look for failing rows. For example, the not null tests looks for values where the column being tested is null. If there are no rows that fail, or appear in the query result, the test passes. To create a custom test, the simplest way is to write a test in SQL that checks a condition. In dbt, this can be enhanced by referring to the model involved using Jinja to refer to a source or model. This is called a singular test. It needs to be added to the tests folder that is configured in the dbt\_project.yml file. In this tutorial, the configuration has not been modified, it is the tests folder in the dbt\_tutorial folder. Lets add a test to it, create an sql file with the name test\_due\_date\_before\_order\_date.sql:

Custom singular test

After adding, this test along with other tests can be executed by running dbt test command. We can see that it passes. Its compiled form is also present inside the target folder.

The above test, although written specifically for the orderdate and duedate columns, should be a generic test that takes the column to check as parameter. Implementation of such generic tests that take model, column(s) and other properties as parameters is supported in dbt. This is called generic tests, and they should reside inside the generic folder within the configured tests folder. The generic test uses Jinja syntax to define the test as a function, and that function can later be re-used.

Create the generic folder inside tests folder, and inside it, create a file with the name test\_greater\_than\_column.sql, and define the test as:

Custom generic test

In the above definition, the first line defines the test with the name greater\_than\_column. This is the Jinja syntax for defining the test as a function with parameters. This test can then be used in the model .yml files, either at model level or column level. When used at a column level, the values of first two parameters: model and column\_name are passed by dbt, and only the value of the last parameter needs to be passed to the test. It can be used for validating the duedate column in the sales\_order\_header.yml file:

Using the custom generic greater\_than\_column test

When dbt test command is run, this test is also executed in addition to the other tests:

A screenshot of a computer program

Description automatically generated

Executing the custom generic test

The compiled form for the test is available at path dbt\_tutorial/target/compiled/dbt\_tutorial/models/warehouse/sales\_order\_header.yml/greater\_than\_column\_sales\_order\_header\_duedate\_\_orderdate.sql after running the dbt test command and it contains the following SQL:

Compiled SQL of the custom greater\_than\_column test for the sales\_order\_header model

As mentioned before, generic tests can also be used at model level, and this test can be called alternatively as:

Calling the test at the model level in the configuration YAML file

In this case, both column parameters need to be passed, whereas model is passed by dbt.

To conclude, testing and validation are essential components of any data pipeline, and dbt provides a robust set of tools to ensure data quality. By default, failed tests fail the test run, but this behavior can be customized to generate a warning or fail only if the number of failed rows exceeds a certain threshold. Moreover, dbt allows for sending failed rows to a separate table for further assessment and debugging.

The testing features and capabilities demonstrated in this article are just the tip of the iceberg, and dbt offers much more in terms of ensuring data quality and reliability.

**Building simple star schema: existing models, relationship tests, and ephemeral persistence**

After getting a basic understanding of how dbt transformations are put together, lets build a fact table (the fact table in star schema) using two source tables: sales.salesorderheader and sales.salesorderdetail.

**Creating sales\_order\_detail relation**

For demonstration, like sales.salesorderheader, an intermediate model will first be created from the sales.salesorderdetail table, called warehouse.sales\_order\_detail. At this point, there is no new concept, so following is the code of sales\_order\_detail.sql file and sales\_order\_detail.yml file:

Sales Order Detail model

And the configuration YAML file is:

Sales Order Detail configuration YAML file

The table is already added as a source in the source\_adventureworks.yml file at the beginning of this tutorial, so it can be referred in model definitions. After running the model with command dbt run, the table gets created. Next, the dimension tables.

**Creating product dimension — constraints**

At the beginning, one more data source was added to the sources file (source\_adventureworks.yml): product. The product table is present in the production schema, and therefore requires adding a new source to point to the schema production:

Using the other source — production to read the product table in the source\_adventureworks.yml file

To create dimension tables from production, the same process needs to be followed:

* Define the model file: dim\_product.sql.
* Define the associated YAML file dim\_product.yml, add tests and comments as required.

While defining the dim\_prodct model and configuration, constraint will be added to add the primary key constraint on the materialized dim\_product table. Constraints is a relatively new feature in dbt, introduced in version 1.5. It is part of Contracts, also a new feature introduced in dbt 1.5. Contracts are used to define configuration that is enforced by dbt:

When the contract configuration is enforced, dbt will ensure that your model's returned dataset exactly matches the attributes you have defined in yaml:

*\**name*and*data\_type*for every column*

*\* Additional*constraints*, as supported for this materialization and data platform*

**[contract | dbt Developer Hub](https://docs.getdbt.com/reference/resource-configs/contract?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[Read this guide to understand the contract configuration in dbt.](https://docs.getdbt.com/reference/resource-configs/contract?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/reference/resource-configs/contract?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Contracts ensure that the model when re-used later, have clearly defined data types and other constraints, and they are enforced before models are materialized. Link to dbt documentation on constraints is:

**[constraints | dbt Developer Hub](https://docs.getdbt.com/reference/resource-properties/constraints?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[This functionality is new in v1.5.](https://docs.getdbt.com/reference/resource-properties/constraints?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[docs.getdbt.com](https://docs.getdbt.com/reference/resource-properties/constraints?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

The code to define the aforementioned dimension model is:

dim\_product.sql file

dim\_product.yml file

In the above YAML configuration file, at the beginning, contract have been set to be enforced, and each column has a data type. Additionally the primary key column productid has two constraints: not\_null and primary\_key.

To execute, run the dbt run --full-refresh command. The table gets created in the database. Additionally, due to the added data types and primary key constraint, the table has data types that we specified, and also a primary key column:

A screenshot of a computer

Description automatically generated

Custom datatypes defined in the contract sent to the database

A screenshot of a computer

Description automatically generated

Primary Key constraint applied to the table by dbt

**Creating fact\_sales fact table**

Last, lets create the fact table called fact\_sales. The steps required are:

* Join the two models warehouse.sales\_order\_header and warehouse.sales\_order\_detail using the salesorderid column. This time, instead of directly referring the source, the intermediate models will be referred. Additionally, there is no need to persist the intermediate models. The models can exist and not get materialized if their materialization is set to ephemeral. Because ephemeral models do not exist as a views or tables, when referred in other models, their code is inserted as CTE at the time of materialization of those dependent models. For additional details about the types of materializations, please refer to the documentation <https://docs.getdbt.com/docs/build/materializations>.
* Create test to validate the foreign key relationship to ensure the keys present in the fact table are all present in the product dimension.

We have already set materialization of all models in the models folder to materialize as tables. To alter this configuration for warehouse.sales\_order\_header and warehouse.sales\_order\_detail tables, one way is to add materialization configuration in the respective SQL files in Jinja. Remember that this configuration, if present, takes precedent over the configuration in the dbt\_project.yml configuration file. Add the following in the first line of both sales\_order\_header.sql and sales\_order\_detail.sql files:

{{ config(materialized=’ephemeral’) }}

And then define the fact\_sales.sql file as:

fact\_sales.sql file

In this model definition, the sources are not the sources defined in the source\_adventureworks.yml file for this project. Instead, the sources of this fact table are existing models, i.e. sales\_order\_header and sales\_order\_detail models that were created before. Additionally, when this model gets compiled, because sales\_order\_header and sales\_order\_detail have been defined as ephemeral models, their respective SQL will be added to the compiled definition of this fact\_sales model.

Following is the fact\_sales.yml configuration file:

fact\_sales.yml file

An additional test has been added to the productid column that validates the foreign key relationship with the productid column in the dim\_product model.

To see all of this in action, run the commands:

* dbt clean
* dbt build

The command dbt build is equivalent to running dbt run followed by dbt test command. After running the command, only the fact\_sales table gets created in the database inside the warehouse schema. After setting materialization to ephemeral, the sales\_order\_header and sales\_order\_detail tables no longer get created:

A screenshot of a computer

Description automatically generated

Tables set to materialize as ephemeral are not created in the database

**Note:** If the sales\_order\_header and sales\_order\_detail tables remain visible, remove the warehouse schema, and then run dbt-run command again.

Next, open the dbt\_tutorial/target/run/dbt\_tutorial/models/warehouse/fact\_sales.sql file to examine the SQL that has been executed to materialize the fact\_sales table. It is:

Compiled SQL to create the fact\_sales table

Both ephemeral tables are not materialized, instead, dbt added their SQL definition to the code as CTEs. Next, open the foreign key test, it should be in the file dbt\_tutorial/target/run/dbt\_tutorial/models/warehouse/fact\_sales.yml/relationships\_fact\_sales\_productid\_\_productid\_\_ref\_dim\_product\_.sql. The test is compiled to SQL as:

Compiled SQL of the relationship test

This checks if all the values of the fields in the fact\_sales model are present in dim\_product.

This is the end of this section, in which a simple star schema with one dimension and one fact table was created in dbt, with tests and constraints. I hope this will serve as an adequate beginners’ introduction to dbt, to experience its capabilities first hand, and also to determine if dbt is an appropriate tool for task at hand or not. Lastly, lets package this project to run in a docker image.

**Dockerizing the dbt project**

After completing the implementation, the final step is to prepare it to execute in a production environment. Since dbt is used to implement transformation steps, it is commonly used in data pipelines, and container image is a common type of package that executes in many environments. In this step, lets look into preparing a Docker image with the project. This can also be modified for use in CI/CD pipelines.

Before creating the Dockerfile, copy profile.yml file from ~/.dbt folder into the project folder at the same level as requirements.txt, and change the host from [localhost](<http://localhost>) to host.docker.internal. This is if you are using Docker for mac or Docker for Windows, the localhost of the host instance can be accessed on at host.docker.internal. Please refer to the following Stack Overflow answer for more details:

**[From inside of a Docker container, how do I connect to the localhost of the machine?](https://stackoverflow.com/a/24326540?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[I have a Nginx running inside a docker container. I have a MySql running on the host system. I want to connect to the…](https://stackoverflow.com/a/24326540?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[stackoverflow.com](https://stackoverflow.com/a/24326540?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

Then create an empty file with the name Dockerfile. The Dockerfile starts with the base image python:3.10.12-slim-bullseye, First is the usual apt-get update and apt-get upgrade. Next, it sets the directory /dbt\_tutorial\_project and both dbt directory and working directory. Next, the requirements.txt file is copied and requirements installed. Finally, the dbt\_tutorial directory is copied. The dbt project so far in this tutorial doesn’t install any external dbt packages, but if there are packages included, then the next step in the Dockerfile should be to install the dbt dependencies using the dbt deps command:

Dockerfile

Next, build the image. It can be given a tag as well. The command is docker build -t dbt-docker-img.

A screenshot of a computer

Description automatically generated

Building the dockerfile

After building, run the docker image with the following command:

docker run dbt-docker-img dbt build — profiles-dir ./dbt\_tutorial/ — project-dir ./dbt\_tutorial

The additional arguments provide the path of the profiles.yml directory, and the project directory containing the dbt-project.yml file.

A screenshot of a computer

Description automatically generated

Running the image

This concludes the introductory tutorial, which aims to stay as simple as possible and focused on dbt by avoiding getting sidetracked by tasks such as setting up accounts and other tools as much as possible. We constructed a basic star schema using the Adventure Works example database and packaged the project as a container image. However, this article only scratches the surface, no, only the top layer of the surface, of the vast array of capabilities of dbt.

**Further learning**

To continue your learning journey with dbt, there are several resources available:

* First, I recommend taking the “Mastering dbt (Data Build Tool) — From Beginner to Pro” course on Udemy. I personally learned a lot from this course, which covers many advanced topics, including creating and using incremental models, seeding data with seed files, snapshots, advanced configurations and execution commands, introduction to dbt cloud, and much more! You can enroll in the course here: <https://www.udemy.com/course/mastering-dbt-data-build-tool-bootcamp/>
* For more advanced data testing and validation, check out the dbt-expectations package, which allows you to create more complex tests and assertions in your dbt projects. Start with this excellent article:

**[Boost your dbt tests using Great Expectations in dbt](https://zoltanctoth.medium.com/boost-your-dbt-tests-using-great-expectations-in-dbt-1c2d33d53fb3?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[The Data Build Tool (dbt™) made a hit as the central component of the modern data stack in the past years. It has been…](https://zoltanctoth.medium.com/boost-your-dbt-tests-using-great-expectations-in-dbt-1c2d33d53fb3?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[zoltanctoth.medium.com](https://zoltanctoth.medium.com/boost-your-dbt-tests-using-great-expectations-in-dbt-1c2d33d53fb3?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

* Another recommended article covers constraints in dbt:

**[dbt Constraints: Automatic Primary Keys, Unique Keys, and Foreign Keys for Snowflake](https://medium.com/snowflake/dbt-constraints-automatic-primary-keys-unique-keys-and-foreign-keys-for-snowflake-d78cbfdec2f9?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)**

[dbt Constraints is a new package that generates database constraints based on the tests in a dbt project. Based on the…](https://medium.com/snowflake/dbt-constraints-automatic-primary-keys-unique-keys-and-foreign-keys-for-snowflake-d78cbfdec2f9?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

[medium.com](https://medium.com/snowflake/dbt-constraints-automatic-primary-keys-unique-keys-and-foreign-keys-for-snowflake-d78cbfdec2f9?source=post_page-----28e335be5f7e--------------------------------" \t "_blank)

* Finally, if you want to run dbt projects on Apache Airflow, read this article:

**[Running dbt with Airflow | Datafold](https://www.datafold.com/blog/running-dbt-with-airflow?source=post_page-----28e335be5f7e--------------------------------" \l "s4" \t "_blank)**

[How to setup and run dbt with airflow on your local machine.](https://www.datafold.com/blog/running-dbt-with-airflow?source=post_page-----28e335be5f7e--------------------------------" \l "s4" \t "_blank)

[www.datafold.com](https://www.datafold.com/blog/running-dbt-with-airflow?source=post_page-----28e335be5f7e--------------------------------" \l "s4" \t "_blank)

[Getting Started with dbt (Data Build Tool): A Beginner’s Guide to Building Data Transformations | by Suffyan Asad | Medium](https://medium.com/@suffyan.asad1/getting-started-with-dbt-data-build-tool-a-beginners-guide-to-building-data-transformations-28e335be5f7e)