dbt and BigQuery: an action oriented approach

Samuel Gachuhi Ngugi

2024-12-10

Contents

Chapter 1

About

Samuel Gachuhi is a geographer who by fate found himself in the programming world. He holds a certificate in data science and machine learning and another in deep learning with Tensorflow from Udemy. He was motivated to write this book on dbt after noticing that most text on dbt was written in a manner only comprehensible to software engineers. Believing that knowledge transfer should be conveyed in a manner that is understandable by all, he sought to write this book in a less technical manner, and infusing it with humour since learning should be enjoyable, not painful.

1.1 Resources

All the code used in this book has been uploaded to this Github repository. Here is the full link:

https://github.com/sammigachuhi/dbt_book_codes

The online version of the book is available from this link. The online book is better than the Portable Document Format (PDF) version as the latter has some pieces of code trimmed from the edge of the page.

https://bookdown.org/sammigachuhi/dbt_book/

1.2 Copyright

Figure 1.1: copyright

- Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

Chapter 2

Introduction

2.1 What is dbt?

dbt, when in full, stands for Data build tool. dbt is a tool that data scientists and analytical engineers use to transform data in their data warehouses. For now, just think of dbt much like a recipe. In a recipe, you have the instructions to cook your favourite meal, say a roasted chicken. Sure enough, your recipe will contain details on the optimal oven temperature, heating duration and setting the table! dbt works in much the same way. We define how we want to transform or build our data. Once we hit `run`, the magic happens.

2.2 Encounter with `dbt`

How did I cross paths with dbt? Coming from the geographical sciences, my first experience with dbt, contrary to the many positive revs from online users, wasn't so good. Perhaps it was because a lot was on my desk back then, for I was having trouble piecing together all the different components that make dbt work, and how it works. It is only after some time, and several hard knocks in between, that I was able to get a semblance of what it does.

Nevertheless, at least I got a few things. dbt could be used to create views of your tables in the data warehouse, it could perform tests and lastly, (the one I liked the most) it could be used to render your documentation!

2.3 dbt, from the professionals...

dbt, from the words of developers, is an open-source tool that analysts and data engineers use to transform data in their data warehouses. Data transformation

is the process of converting data from its source format to the format required for analysis. The data transformation process is part of a three stage process known as Extract, Load and Transform (ELT). Before ELT, Extract, Transform, and Load (ETL) was the king. The former involves transferring data from the source, to the destination, such as a data warehouse or data lake and performing the transformation in there. The latter, ETL, though a traditional approach, involves first identifying the data, transforming it prior to landing it to the destination, in this case a data warehouse.

Here is a better description of the Extract, Load and Transform keywords.

Extract - this is the identification and reading of data from one or more sources, such as databases, internet, comma separated value (csv) files and the like.

Load - just like you would pull up a weight into a lorry, this is the process of transferring data from the source to your data warehouse.

Transform - this is the conversion of data from its state to a format that can be used by downstream users.

You may have seen the term *data warehouse* coming up quite a number of times. A data warehouse is a data management system that stores current and historical data from multiple sources in a business friendly manner for easier insights and reporting. Examples of data warehouses are Google BigQuery, Snowflake, Amazon Redshift, Azure Synapse Analytics, IBM Db2 Warehouse and Firebolt.

2.4 Why use dbt?

If you work with data that needs to be version controlled, that is, it can be rolled back to a previous time, you need to work in dbt. If you want to standardize the data models created across teams, dbt is the tool of choice. If you also want a central place where your data work is also documented, dbt handles this quite well. In other words, dbt should be the swiss knife when working with large datasets and where you want to maintain modularity, order and documentation of your work.

The below image summarises the role of dbt in your data processing work.

Source: Reference

Figure 2.1: The role of dbt

Chapter 3

The `dbt` architecture

In my first time working with dbt, I was overwhelmed with its architecture. It felt like that individual who is sitting before that large screen in a nuclear power plant and in charge of all the controls. Nevertheless, if people can gain confidence in holding a nuclear power plant on their fingertips, then surely you can crack dbt.

The main components that make up dbt are as follows:

- models
- tests
- documentation
- sources

Let's go through each one.

3.1 Models

This is the component of dbt that you will most likely work with. In dbt, a model is simply a SQL statement. As simple as that. dbt will use the SQL statements to perform the transformations in your data warehouse that have been defined in your SQL statement. For example, say I want to create a new column of the table in my Google BigQuery. I will create a SQL statement that does just that. That SQL statement is what is referred to as a model in dbt.

Below is an example of a model that creates a table called `customers`. The model is saved as `customers.sql`.

```
with customer_orders as (
    select
        customer_id,
        min(order_date) as first_order_date,
        max(order_date) as most_recent_order_date,
        count(order_id) as number_of_orders

    from jaffle_shop.orders

    group by 1
)

select
    customers.customer_id,
    customers.first_name,
    customers.last_name,
    customer_orders.first_order_date,
    customer_orders.most_recent_order_date,
    coalesce(customer_orders.number_of_orders, 0) as number_of_orders

from jaffle_shop.customers

left join customer_orders using (customer_id)
```

3.2 Tests

"Do not put me to test", is a familiar statement from an impatient person. However, dbt allows us to test our data and see if it meets certain assertions. In other words, does our data meet the requirements that have been set for it?

dbt offers two ways to perform your tests:

1. generic, and,
2. custom tests.

Generic tests involve just using a pre-defined test that comes packaged in dbt. For example, for every field key you place in a YAML file in dbt, you can specify which kind of test to perform on that particular field from the following options: `unique`, `not_null`, `accepted_values` and `relationships`.

- `unique` - the values should be radically distinctive all through
- `not_null` - there shouldn't be a missing value in the particular column name in the table ***`accepted_values`*** - *only the values contained in the*

accepted values key will be considered valid. Anything outside of this will result in an error `relationships` - the values in this field can be referenced in a different column elsewhere in the table or on a different table altogether.

An example of a generic test is below:

```
version: 2

models:
  - name: orders
    columns:
      - name: order_id
        tests:
          - unique
          - not_null
      - name: status
        tests:
          - accepted_values:
              values: ['placed', 'shipped', 'completed', 'returned']
      - name: customer_id
        tests:
          - relationships:
              to: ref('customers')
              field: id
```

For custom tests, these involve one creating a SQL model and referencing it in a YAML file using Jinja template language.

For example, here is a custom test written in a SQL file called `transaction_limit_test.sql`.

```
-- tests/transaction_limit_test.sql
select user_id, sum(transaction_amount) as total_spent
from {{ ref('transactions') }}
group by user_id
having total_spent > 10000  -- Assuming the limit is 10,000
```

The test is referenced in a YAML file and called over a column called `transactions`.

```
models:
  - name: transactions
    tests:
      - transaction_limit_test
```

3.3 Documentation

Now, the favourite part of dbt, and possibly the easiest is documentation. Documentation is the description of various components of your data. To write a description of any piece of your data, the `description` key is used.

For example here is a description of a field called `event_id` inside a YAML file.

```
version: 2

models:
  - name: events
    description: This table contains clickstream events from the marketing website

    columns:
      - name: event_id
        description: The D-day is the Deed day
        tests:
          - unique
          - not_null
```

Documentation will be performed where you have placed your tests. There is also a more complex, but scalable manner of writing descriptions. It uses jinja template tags. It works well for large data where the descriptions are many or the descriptions are shared across several tables.

A short example of the jinja templates' documentation is shown below. The description is within a markdown file (`.md`) other than the one containing my field names. The descriptions will be like so:

```
{% docs table_events %}

I am not so very robust, but I'll do the best I can.

Some text here

1) and here
2) and here
3) and also here

{% enddocs %}
```

So when one returns to their YAML file, they will reference the particular field of interest with the above description like so:

```
version: 2

models:
  - name: events
    description: '{{ doc("table_events") }}'

    columns:
      - name: event_id
        description: The D-day is the Deed day
        tests:
            - unique
            - not_null
```

3.4 Sources

`sources` enable one query the data in your data warehouse. Once you specify the existing table in your data warehouse under the `sources` key, you can access every data from within this table using SQL. To work with a source table, you first have to wrap it inside a `{{ source(table-name) }}` jinja template. Below is an example of how to declare a source.

```
version: 2

sources:
  - name: jaffle_shop
    database: raw
    schema: jaffle_shop
    tables:
      - name: orders
      - name: customers

  - name: stripe
    tables:
      - name: payments
```

You can reference the above source inside a SQL model like so:

```
select
  ...

from {{ source('jaffle_shop', 'orders') }}

left join {{ source('jaffle_shop', 'customers') }} using (customer_id)
```

dbt will thereafter know that it will perform some operations using data from the `orders` and `customers` data from the `jaffle_shop` –the origin of all our data in this example.

Chapter 4

Data storage

As as been repeatedly mentioned, to the point of boredom, dbt transforms the data in your data warehouse. Now, before expanding the concept of a data warehouse, the following two are also terms you will hear mentioned quite often in the field of data engineering. They are data lake and data lakehouse.

4.1 Data warehouse

At the very beginning, when introduced to data engineering concepts with a test paper to boot in four weeks time, I thought that a data warehouse was some storage system akin to that found in Google Drive. I could have been partly right, but I was still far off the mark. A data warehouse is more than just a storage system. It is where data is not only stored but also queried, by means of SQL. It allows data from multiple sources such as internet of things, apps, from emails to social media and keeps a historical record of any changes. Examples of data warehouses are Snowflake, Google Big Query, Amazon Redshift and Azure Synapse Analytics.

The following are the components of a data warehouse.

1. Data sources - this refers to the origins of the data that lands in your data warehouse.

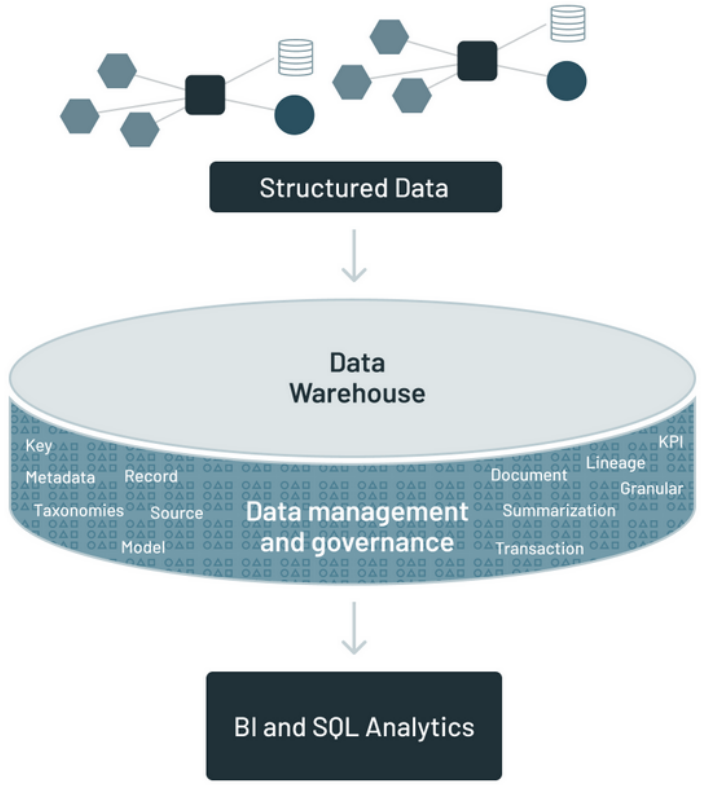
2. Extract, Transform and Load (ETL) Processes - these are the processes involved in extracting, transforming and loading the data into your data warehouse.

3. Data warehouse database - this is the central repository where the cleansed, integrated and historical data is stored.

4. Metadata repository - metadata is essentially data about data. Metadata will typically contain the source, usage, values and other features that comprise your data.

5. Access tools - imagine having to figure a way how to write a document in your computer without Microsoft Word. How hard would that be? Access tools are similar to Microsoft Word. They are the tools that enable a user to interact with the data. They include querying, reporting and visualization tools.

As you can see from above, a data warehouse is more than just a storage area for your data. It is like a whole community that will provide the services that you desire, so long as they are integrated into the data warehouse.

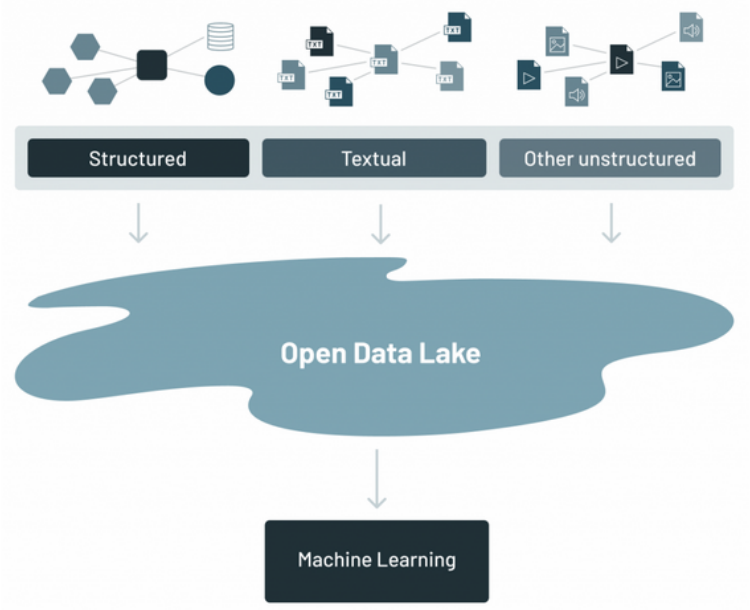


Source: Reference

4.2 Data lake

A data lake is a centralized repository that ingests and stores large volumes of data in its original form. Due to its open, scalable architecture, a data lake can store structured (database tables, excel sheets), semi-structured (xml, json and web pages) and unstructured data (images, audio, tweets) all in one place. Data in the data lake is stored in its original format.

So if data lakes and data warehouses store data, then what is the difference? For one, a data lake can store data of any type, so long as it falls within the three classes of structured, semi-structured and unstructured data. On the other hand, data warehouses deal with more standardized data. That is, data in a data warehouse has undergone some refinement of some kind to be in a structure that fits the organizations's goals.



Source: Reference

4.3 Data lakehouse

A data lakehouse is simply a hybrid of a data warehouse and data lake. It is like a product that combines the best of both worlds. A data lakehouse provides both scalability of large sums of data from the data lake and additionally, the application of a structural schema to data inherent in data warehouses.

Even with the above definition, it is still hard to decipher the advantage that a data lakehouse offers above that of a data warehouse. Apart from allowing the querying of unstructured data, storage costs are lower in a data lakehouse compared to a data warehouse.

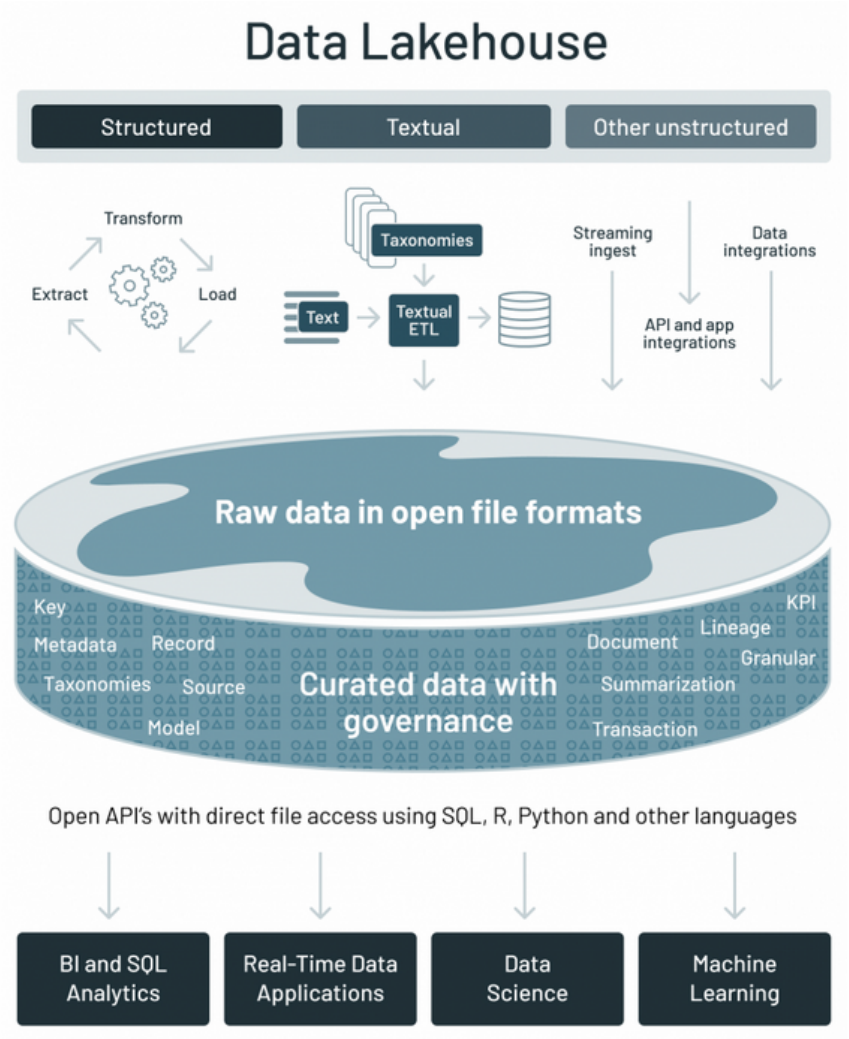


Figure 4.1: Data lakehouse

Source: Reference

4.4 A brief history

At the very beginning, companies used to rely on relational databases and these were sufficient. However, they became too numerous as the needs and services of companies grew. Therefore, experts decided to look for a way of how they could merge all these single databases into one repository which would hold everything while allowing for permission controls to who gets access to what. Believe it or not, the concept of the data warehouse began in the 1960s but it is in the 1980s and 1990s that interest in this topic really began to get traction. That's until the need for storing unstructured data from emails, images and audio began to grow. Data warehouses were not so efficient in storing this thanks to their strict schema enforcement (read, they store data in a structured format).

The beginning of 2000s saw the rise in the need to properly manage unstructured data with the growth of online platforms such as Google and Yahoo. Companies needed a way to store and retrieve unstructured data quickly and efficiently, which wasn't possible with data warehouses. Data lakes excelled in storing all sorts of raw data, from structured to unstructured and everything in between. If you read on the history of data lakes, you willquite often come across the word 'Hadoop'. Hadoop is the pioneer of the data lakes we use today.

However, despite being a good storage for any sort of data, the pesky question of maintaining some quality and order resurfaced again! How could we maintain some structure while allowing the data to be in any structure?!

From 2010s and onwards, after a decade of success with data lakes, companies wanted a better storage system from which to run their machine learning models but had the best capabilities of both a data warehouse and a data lake. Before lakehouses, companies would first ingest data into a data lake, then load into a data warehouse from where analytics would be done. But how could we just merge it into one place where storage and analytics could happen? This is how the data lakehouse concept came to be. Data lakehouses provide the following benefits:

1. ACID (Atomicity, Consistency, Isolation and Durability) transactions - ACID transactions promote integrity during data transfer.
2. Delta lake - initially developed by the Databricks team, this is a layer on top of your data in the data lake that provides a schema, keeps a record of changes in your data (versioning) and stores metadata.
3. Machine learning support - because a data lakehouse can store more data types than the data warehouse, it is a better place to perform machine learning modeling.

For more information on the evolution of data storage systems, this is a definitive guide.

	Data warehouse	Data lake	Data lakehouse
Data format	Closed, proprietary format	Open format	Open format
Types of data	Structured data, with limited support for semi-structured data	All types: Structured data, semi-structured data, textual data, unstructured (raw) data	All types: Structured data, semi-structured data, textual data, unstructured (raw) data
Data access	SQL-only, no direct access to file	Open APIs for direct access to files with SQL, R, Python and other languages	Open APIs for direct access to files with SQL, R, Python and other languages
Reliability	High quality, reliable data with ACID transactions	Low quality, data swamp	High quality, reliable data with ACID transactions
Governance and security	Fine-grained security and governance for row/ columnar level for tables	Poor governance as security needs to be applied to files	Fine-grained security and governance for row/columnar level for tables
Performance	High	Low	High
Scalability	Scaling becomes exponentially more expensive	Scales to hold any amount of data at low cost, regardless of type	Scales to hold any amount of data at low cost, regardless of type
Use case support	Limited to BI, SQL applications and decision support	Limited to machine learning	One data architecture for BI, SQL and machine learning

Source: Reference

Chapter 5

Our data in BigQuery

In an earlier chapter, we saw that in data engineering data mainly goes through three processes: extract, load and transform (ELT). The Extract, Transform, Load (ELT) is more of a traditional approach and we will not use it in this case. We will be using Google Bigquery as our data warehouse when working with dbt.

As a reminder, let's go through the definitions of ELT.

Extract - the process of identifying and reading data from one or more source systems. We won't have to do this since the New York City (NYC) bikes data that will be using has already been *extracted* from its source by BigQuery creators.

Load - the process of adding the extracted data to the data warehouse, –in this case Google BigQuery. Again, Google has done this for us. Therefore we won't have to do it.

Transform - the process of converting data from its raw format to the format that it will be used for analysis. This falls definitely within our forte. And we shall use dbt for this. Examples of data transformations that can be done with SQL modeling in dbt are:

- Replacing codes with values
- Aggregating numerical sums
- Applying mathematical functions (SQL can do some maths too, but can be very verbose here)
- Converting data types
- Modifying text strings
- Combining data from different tables and databases.

5.1 Accessing Big Query

BigQuery is a data warehouse provided by Google.

To access it, open an incognito window and go to this link. Sign in using your gmail account.

Click on **Console** button at the top right. That step of bravery will take you to an interface that looks like this:

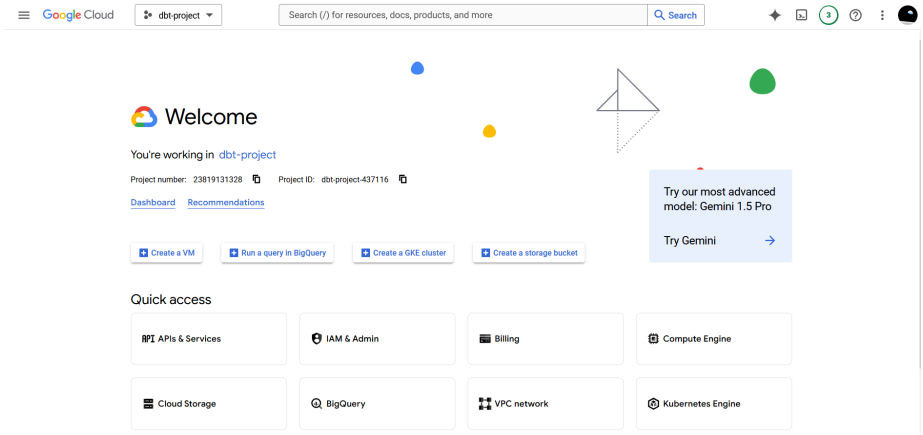


Figure 5.1: GCP Interface

Click on the dropdown at the top. Select **NEW PROJECT**. We want to create a new project that will contain some tables that we will work with in dbt.

Name your project as `dbt-project1` or any other name you prefer. Then select **CREATE**.

Once the project has been created, you will be returned to the original page as at first. However, when you select the project dropdown again, you should see your newly created project as one of the options.

Click on your project, the interface will refresh and the dropdown should now reflect `dbt-project1`.

Click on the **Dashboard** link on the homepage.

The below interface should appear. It can seem overwhelming at first. Lots of things in one place.

In one of the "boxes" within the **Dashboard** tab, you will find one called **Resources** with the **BigQuery** button underneath. Click on this button. It will take you to a page asking you to *Enable the BigQuery Application Programming Interface (API)*. Do comply!

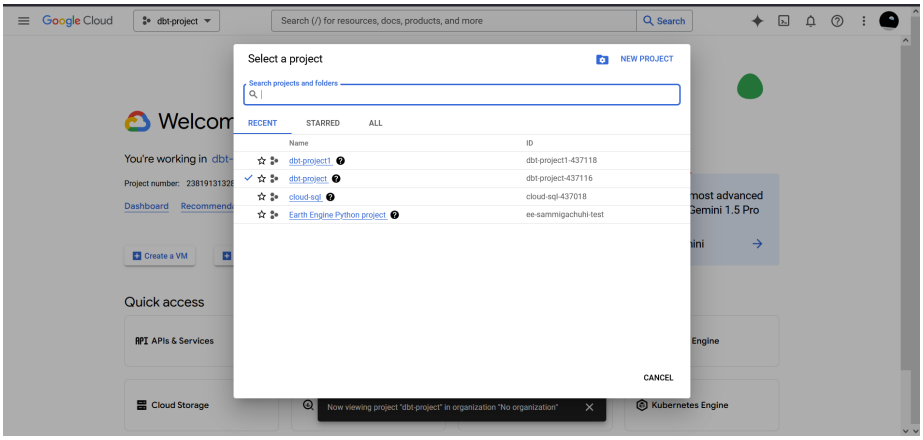


Figure 5.2: GCP Project

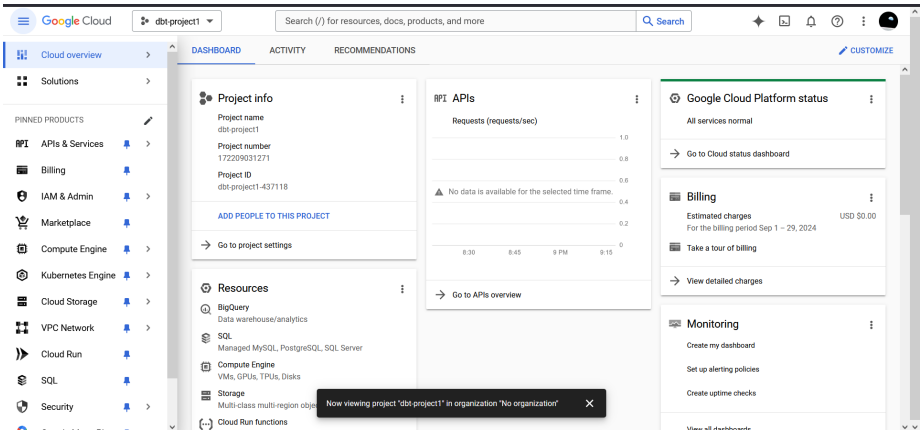


Figure 5.3: Dashboard

Behold, below is the BigQuery interface.

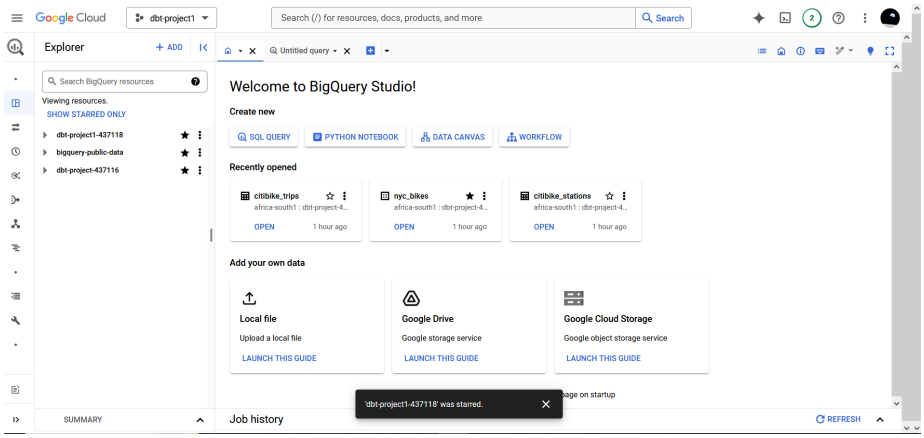


Figure 5.4: BigQuery interface

You will see one of the resources as `dbt_project1<some-random-number>` in case you had other resources. Star this project for quick access in future.

5.2 Copying the New York City Bikes data

One of the datasets we will be working with is the "New York City Bikes dataset". To access it, click on the **ADD** button. A sidebar will open up. Go to **Public Datasets**.

In the **Search Marketplace** searchbar, type 'bikes'.

Click on the NYC Citi Bike Trips tab. A new sidebar will popup with a button of **View Dataset**. Click this button and the Google Cloud Platform (GCP) Dashboard will reappear but this time round the `bigquery-public-data` resource will appear.

Click on this particular resource's dropdown on the left and scroll down to the `new_york_citibike` dataset. We want to copy this dataset from that of `bigquery-public-data` to that of `dbt_project1-437718`. The random numbers will be different in your case.

Scroll up again to your `dbt_project1` resource. On the kebab menu on the right of this resource, select *Create Dataset*.

A new sidebar will open. Insert the following for each parameter:

- Dataset ID - `nyc_bikes`
- Location type - `Region`

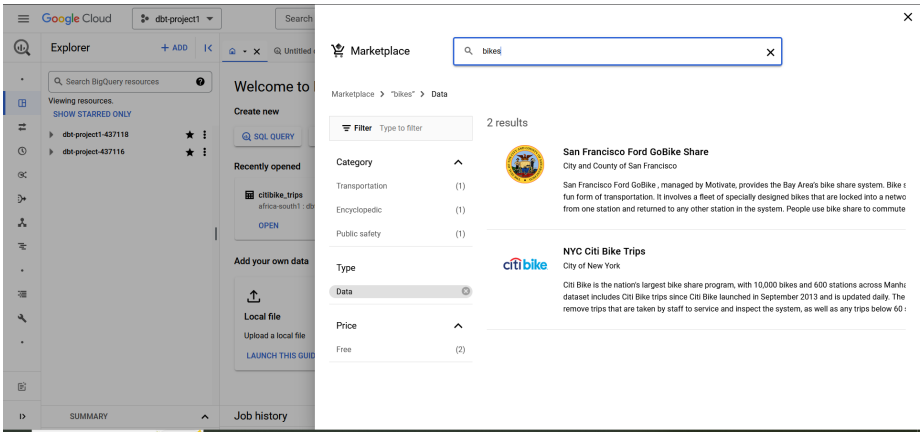


Figure 5.5: Marketplace

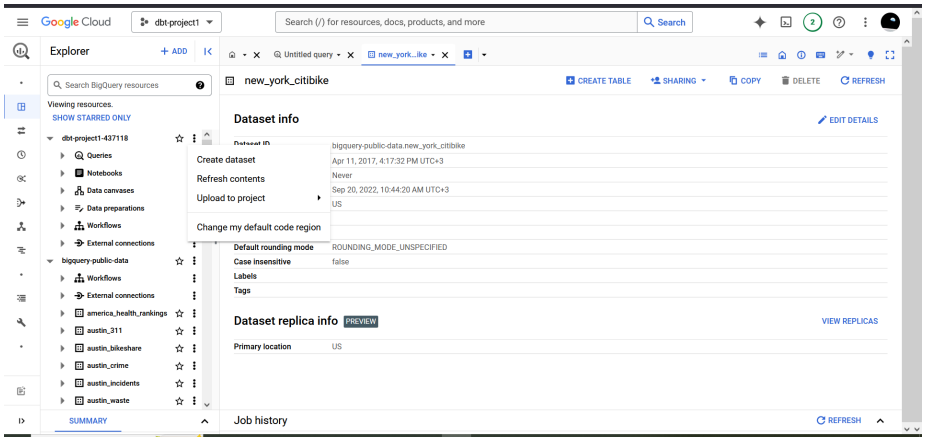


Figure 5.6: Create dataset

- Region - `africa-south1 (Johannesburg)` or your preferred region

Thereafter, click on **CREATE DATASET**.

The `nyc_bikes` dataset should now appear under the `dbt-project1` resource. We want to copy the contents of the `new_york_citibike` dataset into our `nyc_bikes` dataset. So how do we proceed?

Scroll down to the `new_york_citibike` dataset under the `bigquery-public-data` resource and click on it. On the menu for this dataset, you will see the **Copy** button. Click on this button.

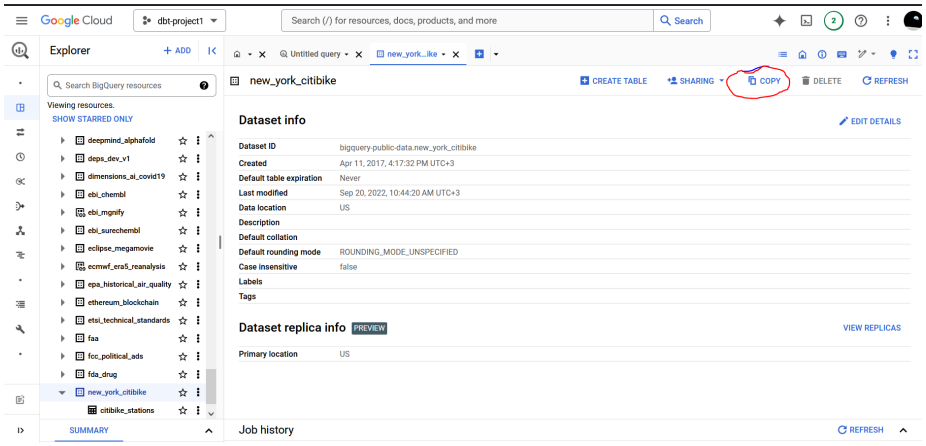


Figure 5.7: Copy dataset

In the **Destination** searchbar, type `nyc_bikes` in reference to where we want to copy the contents into. You may need to enable the data transfer API to perform the copy operation. Do so if BigQuery necessitates that *it* must be enabled.

Once you copy the dataset, a small bar will appear on the screen saying **View Details**. Click on it to stop the run operation since BigQuery will be rerunning the copy operation after every 24 hours. Disable the transfer process and delete it.

Going back to your `dbt_project1` resource, your `nyc_bikes` dataset should now be having two tables under it. That is:

- citibike_stations
- citibike_trips

Click on any of the tables and preview the data therein using the **PREVIEW** button of each tables' interface.

Congratulations on loading your first table in BigQuery!

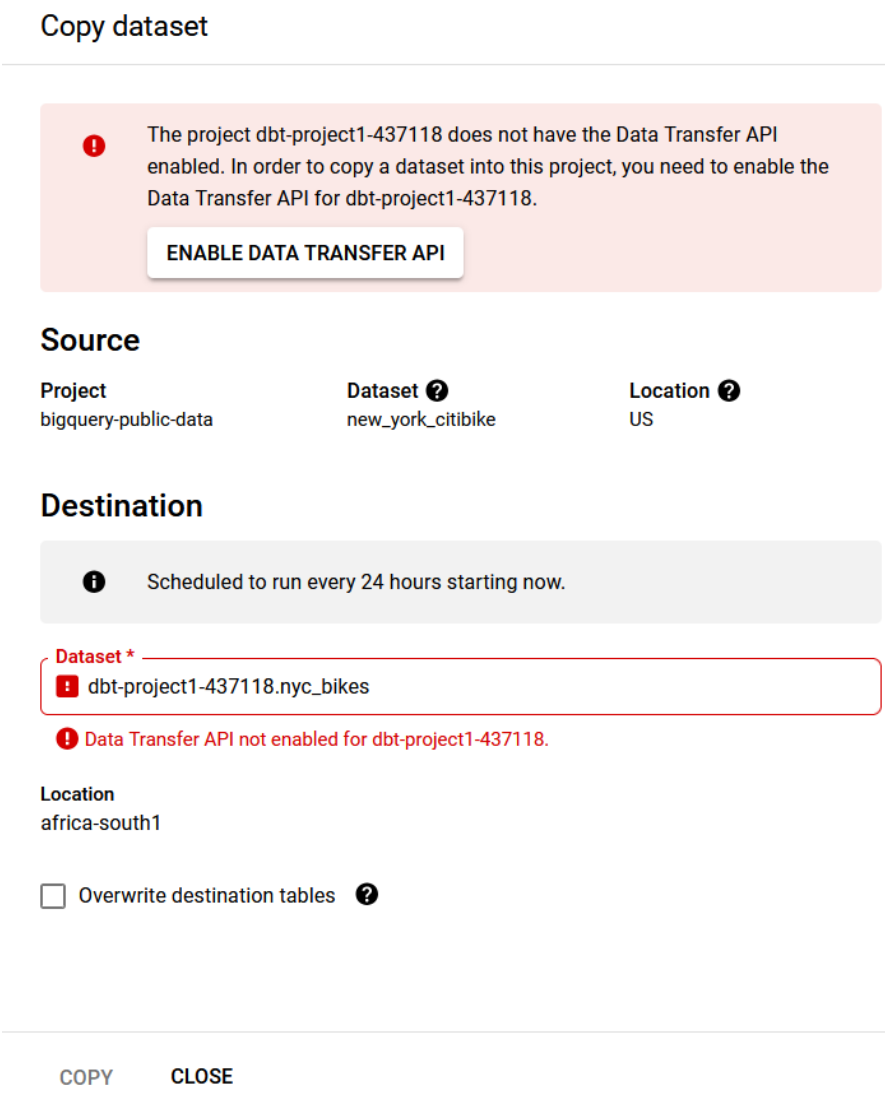


Figure 5.8: Copy sidebar

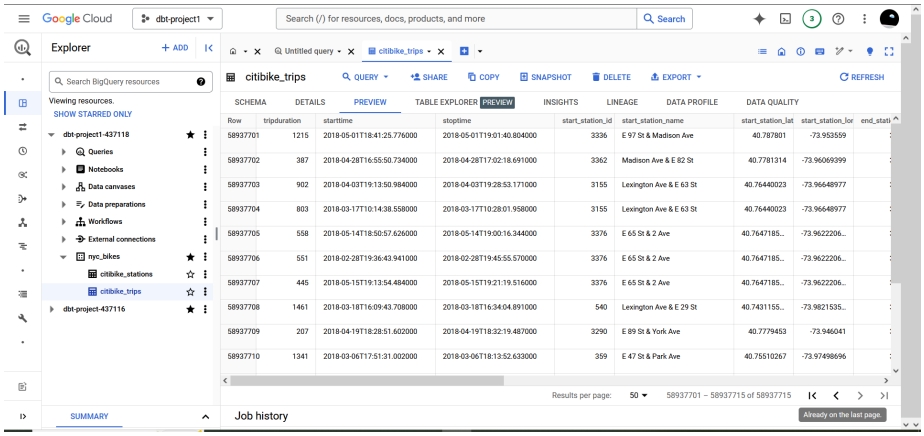


Figure 5.9: Table

Chapter 6

Installing dbt

Now that we've seen how to access a dataset inside our data warehouse, now let's proceed to installing dbt. This section assumes you already have Visual Studio (VS) Code installed. All code in this book has been operated within a Linux environment. For Window's users, a Linux environment can be enabled by installing the Windows Subsystem for Linux (WSL2) virtualization platform.

6.1 Setting the environment

Open your VS Code.

Create a new folder called `dbt_book`. Move into this directory in your VS Code by typing `cd dbt_book/` in your terminal.

The first thing we shall do is create a virtual environment from which we shall conduct all our dbt operations. A virtual environment is useful in preventing conflicts between packages across your various programming projects.

```
python3 -m venv venv
```

The first `venv` tells python that you're creating a virtual environment while the second refers to the name of the virtual environment. In this case, our virtual environment shall still share the name `venv`.

Now let's activate our virtual environment.

```
source venv/bin/activate
```

You will see your namespace appended with `venv` which means that your virtual environment is now active. For example:

```
(venv) sammigachuhi@Gachuhi:~/dbt_book$
```

Now here comes the big part: installing dbt for Big Query. The following code will install everything we need; both dbt and the dependencies needed to connect it to BigQuery.

```
python3 -m pip install dbt-core dbt-bigquery
```

6.2 Connecting to your BigQuery data warehouse

We wish connecting to a data warehouse for dbt were as easy as providing a username and password. Far from it, but it is definitely possible. To connect dbt to a data warehouse, we use a keyfile. A keyfile is a file that contains encryption keys or licenses for a particular task. The keyfile we shall use shall be the doorway to our data warehouse.

As the first step, go to your GCP Credentials Wizard page.

Ensure that your project is set to the dbt project you created in the previous chapter. For my case, I reverted to an earlier created project called `dbt_project` since my other project `dbt_project1` started incurring costs.

For **Credential Type**:

- From the **Select an API** dropdown, choose **BigQuery API**
- Select **Application data** for the type of data you will be accessing
- Click **Next** to create a new service account.

In the service account page:

- Type `dbt-book` as the Service account name or any other name you prefer.
- From the **Select a role** dropdown, choose **BigQuery Job User** and **BigQuery Data Editor** roles and click **Continue**
- Leave the **Grant users access to this service account** fields blank
- Once everything is fine, it is as good as clicking **Done**!

Your credentials interface will look like this:

Click on your service account name.

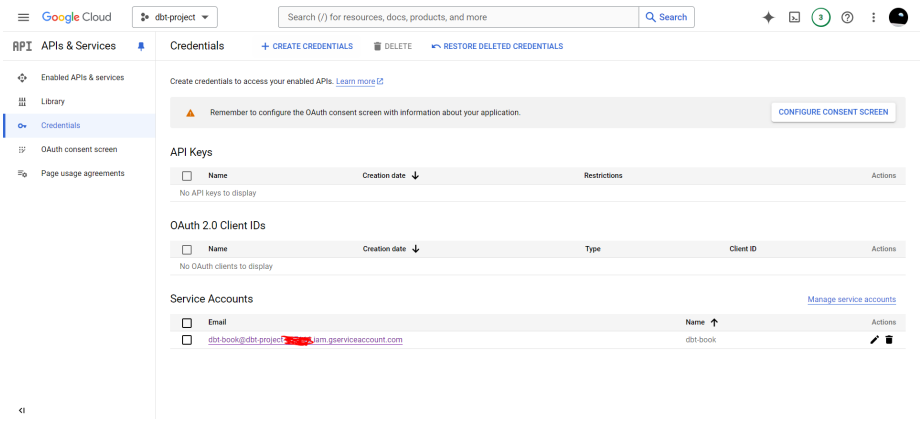


Figure 6.1: Service account

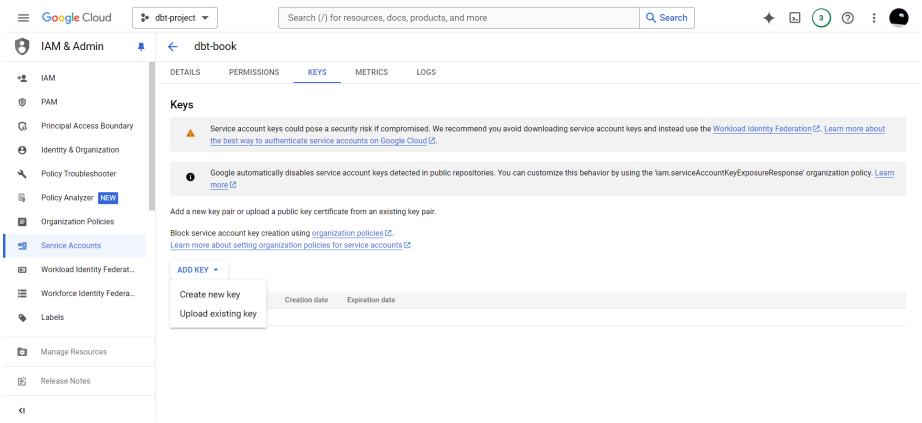


Figure 6.2: Add key

Click on the **KEYS** tab. We want to create a key that dbt will use to connect to our data warehouse.

Click on **ADD KEY>Create new key**.

Select **JSON** on the interface that appears and click **CREATE**.

This will download a json file containing the encryption keys that dbt will use to connect to your data warehouse.

Store this json file in a safe place.

6.3 Initializing a dbt project

To create a dbt project, we run the open sesame key: `dbt init`.

It will create a string of outputs. It's important to key in the right details if you want to create a dbt project.

The first output will ask for the name of your dbt project. Insert `dbt_book` or any other name you prefer.

```
19:07:31  Running with dbt=1.8.7
Enter a name for your project (letters, digits, underscore): dbt_book
```

If you had an already pre-existing dbt project with the same name, dbt will ask if it can overwrite that project. Type `y` if you wish to do so.

dbt will thereafter ask you which database you would like to use. Since we had installed dbt with the package dependancies for BigQuery, you will see the sole option for BigQuery. Type `1` to select BigQuery.

```
Which database would you like to use?
[1] bigquery

(Don't see the one you want? https://docs.getdbt.com/docs/available-adapters)

Enter a number: 1
```

You will thereafter be asked the authentication method you would like to use. Since we had already created a service account and downloaded the JSON file containing the encryption keys, we shall select option 2.

```
Enter a number: 1
[1] oauth
[2] service_account
Desired authentication method option (enter a number): 2
```

For the keyfile, provide the path to where you had saved the json file. As a note, this path should be somewhere different than where your dbt project is located. Github will set off alarms and send alert emails in case it finds the keyfile within your repository. Keyfiles are never meant to be shared or stored somewhere accessible. They are considered sensitive information.

```
keyfile (/path/to/bigquery/keyfile.json): /home/sammigachuhi/dbt_credentials/dbt_book.json
```

You will also be asked to provide your project ID. This is available under your dbt project's dashboard under the **Project ID** heading.

```
project (GCP project id): dbt-project-437116
```

For the dataset name, we will use `nyc_bikes` which is the dataset we want to conduct our dbt operations on.

```
dataset (the name of your dbt dataset): nyc_bikes
```

For the rest of the options, you can fill them as below:

```
threads (1 or more): 1
job_execution_timeout_seconds [300]:
[1] US
[2] EU
Desired location option (enter a number): 1
19:09:24  Profile dbt_book written to /home/sammigachuhi/.dbt/profiles.yml using target's profile
```

Now, in order to test whether your dbt installation is correct, you will have to change directory (cd) into your `dbt_book` subfolder we created as part of the `dbt init` prompts. At first, we had created a directory called `dbt_book` in which we also activated the virtual environment. When we ran `dbt init` from this directory, we specified our project name to be `dbt_book` as well. It is from here we want to check if our dbt initialization and access to BigQuery was successful.

So move into this subfolder via `cd dbt_book/`.

```
(venv) sammigachuhi@Gachuhi:~/dbt_book$ cd dbt_book/
(venv) sammigachuhi@Gachuhi:~/dbt_book/dbt_book$ dbt debug
```

Inside the `dbt_book` subfolder we created as part of the dbt initialization prompts, run `dbt debug`. If the final output of the run is `All checks passed!`, you are good to go!

```
19:10:20  Running with dbt=1.8.7
19:10:20  dbt version: 1.8.7
19:10:20  python version: 3.10.12
19:10:20  python path: /home/sammigachuhi/dbt_book/venv/bin/python3
19:10:20  os info: Linux-5.15.153.1-microsoft-standard-WSL2-x86_64-with-glibc2.35
19:10:21  Using profiles dir at /home/sammigachuhi/.dbt
19:10:21  Using profiles.yml file at /home/sammigachuhi/.dbt/profiles.yml
19:10:21  Using dbt_project.yml file at /home/sammigachuhi/dbt_book/dbt_book/dbt_proje
19:10:21  adapter type: bigquery
19:10:21  adapter version: 1.8.2
19:10:22  Configuration:
19:10:22    profiles.yml file [OK found and valid]
19:10:22    dbt_project.yml file [OK found and valid]
19:10:22  Required dependencies:
19:10:22   - git [OK found]

19:10:22  Connection:
19:10:22    method: service-account
19:10:22    database: dbt-project-437116
19:10:22    execution_project: dbt-project-437116
19:10:22    schema: nyc_bikes
19:10:22    location: US
19:10:22    priority: interactive
19:10:22    maximum_bytes_billed: None
19:10:22    impersonate_service_account: None
19:10:22    job_retry_deadline_seconds: None
19:10:22    job_retries: 1
19:10:22    job_creation_timeout_seconds: None
19:10:22    job_execution_timeout_seconds: 300
19:10:22    timeout_seconds: 300
19:10:22    client_id: None
19:10:22    token_uri: None
19:10:22    dataproc_region: None
19:10:22    dataproc_cluster_name: None
19:10:22    gcs_bucket: None
19:10:22    dataproc_batch: None
19:10:22  Registered adapter: bigquery=1.8.2
19:10:26    Connection test: [OK connection ok]

19:10:26  All checks passed!
```

Chapter 7

Models

A model in dbt is any SQL file. It is what dbt will use to build tables, views and any other transformations in your data warehouse. In dbt, models are executed with the hit and run command: `dbt run`.

7.1 Running a model

dbt did us a very big favour during installation; it came with two models already created for us. These are namely the `my_first_dbt_model.sql` and `my_second_dbt_model.sql` within the `models/example` directory. It also provided a `schema.yml` file within the same directory which provides definitions for the models' schema.

Alright. Assuming that you are within the `dbt_book` subdirectory and your virtual environment `(venv)` already activated, type `dbt run` in your terminal like so:

```
(venv) sammigachuhi@Gachuhi:~/dbt_book/dbt_book$ dbt run
```

This will initiate a series of printouts. However, before we go to the expected output, you may run into an error related to the location not being found.

```
404 Not found: Dataset dbt-project-437116:nyc_bikes was not found in location US; reason: notFoun
```

When we were initializing our project using `dbt init` we selected option 1 for US. Luckily, there is a work around to this. It involves editing the `projects.yml` file. If you run `dbt debug` it will also show the path of your `projects.yml` alongside other configuration information.

```
18:19:56  Running with dbt=1.8.7
18:19:56  dbt version: 1.8.7
18:19:56  python version: 3.10.12
18:19:56  python path: /home/sammigachuhi/dbt_book/venv/bin/python3
18:19:56  os info: Linux-5.15.153.1-microsoft-standard-WSL2-x86_64-with-glibc2.35
18:19:57  Using profiles dir at /home/sammigachuhi/.dbt
18:19:57  Using profiles.yml file at /home/sammigachuhi/.dbt/profiles.yml
--snip--
```

Go to the provided path for `profiles.yml` which, in my case, is found at the path - `/home/sammigachuhi/.dbt`. Open it and change the line with `location` to read from `US` to `africa-south1`.

```
dbt_book:
  outputs:
    dev:
      dataset: nyc_bikes
      job_execution_timeout_seconds: 300
      job_retries: 1
      keyfile: /home/sammigachuhi/dbt_credentials/dbt_book.json
      location: africa-south1
--snip---
```

Now come back, and rerun `dbt run` again. dbt should now be able to run against your warehouse and create a table called `my_first_dbt_model` and a view `my_second_dbt_model` in BigQuery.

"How do I know that my queries ran successfully?", you may ask. It is when the terminal prints out: `Completed successfully`. Beneath this message, will be a single line statement of the number of models ran and if there have been any errors. We had two models, so we expect to see this reflect in the log. And it did.

```
18:21:16  Running with dbt=1.8.7
18:21:17  Registered adapter: bigquery=1.8.2
18:21:17  Unable to do partial parsing because saved manifest not found. Starting full
18:21:19  Found 2 models, 4 data tests, 479 macros
18:21:19
18:21:22  Concurrency: 1 threads (target='dev')
18:21:22
18:21:22  1 of 2 START sql table model nyc_bikes.my_first_dbt_model ..................
18:21:30  1 of 2 OK created sql table model nyc_bikes.my_first_dbt_model .............
18:21:30  2 of 2 START sql view model nyc_bikes.my_second_dbt_model ..................
18:21:34  2 of 2 OK created sql view model nyc_bikes.my_second_dbt_model .............
18:21:34
```

```
18:21:34  Finished running 1 table model, 1 view model in 0 hours 0 minutes and 14.88 seconds (14
18:21:34
18:21:34  Completed successfully
18:21:34
18:21:34  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

The second way, and the most obvious, is checking the results in your BigQuery data warehouse. If you see the table and view `my_first_dbt_model` and `my_second_dbt_model` respectively, the query ran successfully.

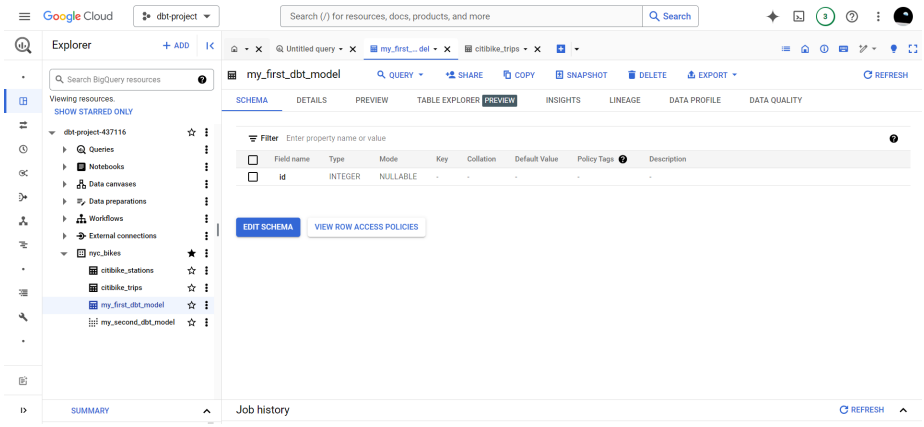


Figure 7.1: Models

It was that simple, isn't it?

7.2 Model structure

Let's take a look at the model `my_first_dbt_model.sql`.

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

with source_data as (

    select 1 as id
    union all
    select null as id

)

select *
```

```
from source_data

/*
    Uncomment the line below to remove records with null `id` values
*/

-- where id is not null
```

The line `{{ config(materialized='table') }}` tells dbt to make the output `my_first_dbt_model` as a table. Any configurations set at the model level will overide the overarching ones set at `dbt_project.yml` file.

If you quickly take a sneek peek at the `dbt_project.yml` file and scroll to the bottom, you will see this configuration:

```
models:
  dbt_book:
    # Config indicated by + and applies to all files under models/example/
    example:
      +materialized: view
```

This configuration simply says that for every model inside the `dbt_book/example` directory, materialize the result as a view[1]. However, inside our `my_first_dbt_model.sql` file, we set the materialization as `table`. What is in the sql model will override what is in the `dbt_project.yml`. However, for `my_second_dbt_model.sql`, we didn't specify the materialization. Therefore, by default, the materialization specified at the `dbt_project.yml` level will be used.

Materialization is the persistence of a model in the data warehouse.

The second part of our first model is the actual SQL statement.

```
with source_data as (

    select 1 as id
    union all
    select null as id

)

select *
from source_data
```

[1] A view is a virtual table, similar to the original table, the physical dataset it was created from

This is a `WITH` SQL statement. In very simple terms, the SQL statement in parentheses `()` is what is referenced as `source_data`. Once we define our SQL statement and close it with parentheses, we can now select the data referenced by `source_data`.

The SQL statements in parentheses is what is referred to as Common Table Expression (CTE). They were designed to simplify complex queries and be used in the context of a larger query.

The second model doesn't have much to provide but it introduces a new trick: the `ref` function.

```
select *
from {{ ref('my_first_dbt_model') }}
where id = 1
```

The `ref` function is part of the jinja templating language. It is used to reference a model that has been provided within the parentheses and enclosed with quotes `('')`. Therefore, in essence, our model simply returns the row(s) from `my_first_dbt_model` that contain the value `1` in the `id` column.

Here is the result of `my_second_dbt_model` view when I query BigQuery to show its contents.

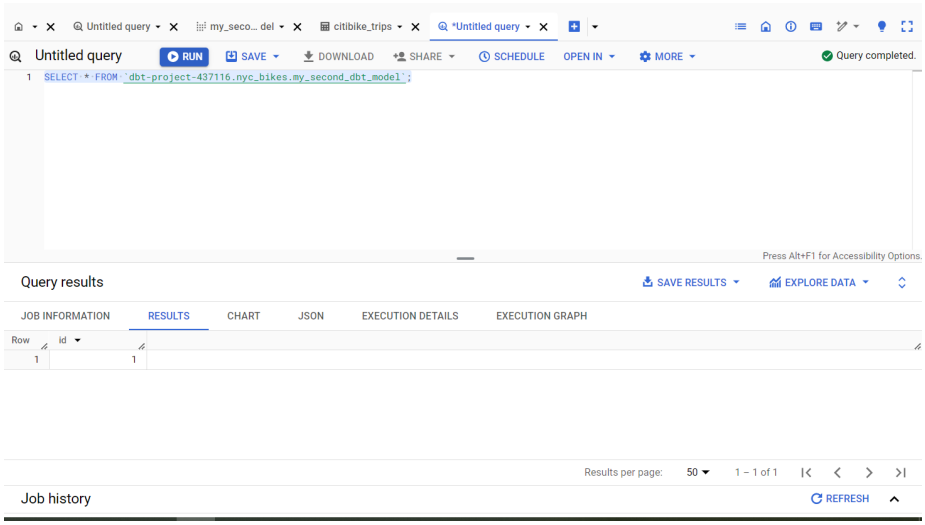


Figure 7.2: Second model

7.3 A custom model

Now that we have seen how to run our models, it's now time to fold our shirts and run our own custom model.

Let's start easy. If you know SQL, there is no limit to the models, both in number and complexity, that you can do in dbt.

Our first model will endeavour to perform a change on the `citi_trips` table within our `nyc_bikes` dataset. In this dataset, there is a column called `tripduration` which shows the time in seconds for every cab trip. Humans prefer to read in minutes so a trivial dbt job would involve creating a column that shows the `tripduration` in minutes. By doing so, dbt would be performing the **T**ransform part in the ELT.

In the `citi_trips_minutes.sql` model, we have written a code that creates a view of this result. Remember, views are just virtual tables but they do save on storage!

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

WITH citi_trips AS (
    SELECT *,
    tripduration / 60 AS trip_duration_min
    FROM
    `dbt-project-437116.nyc_bikes.citibike_trips`
)

SELECT * FROM citi_trips
```

The above SQL statement is also a form of Common Table Expression (CTE). The variable `citi_trips` just references the SQL statement in parentheses.

There is another small trick of the trade. What if this very simple dbt model was just part of thousands, and longer running SQL models? Would we have to run the entire lot of models? No. To run only a specific model we use the `--select` keyword.

Below we run the `citi_trips_minutes` model.

```
dbt run --select models/example/citi_trips_minutes.sql
```

Below is the output.

```
Concurrency: 1 threads (target='dev')
18:50:33
18:50:33  1 of 1 START sql view model nyc_bikes.citi_trips_minutes ..................
```

```
18:50:37  1 of 1 OK created sql view model nyc_bikes.citi_trips_minutes ..... [CREAT
18:50:37
18:50:37  Finished running 1 view model in 0 hours 0 minutes and 8.17 seconds (8.17s).
18:50:37
18:50:37  Completed successfully
18:50:37
18:50:37  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

We are always glad when we see the soothing words "Completed successfully". If you go to BigQuery, you will see a new view `citi_trips_minutes` already created. By default, the name of the newly formed view or table in the data warehouse will be that of the model used to create it.

Similar to the above, the below run command will also work. It doesn't have the file type (`.sql`) suffix.

```
dbt run --select citi_trips_minutes
```

But how do we view this newly created result. Unlike a table, BigQuery does not offer the **PREVIEW** button. However, there is a way…

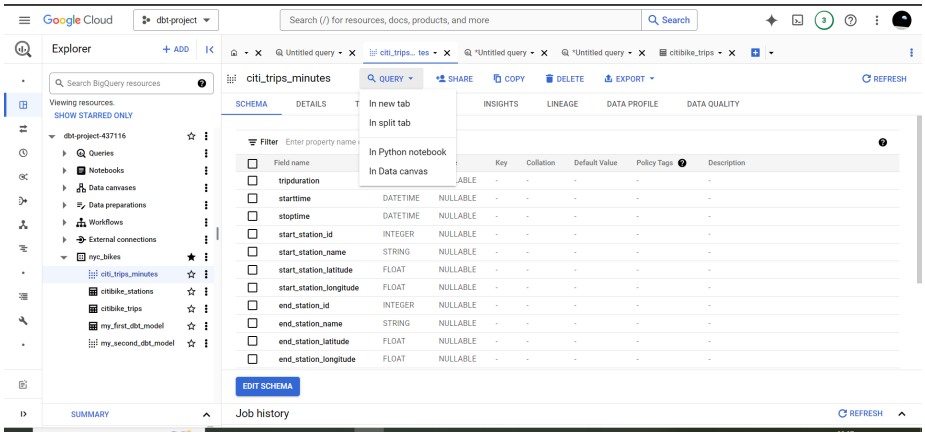


Figure 7.3: Query

Click on the **Query** button, select **In new tab** and a new tab will form. Copy this SQL query onto the tab and run it to display the result of our `citi_trips_minutes` view.

```
SELECT * FROM `dbt-project-437116.nyc_bikes.citi_trips_minutes`
```

Our `trip_duration_min` column of interest is onto the far right of our view.

We had hinted earlier that there is no limit to the complexity or number of models you can create in dbt. Below is a more complex model. This model not only rounds off the minutes to one decimal place but also removes the null columns, thus reducing number of rows from 58,937,715 to 53,108,721. The model is saved as `citi_trips_round.sql`.

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

WITH citi_trips_round AS (
    SELECT *, ROUND(trip_duration_min, 1) AS trip_min_round
    FROM (
        SELECT *,
        tripduration / 60 AS trip_duration_min
        FROM
        `dbt-project-437116.nyc_bikes.citibike_trips`
        )
    WHERE tripduration IS NOT NULL
)

SELECT * FROM citi_trips_round
```

Our model is saved as the `citi_trips_round` view in BigQuery.

Create a new query tab in BigQuery to view the results. Also use the query tab to count the number of rows and compare them to those from the `citibike_trips` table. About 5.8 million rows with null value have been removed.

```
SELECT * FROM `dbt-project-437116.nyc_bikes.citi_trips_round`;

SELECT COUNT(*) FROM `dbt-project-437116.nyc_bikes.citi_trips_round`;

-- This is the count of the original citibike_trips table
SELECT COUNT(*) FROM `dbt-project-437116.nyc_bikes.citibike_trips`;
```

You may be wondering what's the advantage that dbt provides. After all, can't one just use the SQL Query tab in BigQuery just to do the transformations and save the table? Fine, that can work. However, dbt offers a form of persistence to your models. Though modifiable like those in BigQuery, they can be versioned when saved into a code versioning platform like Github.

One more thing, you don't have to use the `models/example` folder. You can choose to rename this one, or save the two models we created in a different folder. They will still run fine.

Create a new folder within the `models` directory called `my_models`. Move the `citi_trips_minutes` and `citi_trips_round` models into the `my_models`

directory. Thereafter, run these two models using `dbt run --select models/my_models`. The models should run just fine.

As an extra bit of information, all successfully compiled and ran models will appear under the `/target` directory.

Chapter 8

Documentation

In the book *The voyages and adventures of Captain Hatteras* the sailors aboard a ship whose expedition was to the North Pole relied on the writing of former captains, explorers and sailsmen to not only find the best possible route to the North Pole, but also the hazards and the places where coal was hidden. So how does this tie to data engineering and dbt? Well, in any digital organization, there is sure to be some turnover. There is sure to be some new chap who would want to wrap their heads around what the organization was doing, and the data being used. Documentation is one way to enable these experts start on a sure footing, but this is rarely the norm. The good thing with dbt is that it provides a way to create documentation at the same place you write code to transform it, and not in a spare pdf[1]!

8.1 The yml files

In dbt, yml files can do a lot of things. One of the stuff it does is documentation and creation of tests for your data. But first, here are the rules of writing a yml file.

1. Indents should be two spaces
2. List items should be indented
3. Use a new line to separate list items that are dictionaries where appropriate

For this case, we shall use the YAML files to create documentation for our data. We already have a template to start us off with. This is the `schema.yml` file inside the `models/example` directory.

[1] Portable Document Format

```
version: 2

models:
  - name: my_first_dbt_model
    description: "A starter dbt model"
    columns:
      - name: id
        description: "The primary key for this table"
        data_tests:
          - unique
          - not_null

  - name: my_second_dbt_model
    description: "A starter dbt model"
    columns:
      - name: id
        description: "The primary key for this table"
        data_tests:
          - unique
          - not_null
```

Let's go through the above structure briefly.

8.1.1 `version: 2`

There is a story behind this. It is that at the very beginning of dbt development, the structure was very different and inserting `version: 2` enabled developers know which version of dbt they were working with. Don't expect a `version: 3` to come any time soon, but this is just a required necessity.

8.1.2 `models`

Remember when we said that a model in dbt is simply a SQL file? Well, next to the `name` key which is under this key you specify the name of your SQL file, minus the `.sql` extension.

8.1.3 `description`

This is where you insert the description of your table.

8.1.4 `columns`

These are the fields contained in your table. You name them here and under each column are three mappings.

- `name` - this is the name of the field in your table. In other words, it is the column name.
- `description` - this is a short explanation of your column.
- `data_test` - the kind of tests that you would like to perform on your field are inserted here. dbt comes with generic tests such as `unique`, `not_null` and `integer` but you can create your own custom tests too.

8.2 Definition for our model

Alright, having gone through the template, we can create our own `yml` file under the `my_models` directory. Let's call it `my_models.yml`. Copy past the YAML structure from `schema.yml` to the `my_models.yml` and let the first YAML structure for `citi_trips_minutes.sql` look like below.

```
version: 2

models:
  - name: citi_trips_minutes
    description: "This is a table with an extra column showing the trip duration in minutes"
    columns:
      - name: tripduration
        description: "Trip Duration (in seconds)"

      - name: starttime
        description: "Start Time, in NYC local time."

      - name: stoptime
        description: "Stop Time, in NYC local time."

      - name: start_station_id
        description: "Start Station ID"

      - name: start_station_name
        description: "Start Station Name"

      - name: start_station_latitude
        description: "Start Station Latitude"
```

```
      - name: start_station_longitude
        description: "Start Station Longitude"

      - name: end_station_id
        description: "End Station ID"

      - name: end_station_name
        description: "End Station Name"

      - name: end_station_latitude
        description: "End Station Latitude"

      - name: end_station_longitude
        description: "End Station Longitude"

      - name: bike_id
        description: "Bike ID"

      - name: usertype
        description: "User Type (Customer = 24-hour pass or 7-day pass user, Subscribe

      - name: birth_year
        description: "Year of Birth"

      - name: gender
        description: "Gender (unknown, male, female)"

      - name: customer_plan
        description: "The name of the plan that determines the rate charged for the tr

      - name: trip_duration_min
        description: "The trip duration in minutes"
```

What we've done is quite straightforward. We have simply typed out the descriptions next to the `description` mapping key.

However, imagine you were working with hundreds of models which use similar definitions. Would you have the nerve to copy paste every definition to its respective model? Perhaps not. There is a function by the name of the `docs()` function which can reference to descriptions in a separate markdown file.

8.3 Using the doc function

To use the `doc()` function, we write our definitions in a separate markdown (`.md`) file and place the descriptions within `{% docs <field-name> %}` `{% enddocs %}` tags. For this tutorial, we created three markdown tables.

- `references.md` - this contains the descriptions for our column names of interest
- `tables.md` - contains the descriptions for our tables of interest
- `overview.md` - contains the text that will go to the overview page

Here is what our `references.md` contains. As you can see, we have provided some textual information for some of our column names. We can also add some more style to our descriptions since they are now on a separate markdown. For example we could insert links, make the text italic, and bold if you wish!

```
{% docs tripduration %}

Trip Duration (in seconds). Like:

- How long did the trip take?
- What is the time in seconds?
- More info on time, see [here](https://www.poemhunter.com/poem/time-xxi/)

*https://www.poemhunter.com/poem/time-xxi/*

{% enddocs %}

{% docs starttime %}

Start Time, in NYC local time. As accurate as could ever be.

{% enddocs %}

--snip--
```

The `tables.md` just contains a description of our `citi_trips_round` table.

```
{% docs citi_trips_round %}

This table contains the trip duration in minutes to one decimal place only.

{% enddocs %}
```

Now, in order to enable dbt reference these descriptions from our YAML file, we would simply use the `doc ()` function as shown below:

```
- name: citi_trips_round
    description: '{{ doc("citi_trips_round") }}'
    columns:
      - name: tripduration
        description: '{{ doc("tripduration") }}'

      - name: starttime
        description: '{{ doc("starttime") }}'

      - name: stoptime
        description: '{{ doc("stoptime") }}'

      - name: start_station_id
        description: "Start Station ID"

--snip--
```

The file saved as `overview.md` in our project will be used to display the home page of our dbt documentation website. However, the homepage uses a different syntax, like so:

```
{% docs __overview__ %}

Some more text here...

{% enddocs %}
```

Therefore, here is some dummy text for our overview page.

```
{% docs __overview__ %}

# Learning dbt

Learning is not merely the acquisition of knowledge, but the cultivation of the mind.

--snip--

{% enddocs %}
```

8.4 Images in dbt documentation

They say an image is worth a thousand words. In dbt, we store images in a folder called `assets`. Ideally, one can create any folder in dbt to store images provided you reference it correctly in the documentation. However, for versioning purposes, it is better you store it in an `assets` folder. Furthermore, images in dbt, once run as part of your document generation, will also appear under the `targets/` folder just like your SQL models.

Therefore, going with the recommended approach, create an `assets/` folder under `dbt_book`. Place your image in there.

Go to the `dbt_projects.yml` file and create a new line with the following code:

```
asset-paths: ["assets"]
```

This path tells dbt to copy all items within `assets` into the `target` directory. Any image in a different directory will not get copied into the `target` directory when documents are generated.

Finally, as the missing piece to the puzzle, insert a reference to your image in the `overview.md` file.

```
![Example image](assets/image_example.jpg)
```

One can also create custom overviews for the dbt packages they used.

Below is our complete `overview.md` file.

```
{% docs __overview__ %}

# Learning dbt

Learning is not merely the acquisition of knowledge, but the cultivation of the mind. It is throu

![Example image](assets/image_example.jpg)

Some more text here...

{% enddocs %}
```

8.5 Generating the document

Now is the time where we ignite the rocket engines and shoot off. To generate a dbt documentation, first run `dbt docs generate`. This command tells dbt to

compile the necessary information of your project into the `catalog.json` and `manifest.json` files. The ignition key for our documentation generation is `dbt docs serve`. dbt will generate a list of outputs and create a popup providing the link to open up your documentation. You can click on the popup or copy-paste the link. Our documentation is in the host: `localhost:8080/`.

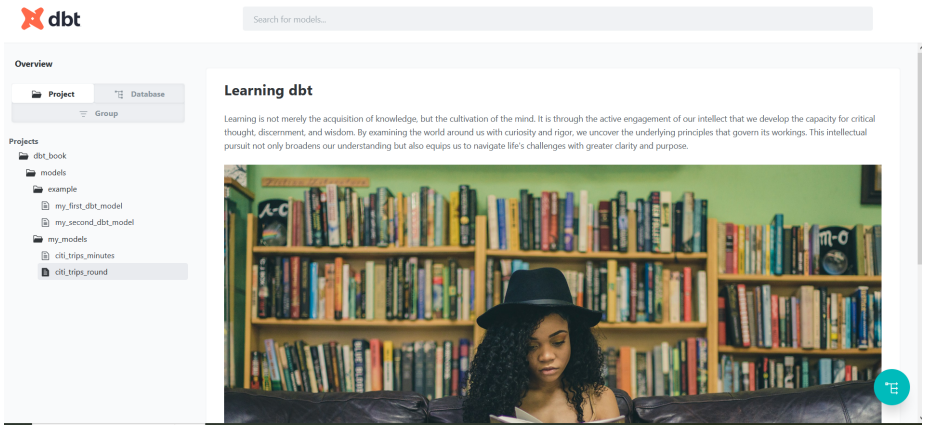


Figure 8.1: Model page

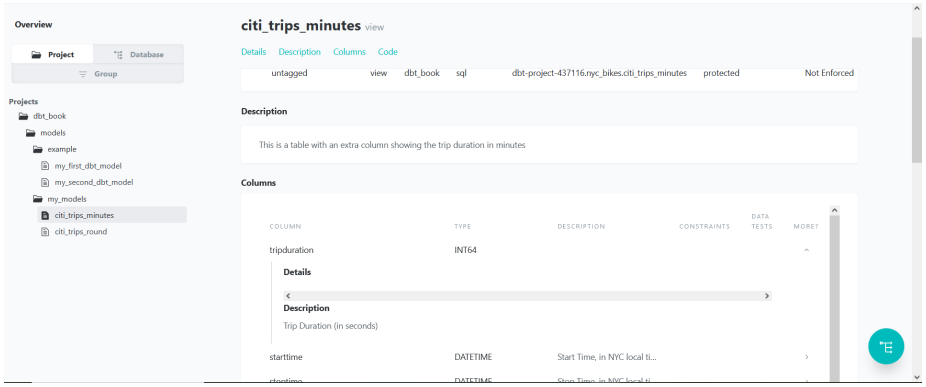


Figure 8.2: Models page

If you go to the `targets` directory, our image(s) will be there!

There is also one more cool functionality of the dbt documentation. On the bottom right, there is a turquoise button for showing the lineage graph for each model. If you click on any model, such as `my_second_dbt_model`, you will see it shows a dependency on `my_first_dbt_model`. If you have worked on a model that has several dependencies, or children, the model will most likely be more complex. A good example will be for the `citi_trips_long` model.

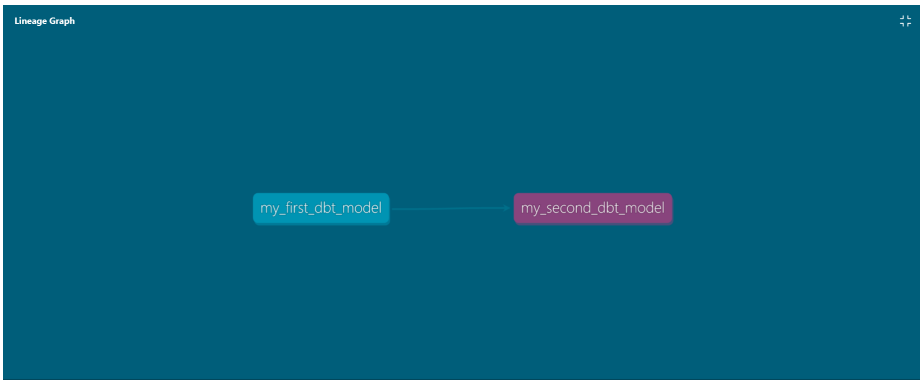


Figure 8.3: Lineage graph

It is highly encouraged to play around with the buttons **resources**, **packages**, **tags**, –**select** and –**exclude**. For the **select** button, play around with inserting **+** both before and after the name of the model. Clue: it has to do with showing or hiding the model's dependencies and/or children.

Below is an example of the `citi_trips_long` model, which has more than one dependency.

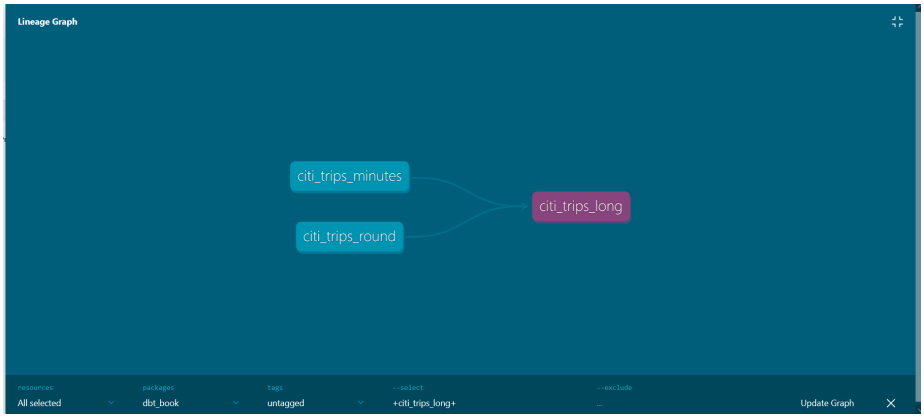


Figure 8.4: A model with more than one dependency

Chapter 9

Tests

Probably if you've worked on a Windows computer, you must have used Microsoft Defender Full Scan at some point. During or at the end of the scan, some results were displayed. dbt works in almost the same way, only that this time they scan your data.

Tests in dbt are basically assertions of your data. That is, they are just some assumptions that you have of your datasets that are correct. Tests enable one to know: 1) which assumptions are wrong about our data, and 2) which parts of our data diverge from the expected norm. Tests can sometimes feel like something to bemoan, can fail (as all tests do) but the overarching advantage is that they make us understand more about our data. They can also be lifesavers in that they can pinpoint a serious problem which could be harder to debug later on! In a nutshell, dbt tests perform much like a programmatic scan that will only spew out errors of something that is suspicious.

9.1 Types of tests in dbt

In dbt, a test can be defined in either of the following two ways:

1. generic test - this is a test written in a SQL model and defined inside a YAML file. The test in the YAML file is defined using jinja Macros. dbt comes with four all-batteries included tests namely:

- unique - asserts that the column has no repeating values
- not_null - asserts that there are no null values
- accepted_values - checks if the values in your field correspond to those in a defined list

- relationships - checks if the field has an existing relationship with another field in a different table.

2. singular test - some normally refer to this as a custom test. This is a SQL query which is used to check assertions in your data. The SQL files that make up your test are defined inside the `tests` directory or the path defined in the `test-paths` key inside the `dbt_project.yml` file. Each SQL file will have one test only. Nevertheless, if this test will be used across many fields and files, you can reference it using jinja macros `{{ }}`. If this is the case, it is no longer a singular test but a generic *custom* test.

Let's start with a dbt out-of-the-box generic test.

9.2 Generic tests in dbt

We will start with a very simple test, the `not_null` test on the `start_station_name` column of `citi_trips_long` model. Surely, unless someone is teleporting from somewhere, every rented bike must have an origin.

```
- name: citi_trips_long
    description: '{{ doc("citi_trips_long") }}'
    columns:
      - name: starttime
        description: '{{ doc("starttime") }}'

      --snip--

      - name: start_station_name
        description: "Start Station Name"
        tests:
          - not_null
```

We use the below code to test only those models under `my_models` folder.

```
dbt test --select my_models
```

Here is the output.

```
--snip--
```

```
19:15:33  Concurrency: 1 threads (target='dev')
19:15:33
19:15:33  1 of 1 START test not_null_citi_trips_long_start_station_name ..... [RUN]
19:15:36  1 of 1 PASS not_null_citi_trips_long_start_station_name ..... [PASS
19:15:36
19:15:36  Finished running 1 test in 0 hours 0 minutes and 4.44 seconds (4.44s).
19:15:36
19:15:36  Completed successfully
19:15:36
19:15:36  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

If we had gone with the more blanket code of `dbt test`, it would have also tested those models under the shipped `examples` folder where there is already a test designed to fail, purely for demonstration purposes. Back to our `citi_trips_long` model, we can see that the test passed successfully. As expected, there are no null values in our `start_station_name` field. If the opposite happened, then there must be a row with a null value under this field.

Now let's create a test that will fail on purpose. From a large dataset, obviously the station names can't be unique all through. So we insert the `unique` value under the `tests` key as follows:

```
- name: start_station_name
        description: "Start Station Name"
        tests:
          - not_null
          - unique
```

The output is as below. Note that it results in a failure and also shows the path to the SQL query inside the `target/` directory that was used to carry out the test. If you paste this SQL query from the `target` directory into your data warehouse, it will return the number of rows that failed the test.

```
19:23:35  Concurrency: 1 threads (target='dev')
19:23:35
19:23:35  1 of 2 START test not_null_citi_trips_long_start_station_name ..... [RUN]
19:23:38  1 of 2 PASS not_null_citi_trips_long_start_station_name ..... [PASS
19:23:38  2 of 2 START test unique_citi_trips_long_start_station_name ..... [RUN]
19:23:41  2 of 2 FAIL 910 unique_citi_trips_long_start_station_name ..... [FAIL
19:23:41
19:23:41  Finished running 2 data tests in 0 hours 0 minutes and 6.78 seconds (6.78s).
19:23:41
19:23:41  Completed with 1 error and 0 warnings:
19:23:41
19:23:41  Failure in test unique_citi_trips_long_start_station_name (models/my_models/my_models.y
```

```
19:23:41    Got 910 results, configured to fail if != 0
19:23:41
19:23:41    compiled code at target/compiled/dbt_book/models/my_models/my_models.yml/u
19:23:41
19:23:41  Done. PASS=1 WARN=0 ERROR=1 SKIP=0 TOTAL=2
```

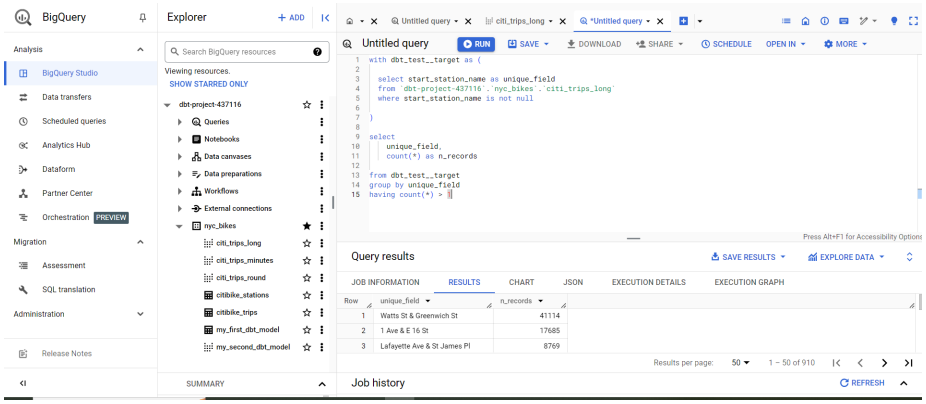


Figure 9.1: Failed test

9.3 Singular tests in dbt

As mentioned earlier, singular tests are SQL models in your `tests` folder. They differ from the generic tests in that they are bespoke to your data needs. For example, if one of your tests doesn't conform to the four tests shipped with dbt, you can create one inside the `tests` folder.

Below we create a singular test called `unnecessary_trips.sql` inside the `test` folder. The purpose of the test is to raise an error if a trip is less than 10 min. The latter must surely generate an error! Notice that one can reference another model within a test using the `ref()` function.

```
SELECT bikeid, start_station_name, end_station_name,
    birth_year, gender, tripduration
FROM {{ ref('citi_trips_round') }}
WHERE trip_min_round < 10
```

So to perform the litmus test, run our usual magic phrase:

```
dbt test --select unnecessary_trips
```

Below will be the output.

```
19:51:50  Concurrency: 1 threads (target='dev')
19:51:50
19:51:50  1 of 1 START test unnecessary_trips ..................................... [RUN]
19:51:53  1 of 1 FAIL 25183763 unnecessary_trips ................................. [FAIL
19:51:53
19:51:53  Finished running 1 test in 0 hours 0 minutes and 4.70 seconds (4.70s).
19:51:53
19:51:53  Completed with 1 error and 0 warnings:
19:51:53
19:51:53  Failure in test unnecessary_trips (tests/unnecessary_trips.sql)
19:51:53    Got 25183763 results, configured to fail if != 0
19:51:53
19:51:53    compiled code at target/compiled/dbt_book/tests/unnecessary_trips.sql
19:51:53
19:51:53  Done. PASS=0 WARN=0 ERROR=1 SKIP=0 TOTAL=1
```

As you can see, a total 25183763 rows failed the test.

A singular test can also be transformed into a generic test when its reused across the fields of your table(s) in the data warehouse. To demonstrate this, let's create some generic tests.

9.4 Creating a generic test

Custom generic tests are created within a `generic` folder within the `tests` directory. Thus, within your `tests/generic` directory, you will place the SQL models for your tests. Anything returned by your SQL models is in fact your tests failing! If nothing is returned, then your test passed!

Create a SQL model called `long_characters` within the `tests/generic` directory.

```
{% test long_characters (model, column_name) %}
SELECT * FROM {{ model }}
WHERE LENGTH({{ column_name }}) > 15
{% endtest %}
```

Let's go through the above query line by line. We begin a generic test by using the `{%test <model-name> (model, column_name) %}` *SQL statement in here* `{% endtest %}` tags. The `model` and `column_name` are standard arguments where one or both should be defined.

- `model` - the resource on which the test will be operated on. In our case, this is any model which the test will run on.

- `column_name` - this is a field within which the model will be run against. In other words, the column at which this model will operate on.

The SQL statement within the test tags references the model and the columns using the `{{ model }}` and `{{ column_name }}` respectively. For example, if the test is placed in the `my_models` YAML file, under the `citi_trips_long` model name, for the field `start_station_name`, it is as though the test is running this SELECT statement:

```
SELECT * FROM citi_trips_long
WHERE LENGTH(start_station_name) > 15
```

In very few words, the above SQL tells dbt to shout out an error if any station name is greater than 15 characters.

Let's create another test that will raise alarms if there are special characters within your `start_station_name`.

```
{% test special_characters (model, column_name) %}
SELECT * FROM {{ model }}
WHERE {{ column_name }}
LIKE '%^[a-zA-Z0-9+-]%'
{% endtest %}
```

Okay, it's now time for our litmus test. Going back to the `start_station_name` mapping that we have performed some litmus tests, lets also add our two *custom* generic tests of `long_character` and `special_characters`.

```
- name: citi_trips_long
    description: '{{ doc("citi_trips_long") }}'
    columns:
      --snip--

      - name: start_station_name
        description: "Start Station Name"
        tests:
          - not_null
          - unique
          - long_characters
          - special_characters
```

If we go for the big bull and run all our tests with the trusty `dbt test` keyword, we can see the output of the nine tests we have so far.

```
19:08:15  Concurrency: 1 threads (target='dev')
19:08:15
19:08:15  1 of 9 START test long_characters_citi_trips_long_start_station_name ..... [RUN]
19:08:29  1 of 9 FAIL 11463868 long_characters_citi_trips_long_start_station_name ..... [FAIL

--snip--

19:08:55  5 of 9 START test special_characters_citi_trips_long_start_station_name ..... [RUN]
19:08:59  5 of 9 PASS special_characters_citi_trips_long_start_station_name ..... [PASS

--snip--

19:09:07  9 of 9 START test unnecessary_trips ..... [RUN]
19:09:10  9 of 9 FAIL 25183763 unnecessary_trips ..... [FAIL
```

From the above output, we can see that there were some station names with quite some long names and (thankfully) no station names with special characters.

There is also another output following the above which shows how many records failed, depending on your test.

```
19:09:10  Completed with 4 errors and 0 warnings:
19:09:10
19:09:10  Failure in test long_characters_citi_trips_long_start_station_name (models/my_models/my
19:09:10    Got 11463868 results, configured to fail if != 0

--snip--

19:09:10  Failure in test unique_citi_trips_long_start_station_name (models/my_models/my_models.y
19:09:10    Got 910 results, configured to fail if != 0
```

9.5 Configuring custom generic tests

What if you feel that the *FAIL* alert like in the above tests is shouting too much?! That you would prefer them to be *WARNINGS* because they are non-critical?

You can do so by setting the configuration to display as a warning rather than an error.

```
{% test long_characters (model, column_name) %}
    {{ config(severity = 'warn') }}
    SELECT * FROM {{ model }}
    WHERE LENGTH({{ column_name }}) > 15
{% endtest %}
```

Here is the output as a warning.

```
19:16:34  1 of 9 START test long_characters_citi_trips_long_start_station_name .......
19:16:36  1 of 9 WARN 11463868 long_characters_citi_trips_long_start_station_name ....
19:16:36  2 of 9 START test not_null_citi_trips_long_start_station_name ..............
19:16:39  2 of 9 PASS not_null_citi_trips_long_start_station_name ....................
-- snip --
19:16:56  Warning in test long_characters_citi_trips_long_start_station_name (models/my
19:16:56  Got 11463868 results, configured to warn if != 0
-- snip --
```

However, in case you immediately change your mind that the failing tests of `long_characters` should be an error rather than a warning, you can override your custom generic SQL models by specifying the severity within the YAML definition files.

```
- name: start_station_name
        description: "Start Station Name"
        tests:
          - not_null
          - unique
          - long_characters:
            severity: 'error'
          - special_characters
```

In the generated output, it will be back to business as usual with the 'FAIL' keyword.

```
-- snip --
19:21:25  1 of 9 START test long_characters_citi_trips_long_start_station_name .......
19:21:28  1 of 9 FAIL 11463868 long_characters_citi_trips_long_start_station_name ....
-- snip --

19:21:50  Failure in test long_characters_citi_trips_long_start_station_name (models/my
19:21:50    Got 11463868 results, configured to fail if != 0
-- snip --
```

9.6 Storing test failures

If you thought that dbt tests are a cool feature, then there is one more trick in the bag if you want a neat one-liner to view a dataset of the failing records. The

open sesame key word is `--store-failures`. dbt will store the failing records as a table in the database.

Let's try it out.

```
dbt test --store-failures
```

Of course our test will generate failed records, but we've already seen them. Here is part of the output for the test of long characters above the 15 character threshold.

```
19:26:40  Failure in test long_characters_citi_trips_long_start_station_name (models/my_models/my
19:26:40    Got 11463868 results, configured to fail if != 0
19:26:40
19:26:40    compiled code at target/compiled/dbt_book/models/my_models/my_models.yml/long_charact
19:26:40
19:26:40    See test failures:
  ---------------------------------------------------------------------------------------------
  select * from `dbt-project-437116`.`nyc_bikes_dbt_test__audit`.`long_characters_citi_trips_long
  ---------------------------------------------------------------------------------------------
```

If you paste the `SELECT` statement in one of the SQL tabs for BigQuery, it will not only return the number of failing records but also the data that is part of the failing records.

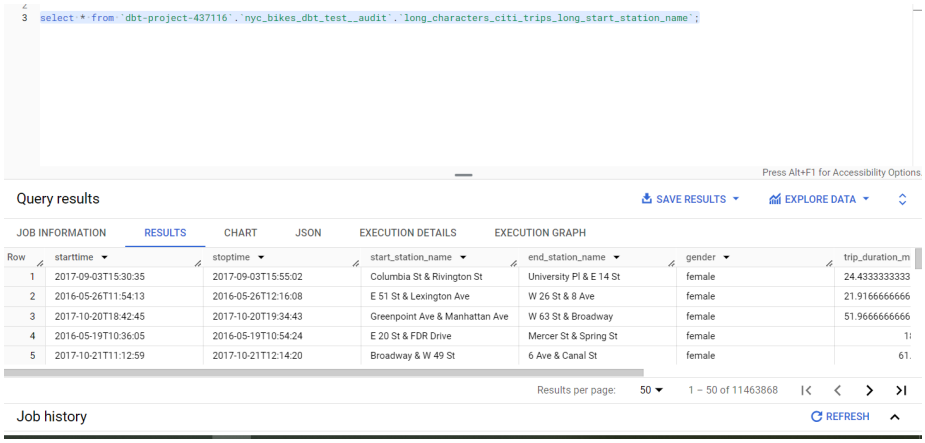


Figure 9.2: Failing records

That's a very convenient one-liner!

Below are other forms of generic tests shipped with dbt, for the `accepted_values` and `relationships`. The latter took too long to run and was cancelled midway.

```
- name: end_station_name
        description: "End Station Name"
         tests:
            - relationships:
                to: ref('citi_trips_round')
                field: end_station_name

      - name: gender
        description: "Gender (unknown, male, female)"
        tests:
          - accepted_values:
              values: ['unknown', 'male', 'female']
```

It's been quite an exciting journey with tests. And for sure dbt tests do grill your data!

Chapter 10

dbt Expectations package

What is dbt-expectations? dbt-expectations is an extension package for dbt which works much akin to the Great Expectations package for Python. It was intentionally designed to provide Great Expectations like features in dbt, but now from dbt itself rather than integrating Great Expectations (GE).

Unless you've used GE, you may be wondering what this is in the first place, and its okay to feel lost. GE is much like tests in the previous chapter, it conducts quality tests on your data, thus flagging those that deviate from the set assertions.

I would put dbt-expectations and GE on the same plane and use an allegory to drive the point home: that of a car. When buying a car, there are some common checklist items, and others bespoke depending on your car model. For example, an ordinary car must have the following features (at least most of them):

- have four wheels
- have a driver's seat
- have a gear (whether manual or automatic)
- have headlights
- have a windshield

The above list can go on and on depending on your knowledge of cars. But your checklist can also contain some unique items, but which are a must-have depending on your car make. For example, here is a checklist of the Volvo XC60 T6:

- 0.9l/100km fuel consumption
- Allowed emissions 22g/km (the less the better)
- Hybrid fuel type

So if you go to a showrooms and the beautiful or handsome sales agent takes you to the Volvo XC60, you will be perusing it as you cross your checklist. dbt-expectations and GE work in the same way.

10.1 `dbt-expectations` installation

According to the documentation dbt-expectations will work for dbt versions 1.7x and higher. Let's first pass this little test.

```
dbt --version
```

If you get your dbt-core version is above 1.7x, then you can proceed. If not, you need to update your dbt. You can do so using `python -m pip install --upgrade dbt-core` or if you want to be more specific, this will do: `python -m pip install --upgrade dbt-core==0.19.0`.

Ours, at the moment of writing this book, was version `1.8.7`. Therefore we have a clean bill of health to proceed.

dbt-expectations isn't installed in the same *type and enter* kind of means like we did for `dbt-core` and `dbt big-query`. Nevertheless, some code is written in some YAML files and from henceforth dbt recognises it.

First create a `packages.yml` file in the same level as your `dbt_project.yml` file. You can do so by running this command:

```
touch packages.yml
```

On the `packages.yml` file, insert the following:

```
packages:
  - package: calogica/dbt_expectations
    version: [">=0.10.0", "<0.11.0"]
```

Apart from that, the `dbt-date` dependency must also be installed. This is because `dbt-expectations` references it. However, this will be installed in the `dbt_project.yml` file rather than the `packages.yml` file. So inside the `dbt_project.yml` paste the following just before the `materializations` dictionary.

```
vars:
  'dbt_date:time_zone': 'Africa/Nairobi'
```

You may insert any valid timezone apart from the one specified above, but we highly suggest that you use your timezone.

Now run `dbt deps` to seal the deal by installing the `dbt-expectations` package.

```
dbt deps
```

Here is the output showing the successful installation of the package in our environment.

```
19:31:10  Running with dbt=1.8.7
19:31:12  Updating lock file in file path: /home/sammigachuhi/dbt_book2/dbt_book/package-lock.yml
19:31:13  Installing calogica/dbt_expectations
19:31:44  Installed from version 0.10.4
19:31:44  Up to date!
19:31:44  Installing calogica/dbt_date
19:31:45  Installed from version 0.10.1
19:31:45  Up to date!
```

10.2 Types of `dbt-expectations` tests

`dbt-expectations` comes with a plethora of tests' functions which can be classified into the following categories.

- Table shape
- Missing values, unique values, and types
- Sets and ranges
- String matching
- Aggregate functions
- Multi-column
- Distributional functions

We will perform one test in each category just to exemplify the potential of `dbt-expectations`.

10.2.1 Table shape

10.2.1.1 `expect_table_row_count_to_equal_other_table`

Description: Expect the number of rows in a model match another model.

We will expect the `citi_trips_round` and the `citi_trips_minutes` tables to have the same number of rows since their respective models used the same

`citi_bike_trips` table. Therefore, the two tables should pass this test, or will they?

Since we only want to concentrate on the models within the `my_models` directory, just run: `dbt test --select models/my_models`

Here is the output.

```
-- snip --
07:57:38  2 of 8 FAIL 1 dbt_expectations_expect_table_row_count_to_equal_other_table_c
07:57:38  3 of 8 START test dbt_expectations_expect_table_row_count_to_equal_other_tabl
07:57:42  3 of 8 FAIL 1 dbt_expectations_expect_table_row_count_to_equal_other_table_c
-- snip --
```

This leaves one puzzled, why?

A close look at the model for `citi_trips_round` reveals the answer. This model was designed to only work on non-null rows, unlike the `citi_trips_minutes` which worked on all rows, null or not. Therefore the `citi_trips_round` had less rows and thus the generated the error. In case your tests results seem incongruent, it is always good to recheck the models to refresh your memory, as we did here.

In fact, dbt did a good job of generating an SQL to show us the error:

```
07:58:03  Failure in test dbt_expectations_expect_table_row_count_to_equal_other_table_
07:58:03    Got 1 result, configured to fail if != 0
07:58:03
07:58:03    compiled code at target/compiled/dbt_book/models/my_models/my_models.yml/db
```

If you click on the destination of the SQL statement and copy the contents to the SQL query tab of BigQuery, you will see the difference in row count for the two tables.

10.2.2 Missing values, unique values, and types

10.2.2.1 `expect_column_values_to_not_be_null`

Description: Expect column values to not be null.

This is a *no-brainer* kind of test. Can you guess which columns in any of our tables in the `nyc_bikes` dataset should never be null? Here is a clue, station_id and station names, unless the biker teleports to or from somewhere!

So on the `my_models` YAML file, insert the below test on the `start_station_id`, `start_station_name`, `end_station_id` and `end_station_name` fields.

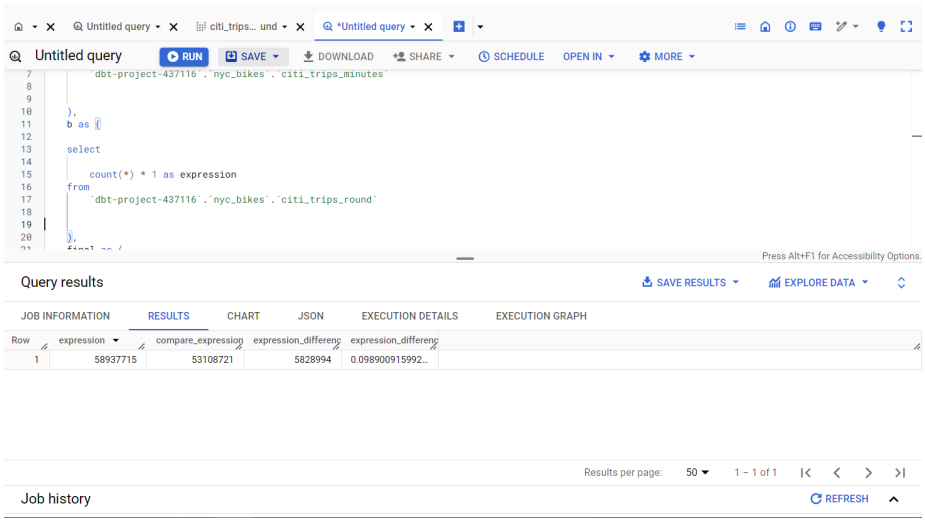


Figure 10.1: Row count difference

```
tests:
  - dbt_expectations.expect_column_values_to_not_be_null
```

You will get some interesting results. some of the tests fail for the `citi_trips_minutes` model because of the many null rows in the table. However, none of this particular test fail for the `citi_trips_round` table; it has zero null rows.

A caveat when using the `dbt_expectations.expect_column_values_to_not_be_null`, only add a colon `:` when specifying more optional parameters such as `row_condition: "id is not null" # (Optional)`. Otherwise, leave it out.

10.2.3 Sets and Ranges

10.2.3.1 `expect_column_values_to_be_in_set`

Description: Expect each column value to be in a given set.

This test works best for where you are sure that a certain column will only accept certain values. A good example is the `gender` column. There can only be three results: male, female and other. Here we insert the test in our `citi_trips_minutes` and `citi_trips_round` models.

```
- name: gender
  description: "Gender (unknown, male, female)"
  tests:
```

```
    - dbt_expectations.expect_column_values_to_be_in_set:
        value_set: ['unknown','male','female']
```

If you run the above test for both the `citi_trips_minutes` and `citi_trips_round` models, the test will fail for the former. Why, because of the pesky null rows. However, to take the null rows into consideration and take them as accepted values in the `citi_trips_minutes` model only, we simply add an empty quotation marks, like so (`''`). Here is our modified test: `value_set: ['', 'unknown','male','female']`. The test will then pass for our `citi_trips_minutes` table.

10.2.4 String matching

10.2.4.1 `expect_column_value_lengths_to_be_between`

Description: Expect column entries to be strings with length between a min_value value and a max_value value (inclusive).

Because our `citi_trips_minutes` table has several null rows, we will put the minimum expected value to be 0. Since we also want to catch those station names with overly long names, we will put the max value as 70. So for both the `start_station_name` and the `end_station_name`, we inserted the following test:

```
tests:
  - dbt_expectations.expect_column_values_to_not_be_null
  - dbt_expectations.expect_column_value_lengths_to_be_between:
      min_value: 1 # (Optional)
      max_value: 70 # (Optional)
```

We are glad to know both models passed this simple test.

10.2.5 Aggregate

10.2.5.1 `expect_column_max_to_be_between`

Description: Expect the column max to be between a min and max value

You may wonder what the purpose of this test is. But don't dismiss it yet, it can come quite in handy when searching for outlier values. We will demonstrate it in catching overly long bike trips. However, this test needs some background knowledge of your data.

Applying the below queries on BigQuery will help.

```
SELECT AVG(trip_duration_min) FROM dbt-project-437116.nyc_bikes.citi_trips_minutes; -- 16

SELECT MAX(trip_duration_min) FROM dbt-project-437116.nyc_bikes.citi_trips_minutes; -- 325167.48

SELECT * FROM dbt-project-437116.nyc_bikes.citi_trips_minutes
WHERE trip_duration_min > 200000;
```

Now lets place the limits of our maximum trip duration values to be between the average of 16 and some intermediate value such as 100, 000 minutes (1, 666 hours)!

```
tests:
  - dbt_expectations.expect_column_max_to_be_between:
      min_value: 16 # (Optional)
      max_value: 100000 # (Optional)
```

That will flag off some errors, but if you change the `max_value` parameter to `360000`, the tests will pass. However, in the `trip_min_round` field of the `citi_trips_round` model, we set the `max_value` as 100000 to demonstrate an error of this test.

10.2.6 Multi-column

10.2.6.1 `expect_column_pair_values_A_to_be_greater_than_B`

Description: Expect values in column A to be greater than column B.

This kind of test comes in handy when you want to ensure that one of your columnar values is greater than, or less than that of a different column. A good example is a comparison of trip duration in seconds in the `tripduration` column versus the trip duration in minutes from `trip_min_round` column in our `citi_trips_round` table. Definitely time in seconds will always have a greater value in terms of length than the more concise minutes values!

In our `citi_trips_round` model, insert the test as follows:

```
- name: citi_trips_round
    description: '{{ doc("citi_trips_round") }}'
    tests:
      - dbt_expectations.expect_table_row_count_to_equal_other_table:
          compare_model: ref("citi_trips_minutes")
      - dbt_expectations.expect_column_pair_values_A_to_be_greater_than_B:
          column_A: tripduration
          column_B: trip_min_round
```

It surely does pass the test.

```
-- snip --
09:40:46  5 of 26 PASS dbt_expectations_expect_column_pair_values_A_to_be_greater_than_
```

10.2.7 Distributional functions

This is another category of shipped-in tests of `dbt-expectations`. However, they require some statistical homework to be conducted on your data prior to applying the tests. The tests under this category include: `expect_column_values_to_be_within_n_moving_stdevs`, `expect_column_values_to_be_within_n_stdevs` and `expect_row_values_to_have_data_for_every_`

Chapter 11

Seeds

Seeds are Comma Separated Values (csv) files stored inside your `seeds` directory which can be loaded into your data warehouse using the `dbt seed` command.

Seeds can also be referenced by your SQL models using the `ref()` function. Seeds in dbt are version controlled. That is, you can revert them to a previous state.

Seeds are best used for data that changes infrequently. A good example is country codes, email accounts and station names. However, seeds should not be used to store sensitive information such as passwords.

To demonstrate about seeds in dbt, we will try to upload a New York City (NYC) bike history data for 2014.

Extract the zip folder and copy the `3e7acf34-19ba-4bf4-8dd2-cf349623dc6b.csv` inside the `dbt_book/seeds` directory. To reduce the verbosity of its name, rename it to `2014-tripdata.csv`.

11.1 Uploading a seed into your data warehouse

Believe you me we had a better csv table to upload, one much more related to the NYC bikes dataset. However, because it was ~320MB and we want to be economical in upload times and bandwith, we settled for this historical data. Nevertheless, keeping on with this chapter, to upload a seed into a data warehouse, we use this command:

```
dbt seed
```

After that, it's a test of patience. Depending on your upload speeds, it shouldn't take long to upload a 36MB file to BigQuery.

```
-- snip --
16:31:51  Concurrency: 1 threads (target='dev')
16:31:51
16:31:51  1 of 1 START seed file nyc_bikes.2014-tripdata ..............................
16:33:17  1 of 1 OK loaded seed file nyc_bikes.2014-tripdata ..........................
16:33:17
16:33:17  Finished running 1 seed in 0 hours 1 minutes and 28.54 seconds (88.54s).
16:33:17
16:33:17  Completed successfully
16:33:17
16:33:17  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

If you go to BigQuery and refresh the contents of your `nyc_bikes` dataset, you should see the `2014-tripdata` table present.

11.2 Referencing seeds in models

Just like you would reference a model in another model, we can also reference seeds in another model. All your need to reference a seed is to place the name of the csv file, excluding the `.csv` extension inside the `ref()` function.

For example, we want to create a view that contains those start station names from our 2014 table that are existent in the `citi_trips_long` model.

Within the `models/my_models` directory, create the `citi_stations_2014.sql` model with the following query:

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

WITH citi_stations_2014 AS (
    SELECT * FROM {{ref ('2014-tripdata') }}
    WHERE `start station name` IN (
        SELECT start_station_name FROM {{ ref('citi_trips_long') }}
    )
)

SELECT * FROM citi_stations_2014
```

Thereafter, run the model using `dbt run --select citi_stations_2014`

That will create a view that contains only those stations within the `2014-tripdata.csv` also within the `citi_trips_long` model. Much to our surprise, all the stations within our 2014 table are also found in the `citi_trips_long` model!

Here are the SQL queries we used to perform a count of each of the two tables in BigQuery.

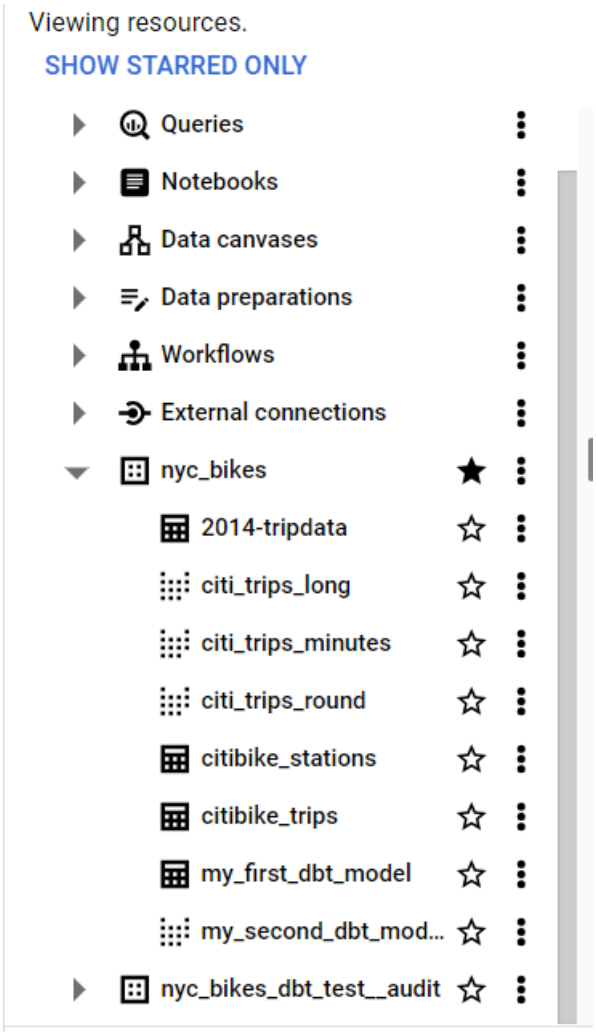


Figure 11.1: Seeds

```
SELECT COUNT(*) FROM `dbt-project-437116`.`nyc_bikes`.`2014-tripdata`;

SELECT COUNT(*) FROM dbt-project-437116.nyc_bikes.citi_stations_2014;
```

Both returned a value of `224736`.

Here is the view of the `citi_stations_2014` model in BigQuery.

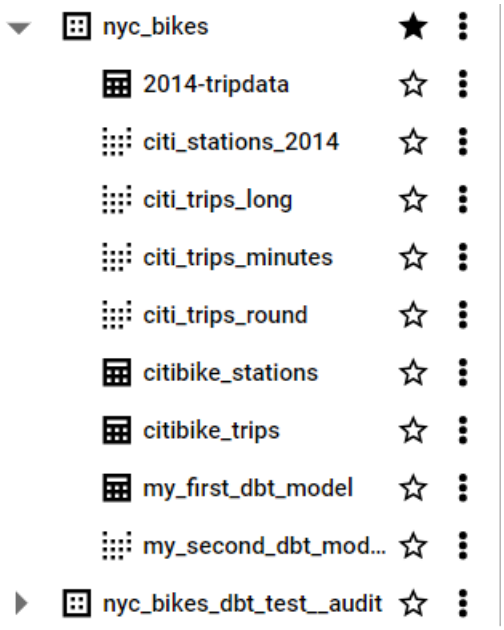


Figure 11.2: View from seed

11.3 Seed Configurations at project level

Though it may sound like there is a lot to do here, there actually isn't. Suffice to only say that seeds are configurable as much as our normal models are. There are two ways to configure seeds in dbt: either in the `dbt_project` YAML file or at the individual seed's YAML properties.

For the purposes of this exercise, at the project level we will set a dictionary that looks as follows:

```
seeds:
  dbt_book:
    2014-tripdata:
      schema: nyc2014_data
```

For any custom schema that we set, the result will be in the following format: `{{ target.schema }}_{{ schema }}`. That means the expected schema for our seed will be `nyc_bikes_nyc2014_data` –quite a mouthful of a name.

Thereafter run `dbt seed`.

```
-- snip --
19:24:43  Concurrency: 1 threads (target='dev')
19:24:43
19:24:43  1 of 1 START seed file nyc_bikes_nyc2014_data.2014-tripdata ..................... [RUN]
19:26:18  1 of 1 OK loaded seed file nyc_bikes_nyc2014_data.2014-tripdata ................. [INSER
19:26:18
19:26:18  Finished running 1 seed in 0 hours 1 minutes and 46.50 seconds (106.50s).
19:26:18
19:26:18  Completed successfully
19:26:18
19:26:18  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

Think of a schema as a folder or container for storing your data (read tables). Therefore, if you were in a company, there would be a schema (read it as folder or container) for sales, customers, products and clients. Inside the schema, (read folder or container) for sales, there would be tables for `january_sales`, `february_sales` and so on.

Just a *nota bene*, don't use hyphens `(-)` for your schema names, otherwise it will result in an error.

Here is our seed data appearing under the `nyc_bikes_nyc2014_data` dataset.

11.4 Seed properties and configurations at properties level

Seeds can also be configured at the properties level. In fact, the configurations at the properties level will override those set at the project level, that is at the `dbt_project` file.

To demonstrate setting seed configurations at the properties level, create `nyc_bikes2014` YAML file. Copy paste the following contents into the file.

```
version: 2

seeds:
  - name: 2014-tripdata
    description: "Seed for NYC 2014 bike data"
    docs:
```

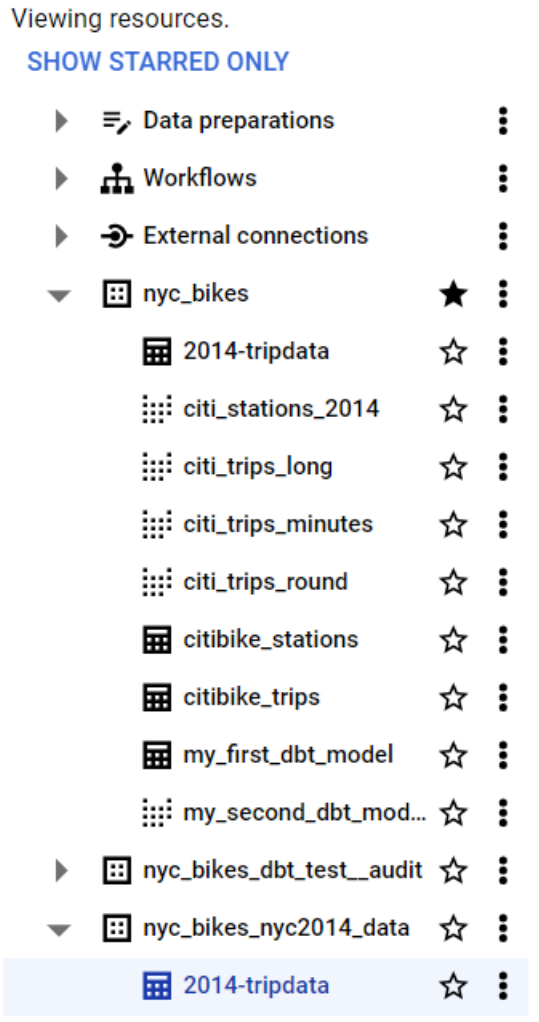


Figure 11.3: Schema for seeds

```
      show: true
      node_color: purple # Use name (such as node_color: purple) or hex code with quotes (such as
    config:
      schema: nyc_bikes2014
```

Not much different from the model properties' files we create, isn't it? In this case, the `name` of the model is not a SQL file but the csv we pushed to the data warehouse. The `docs` key is not so much important as the `config` key which we use to set the schema of our seed in the data warehouse.

What's good for the goose is good for the gander. Much akin to the model properties files were we can insert tests and documentation, the same goes for seed properties' files. In the contents of the below properties file of `nyc_bikes2014.yml` we have inserted documentation at both the table and column levels. We have also inserted tests at both levels as well.

```
version: 2

seeds:
  - name: 2014-tripdata
    description: '{{ doc("seed_2014_tripdata") }}'
    docs:
      show: true
      node_color: purple # Use name (such as node_color: purple) or hex code with quotes (such as
    config:
      schema: nyc_bikes2014
    tests:
      - dbt_expectations.expect_table_column_count_to_be_between:
          min_value: 1 # (Optional)
    columns:
      - name: _id
        description: 'Unique identifier'
        tests:
          - dbt_expectations.expect_column_values_to_be_unique

      - name: tripduration
        description: '{{ doc("tripduration") }}'
```

If you have additional seeds, simply add them to the properties files much like what we have in the `my_models.yml` which consists of the three models `citi_trips_minutes`, `citi_trips_round` and `citi_trips_long`.

11.5 Performing tests on seeds

Tests on seeds are performed in much the same way as other models. The only trick is to insert the name of the csv file. For example, to run a test of our `2014-tripdata.csv` which is our seed model, we execute `dbt test --select 2014-tripdata`.

```
-- snip --
18:19:53  Concurrency: 1 threads (target='dev')
18:19:53
18:19:53  1 of 2 START test dbt_expectations_expect_column_values_to_be_unique_2014-tri
18:19:57  1 of 2 PASS dbt_expectations_expect_column_values_to_be_unique_2014-tripdata_
18:19:57  2 of 2 START test dbt_expectations_expect_table_column_count_to_be_between_20
18:20:02  2 of 2 PASS dbt_expectations_expect_table_column_count_to_be_between_2014-tri
18:20:02
18:20:02  Finished running 2 data tests in 0 hours 0 minutes and 12.62 seconds (12.62s)
18:20:02
18:20:02  Completed successfully
-- snip --
```

As you can see, the two tests of `dbt_expectations.expect_table_column_count_to_be_between` and `dbt_expectations.expect_column_values_to_be_unique` passed.

11.6 Viewing documentation for dbt seeds

Even much less different than running tests is the generation of documentation regarding your dbt seeds. The process is exactly the same. First, run `dbt docs generate`. Assuming that the manifest files have successfully been created in the catalog, run `dbt docs serve`. If no errors appear at this final stage, open the port link that appears, such as `localhost:8080/`, on the terminal in your preferred browser.

You should see the documentation you created for your seed. The lineage graph should also work well for the seeds too.

Every seed at some point is left to grow on its own. We suppose this chapter has provided the necessary nutrition to see you bud to life working with dbt seeds.

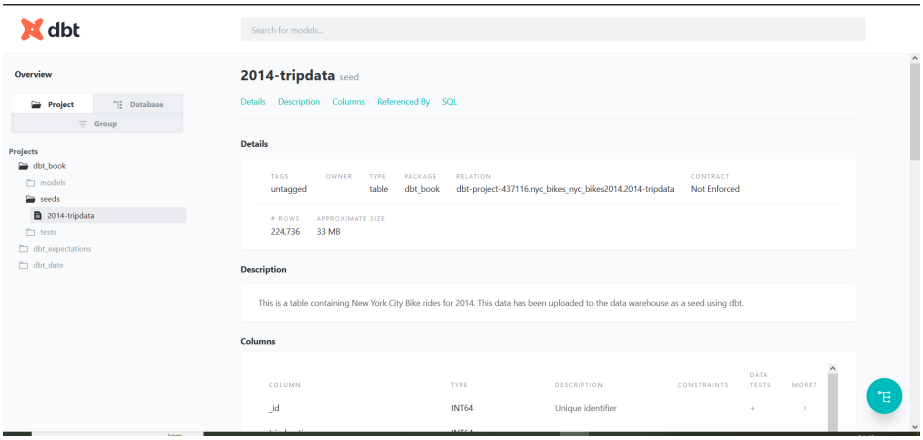


Figure 11.4: Documentation on seeds

Chapter 12

Sources

In the all-time favourite book *The voyages and adventures of Captain Hatteras*, every morning, the second-in-command, a well versed captain of the high seas, Richard Shandon by name, would receive a letter from an anonymous sender directing him on which direction to steer the ship. In dbt, sources are what make your data in the data warehouse be referenced in dbt operations such as running models, tests and checking the 'freshness' of your data.

Just like in the above anecdote where, if any person in the crew would ask Sir Richard Shandon for justification of any task they were commanded to do, Richard would always refer to the authoritative letter from an anonymous source. Likewise, when working with sources, dbt will perform operations by referencing the sources using the `source` function (`{{ source("schema", "table") }}`).

Sources in dbt are defined inside YAML files, and they are referenced inside SQL files, just like regular models again!

12.1 Defining a source

To demonstrate defining sources, we will work with two tables already in our data warehouse. These are the `2014-tripdata` and `citi_trips_round` tables under `nyc_bikes_nyc_bikes2014` (`nyc_bikes_nyc_bikes2014/2014-tripdata`) and `nyc_bikes` tree structures in BigQuery respectively.

Create a new sibling directory called `sources` next to the `docs`, `example`, and `my_models` folders. Inside it, create a new YAML file called `sources_bikes.yml`. The path to this file should be `models/sources/sources_bikes.yml`.

Copy paste these contents into the newly created YAML file.

```
version: 2

sources:
  - name: nyc_bikes_nyc_bikes2014
    schema: nyc_bikes_nyc_bikes2014
    tables:
      - name: 2014-tripdata

  - name: nyc_bikes
    schema: nyc_bikes
    tables:
      - name: citi_trips_round
```

The above should be all too familiar since we've worked with several YAML files so far. Nevertheless, the `name` and `schema` values refer to the schema names in your data warehouse. For the `tables` dictionary, we refer to the names of those tables under a particular schema. For example, the `citi_trips_round` is definitely under the `nyc_bikes` schema.

12.2 Referencing sources

Sources in our data warehouse are referenced using the `source()` function. Remember when referencing other models within models we used the `ref()` function? When working with sources, the `ref()` is now `source()`.

Below is a demonstration of referencing a source to only select male bike riders. Notice the arrangement of the schema and table names within quotation marks (`''`) and separated by a comma. This is how we reference other data acting as the *source* in our `nyc_bikes_male.sql`.

```
SELECT * FROM
{{ source('nyc_bikes', 'citi_trips_round') }}
WHERE gender = "male"
```

The same also works for data uploaded as a seed in our data warehouse as seen in the `nyc_male_2014.sql`.

```
SELECT * FROM
{{ source('nyc_bikes_nyc_bikes2014', '2014-tripdata') }}
WHERE gender = 1
```

We run these two specific models using `dbt run --select sources` and we get this output:

```
19:29:52  Concurrency: 1 threads (target='dev')
19:29:52
19:29:52  1 of 2 START sql view model nyc_bikes.nyc_bikes_male ........................... [RUN]
19:29:56  1 of 2 OK created sql view model nyc_bikes.nyc_bikes_male ...................... [CREAT
19:29:56  2 of 2 START sql view model nyc_bikes.nyc_male_2014 ............................ [RUN]
19:29:59  2 of 2 OK created sql view model nyc_bikes.nyc_male_2014 ....................... [CREAT
19:29:59
19:29:59  Finished running 2 view models in 0 hours 0 minutes and 12.75 seconds (12.75s).
19:30:00
19:30:00  Completed successfully
19:30:00
19:30:00  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

If you check under the `nyc_bikes` schema in your BigQuery, you will notice two new views have been created: `nyc_bikes_male` and `nyc_male_2014`.

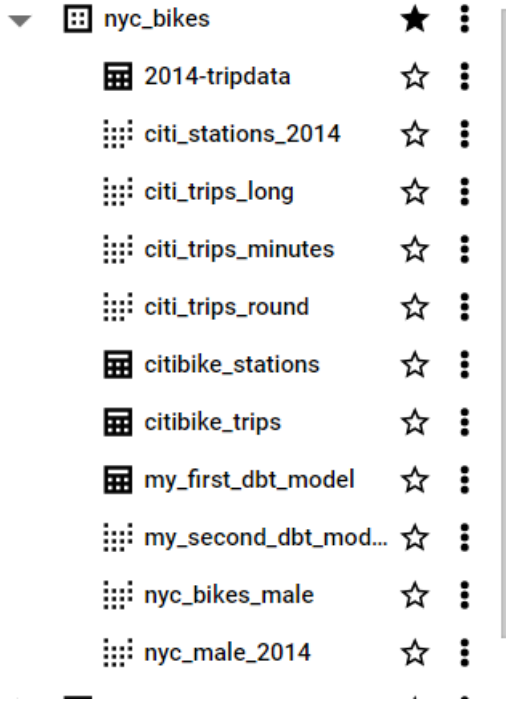


Figure 12.1: Sources

One would have expected the `nyc_male_2014` view to be under the `nyc_bikes_nyc_bikes2014` schema because that's the seed dataset. Our assumption is that we set the `nyc_bikes` as the dataset to work with when setting up dbt, and thus it's very hard to deviate from this. But we stand to

be corrected. One more thing, the dbt source can also work inside a `WITH` SQL statement like so in the `nyc_female_2014` model.

```
WITH nyc_female_2014 AS (
    SELECT * FROM
        {{ source('nyc_bikes_nyc_bikes2014', '2014-tripdata') }}
    WHERE gender = 2
)

SELECT * FROM nyc_female_2014
```

12.3 Defining properties in a `sources` file

Just like you would craft the properties for a given models' YAML file, the same can likewise be done for the sources YAML file. You can define descriptions and tests for your fields in a `sources` file. Again, what's good for the goose is good for the gander. Below is our enriched `sources` YAML file.

```
version: 2

sources:
  - name: nyc_bikes_nyc_bikes2014
    schema: nyc_bikes_nyc_bikes2014
    tables:
      - name: 2014-tripdata
        description: '{{ doc("tripduration") }}'
        columns:
          - name: _id
            description: 'Unique id'
            tests:
              - dbt_expectations.expect_column_values_to_not_be_null

          - name: tripduration
            description: '{{ doc("tripduration") }}'

          - name: starttime
            description: ''

          - name: stoptime
            description: ''

          - name: start station id
            description: ''
```

```
          - name: start station name
            description: ''

          - name: start station latitude
            description: ''

          - name: start station longitude
            description: ''

          - name: end station id
            description: ''

          - name: end station name
            description: ''

          - name: end station latitude
            description: ''

          - name: end station longitude
            description: ''

          - name: bikeid
            description: ''

          - name: usertype
            description: ''

          - name: birth year
            description: ''

          - name: gender
            description: ''

  - name: nyc_bikes
    schema: nyc_bikes
    tables:
      - name: citi_trips_round
```

Let's start by running the sole test at the trusty `tripduration` key via our single-line slingshot code: `dbt test --select sources`.

Everything ran fine meaning there were no null values in this field.

```
19:32:18  Concurrency: 1 threads (target='dev')
19:32:18
19:32:18  1 of 1 START test dbt_expectations_source_expect_column_values_to_not_be_null_nyc_bikes
```

```
19:32:21  1 of 1 PASS dbt_expectations_source_expect_column_values_to_not_be_null_nyc_
```

To see if our descriptions will be updated in the dbt documentation, simply run `dbt docs generate` followed by `dbt docs serve` to start the local server.

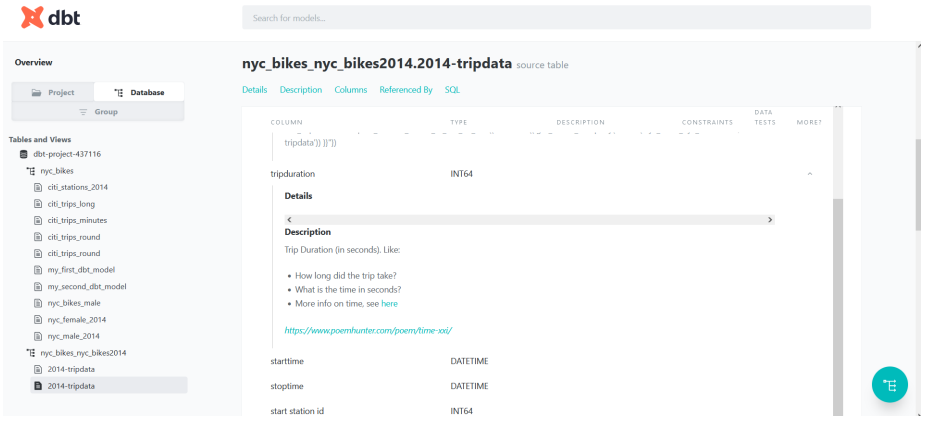


Figure 12.2: Sources descriptions

You should see your dbt documentation updated with the descriptions for `nyc_bikes_nyc_bikes2014` table.

Below is our `sources` YAML file in full with additional descriptions and tests.

```
version: 2

sources:
  - name: nyc_bikes_nyc_bikes2014
    schema: nyc_bikes_nyc_bikes2014
    tables:
      - name: 2014-tripdata
        description: '{{ doc("seed_2014_tripdata") }}'
        columns:
          - name: _id
            description: 'Unique id'
            tests:
              - dbt_expectations.expect_column_values_to_not_be_null

          - name: tripduration
            description: '{{ doc("tripduration") }}'

          - name: starttime
            description: ''
```

```
          - name: stoptime
            description: ''

          - name: start station id
            description: ''

          - name: start station name
            description: ''

          - name: start station latitude
            description: ''

          - name: start station longitude
            description: ''

          - name: end station id
            description: ''

          - name: end station name
            description: ''

          - name: end station latitude
            description: ''

          - name: end station longitude
            description: ''

          - name: bikeid
            description: ''

          - name: usertype
            description: ''

          - name: birth year
            description: ''

          - name: gender
            description: ''

  - name: nyc_bikes
    schema: nyc_bikes
    tables:
      - name: citi_trips_round
        description: '{{ doc("citi_trips_round") }}'
        tests:
          - dbt_expectations.expect_table_row_count_to_equal_other_table:
```

```
      compare_model: ref("citi_trips_minutes")
  - dbt_expectations.expect_column_pair_values_A_to_be_greater_than_B:
      column_A: tripduration
      column_B: trip_min_round
columns:
  - name: tripduration
    description: '{{ doc("tripduration") }}'

  - name: starttime
    description: '{{ doc("starttime") }}'

  - name: stoptime
    description: '{{ doc("stoptime") }}'

  - name: start_station_id
    description: "Start Station ID"
    tests:
      - dbt_expectations.expect_column_values_to_not_be_null

  - name: start_station_name
    description: "Start Station Name"
    tests:
      - dbt_expectations.expect_column_values_to_not_be_null
      - dbt_expectations.expect_column_value_lengths_to_be_between:
          min_value: 1 # (Optional)
          max_value: 70 # (Optional)

  - name: start_station_latitude
    description: "Start Station Latitude"

  - name: start_station_longitude
    description: "Start Station Longitude"

  - name: end_station_id
    description: "End Station ID"
    tests:
      - dbt_expectations.expect_column_values_to_not_be_null

  - name: end_station_name
    description: "End Station Name"
    tests:
      - dbt_expectations.expect_column_values_to_not_be_null
      - dbt_expectations.expect_column_value_lengths_to_be_between:
          min_value: 1 # (Optional)
          max_value: 70 # (Optional)
```

```
      - name: end_station_latitude
        description: "End Station Latitude"

      - name: end_station_longitude
        description: "End Station Longitude"

      - name: bike_id
        description: "Bike ID"

      - name: usertype
        description: "User Type (Customer = 24-hour pass or 7-day pass user, Subscriber = Ann

      - name: birth_year
        description: "Year of Birth"

      - name: gender
        description: "Gender (unknown, male, female)"
        tests:
          - dbt_expectations.expect_column_values_to_be_in_set:
              value_set: ['unknown','male','female']

      - name: customer_plan
        description: "The name of the plan that determines the rate charged for the trip"

      - name: trip_duration_min
        description: '{{ doc("trip_duration_min") }}'
        tests:
          - dbt_expectations.expect_column_max_to_be_between:
              min_value: 16 # (Optional)
              max_value: 326000 # (Optional)

      - name: trip_min_round
        description: '{{ doc("trip_min_round") }}'
        tests:
          - dbt_expectations.expect_column_max_to_be_between:
              min_value: 16 # (Optional)
              max_value: 100000 # (Optional)
```

Chapter 13

Snapshots

Picture this, there is this lady you have been eyeing, after turning and tossing for several nights, you decide to visit her place because she is currently not anywhere interested in being taken out. When you visit her at the rendezvous, you decide to ask her to take a 'selfie' of you and her (big mistake). You think she will send the selfie to you, she never does. Well, that's a true story of yours truly and even though no hard feelings over the incident, snapshots in dbt work in much the same way.

A snapshot in dbt is a recorded change of a mutable table. Think of a snapshot as a way to track changes in your data. For example, you could be a having a crazy table that logs your relationship status with your girlfriend or boyfriend over time. The first row could be as follows:

```
|----------|--------------|---------------|
| id       |  Status      | updated-at    |
| 11       |  Spark       | 21/09/2024    |
|----------|--------------|---------------|
```

Now, let's say you realise something about your girlfriend and boyfriend that puts a freeze on the relationship. So in your relationship table it would have the following update.

```
|----------|--------------|---------------|
| id       |  Status      | updated-at    |
| 11       |  Shaky       | 22/10/2024    |
|----------|--------------|---------------|
```

Our above update will have overwritten the previous record of 'Spark' when the relationship was at cloud nine. dbt offers a way to preserve these past records

so that they can be used for further analysis, or for posterity purposes. For example, keeping a record of the change can be used to analyse how long the relationship lasted from its hey days to when the waves started beating the ship. This kind of analysis can be used for more serious matters, such as when analyzing the time it takes from sending to receiving an order. dbt will help you record these changes and log the time when the change took place. For example, our dbt snapshot for our relationship would be:

|———-|———–|————–|————–-|————–| | id | Status | updated-at | dbt_valid_from | dbt_valid_to | | 11 | Spark | 21/09/2024 | 21/09/2024 | 22/10/2024 | | 11 | Shaky | 22/10/2024 | 22/10/2024 | null | |———-|———–|————–|————–-|————–|

The most up-to-date record will have a value of `null` in the `dbt_valid_to` field. Here is a description of the last two fields and those used internally by dbt.

1. `dbt_valid_from` - The timestamp when this snapshot row was first inserted. This column can be used to order the different "versions" of a record.

2. `dbt_valid_to` - The timestamp when this row became invalidated. The most recent snapshot record will have `dbt_valid_to` set to null.

3. `dbt_scd_id` - A unique key generated for each tracked record. This is used internally by dbt.

4. `dbt_updated_at` - The updated_at timestamp of the source record when this snapshot row was inserted. This is used internally by dbt.

Slowly Changing Dimension (SCD) refers to the way data changes over time in a data warehouse. In today's world, one wouldn't say that data changes slowly, but the term arises from the fact that even though data may change infrequently, such as makeups and breakups in your relationship, they are significant over time even to the future of the relationship or the continuity of your business!

SCDs are typically of three types:

- **Type 1**: This is where old data is overwritten without any preservation of its history. Old data ceases to exist with update of new data.

- **Type 2**: When a new record of data is added, the old record is preserved as historical data. This is the most common type of SCD and which dbt implements.

- **Type 3**: This approach adds a new column for the new data and preserves the old data in the original column. This type is best used to see the progression of changes *rather* than when a change happened.

Therefore, when *snapshoting* in dbt, when a change occurs in the source data, instead of overwriting the existing record (Type 1) or a dding a new column (Type 3), dbt adds a new record with the new data (Type 2). The `dbt_valid_from` and `dbt_valid_to` columns in the snapshot table indicate when each version of the record was valid, allowing you to track the full history of changes over time. This looks much like git commits, only that the commits are in tabular form.

13.1 Create a snapshot

Creating a snapshot in dbt to some extent depends on the version you are using. Starting from version 1.9, you will actually need two files to perform a dbt snapshot. These are the YAML and sql files. However, this tutorial was written using version 1.8.7. To know the dbt version you are using, type `dbt debug`. You will see your version listed like so:

```
--snip--
19:41:25  Running with dbt=1.8.7
19:41:25  dbt version: 1.8.7
19:41:25  python version: 3.10.12
--snip--
```

Now to create a snapshot using dbt versions lower than for 1.9, you will create a snapshot SQL file with the following configurations.

```
{% snapshot tripdata_snapshot %}

{{
  config(
    target_schema='snapshots',
    strategy='check',
    unique_key='_id',
    check_cols='all'
  )
}}

SELECT * FROM {{ source('nyc_bikes_nyc_bikes2014', '2014-tripdata') }}

{% endsnapshot %}
```

Let's go through it line by line. The macros `{% snapshot tripdata_snapshot %}` and `{% endsnapshot %}` indicate that this is a snapshot file. Your configurations will go inside the `config()` function.

`target_schema` - this is the schema in which your snapshot will be stored.

`strategy` - this is the mechanism by which dbt will know that a row has changed. The `timestamp` strategy, and the most recommended for that matter, uses an `updated_at` column to determine if a row has changed. On the other hand, the `check` strategy compares a list of columns between their current and historical state to determine what has changed. Use the `check` strategy if there is no reliable `updated_at` column for tracking changes with time, as in our case.

`unique_key` - this is the unique key in your table that dbt will use.

`check_cols` - These are the columns to check for changes. The `all` parameter can be passed in case you want to track changes in all the columns of the row.

One can also add an additional `invalidate_hard_deletes` configuration to track rows that have been deleted. The `dbt_valid_to` column of deleted rows will be set to the current snapshot time.

Finally, the `SELECT` statement. You will insert inside the `source()` function the table in which you would like to track changes.

Thereafter, run `dbt snapshot`. Below is the output.

```
21:01:01  Concurrency: 1 threads (target='dev')
21:01:01
21:01:01  1 of 1 START snapshot snapshots.tripdata_snapshot ..........................
21:01:11  1 of 1 OK snapshotted snapshots.tripdata_snapshot ..........................
21:01:11
21:01:11  Finished running 1 snapshot in 0 hours 0 minutes and 17.65 seconds (17.65s).
21:01:12
21:01:12  Completed successfully
21:01:12
21:01:12  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

A new table should appear under the `snapshots` schema in BigQuery.

When you run `dbt snapshot` the first time, the `dbt_valid_to` column will be `null` for all records. Thereafter, when you run subsequent `dbt snapshot` executions for a table that has undergone a change, the `dbt_valid_to` will be populated with a timestamp value in the `dbt_valid_to` of the altered row.

13.2 The `check` strategy

Now is the time to truly test if our snapshots work. Go to the SQL tab of your Big Query and insert a new row using this query:

```
INSERT INTO `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata` (`_id`, `tri
VALUES (000000, 1000, 'Nowhere Near Station');
```

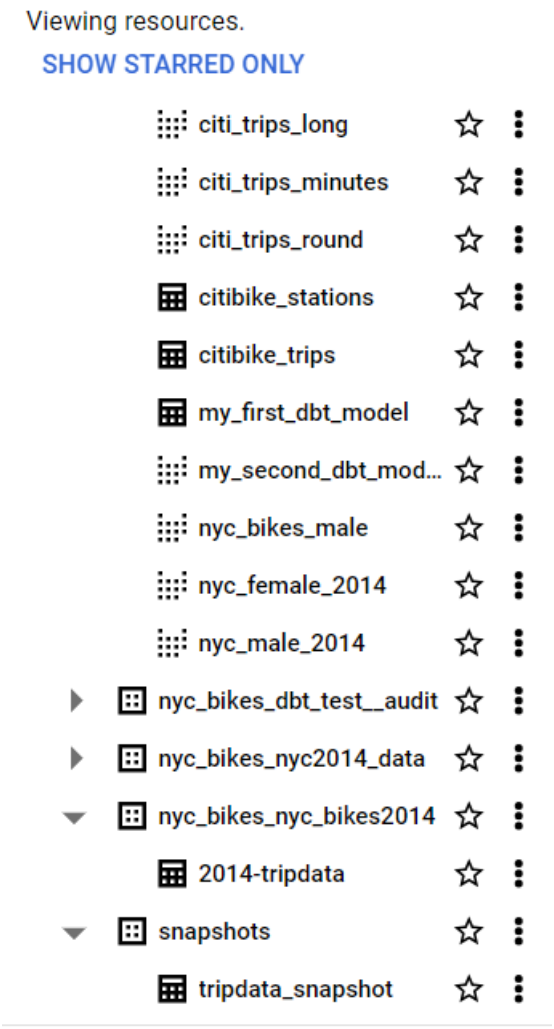


Figure 13.1: Snapshots schema

Thereafter run `dbt snapshot`.

Ensure that the new row has been added by crosschecking its existence via:

```
SELECT * FROM `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata`
WHERE `start station name` = 'Nowhere Near Station';
```

Now check if our `tripdata_snapshot` table has captured the new row using the below query.

```
SELECT * FROM `dbt-project-437116`.`snapshots`.`tripdata_snapshot`
WHERE `start station name` = 'Nowhere Near Station';
```

Its a new row of data, which means all columns have been affected with a new value in each. Remember we set the `check_cols=all`.

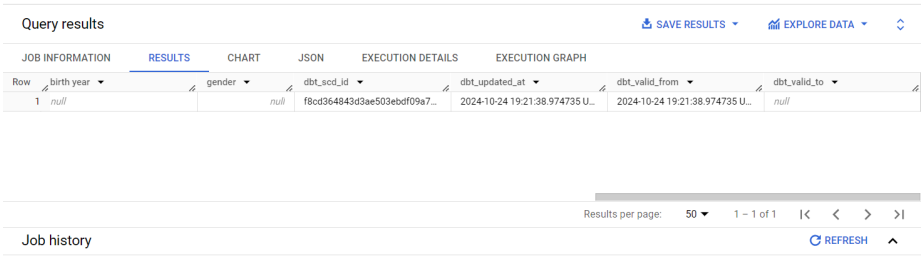


Figure 13.2: Recorded change

Let's go on.

Insert a new row, with an additional extra change in the `_id` column.

```
UPDATE `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata`
SET `start station name` = 'Even Further Station', `_id` = 1001995
WHERE `_id` = 0;
```

Again, run `dbt snapshot`. Always run `dbt snapshot` when your data has received new data update.

Let's check if the new row with two updates has been recorded in our snapshots table.

```
SELECT * FROM `dbt-project-437116`.`snapshots`.`tripdata_snapshot`
WHERE `start station name` = 'Even Further Station';
```

If you run this, you will notice that the `dbt_valid_to` is still `null`. This could possibly be because we have added a new unique key and thus dbt will still treat this as a new record rather than one which was changed from 0 to 1001995.

Now, update the row with `_id` 1001995 using the below SQL query.

```
UPDATE `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata`
SET `start station name` = 'Furth East Station'
WHERE `_id` = 1001995;
```

Now after running `dbt snapshot` , let's see if our snapshot table will have tracked the historical change of `Even Further Station` and `Furth East Station` of row `_id` 1001995. Use the below query to unravel the results.

```
SELECT * FROM `dbt-project-437116`.`snapshots`.`tripdata_snapshot`
WHERE `_id` = 1001995;
```

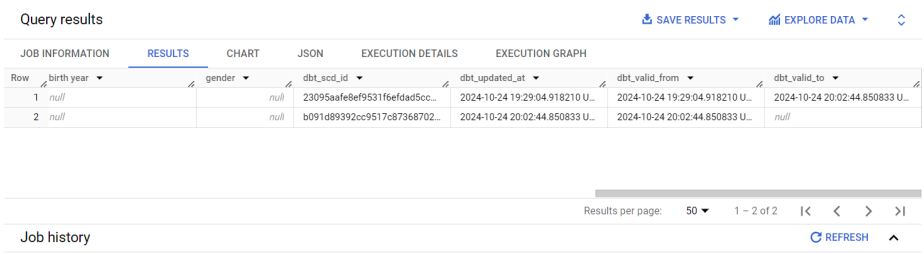


Figure 13.3: Snapshot example

Yes it did! For row 1, which stands for when the 'start station name' was `Even Further Station`, we can see that row was valid from 2024-10-24 19:29 to 2024-10-24 20:02. However, the new row 2, which is where the 'start station name' was switched to `Furth East Station`, we can see it became valid from 2024-10-24 20:02; the exact time when row was updated.

You can indeed check if the latest change is in the `2014-tripdata` table via:

```
SELECT * FROM `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata`
WHERE `start station name` = 'Furth East Station';
```

13.3 The `timestamp` strategy

The `timestamp` strategy in snapshoting relies on an `updated_at` column to check if any changes have occurred on the row. If the configured `updated_at`

column is more recent than when the table was last run, dbt will invalidate the old record and record a new one. If the timestamps are unchanged, dbt will not take any action.

The `timestamp` strategy requires an `updated_at` column which represents when the row was last updated. In order to work with `timestamp` strategy, we need to recreate our `2014-tripdata` but now with an additional `updated_at` column. It can be any table, so long as there is an `updated_at` column, but we settled on this one because it is lightweight. Plus, we already have it as a seed. We will use a dbt model to recreated the `2014-tripdata` seed but with an extra `updated_at` column.

Create a `nyc_bikes_timestamp` SQL model inside the `sources` folder. Copy paste the following contents into the model.

```
{{ config(
    materialized="table",
    schema="nyc_bikes_nyc_bikes2014"
) }}

WITH nyc_bikes_timestamp AS (
  SELECT *, CURRENT_TIMESTAMP() AS updated_at FROM {{ source('nyc_bikes_nyc_bikes2014'
)

SELECT
  *
FROM nyc_bikes_timestamp
```

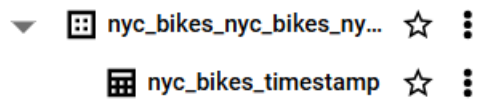


Figure 13.4: Timestamp table

We are configuring it as a table because for some reason, when trying to update fields into this model using BigQuery, an error came up simply because it was a view!

Now run `dbt run --select sources`. You will get the below output.

```
19:08:14  Concurrency: 1 threads (target='dev')
19:08:14
19:08:14  1 of 4 START sql view model nyc_bikes.nyc_bikes_male .....................
19:08:16  1 of 4 OK created sql view model nyc_bikes.nyc_bikes_male ................
19:08:16  2 of 4 START sql table model nyc_bikes_nyc_bikes_nyc_bikes2014.nyc_bikes_time
```

```
19:08:22  2 of 4 OK created sql table model nyc_bikes_nyc_bikes_nyc_bikes2014.nyc_bikes_timestamp
19:08:22  3 of 4 START sql view model nyc_bikes.nyc_female_2014 ..... [RUN]
19:08:24  3 of 4 OK created sql view model nyc_bikes.nyc_female_2014 ..... [CREAT
19:08:24  4 of 4 START sql view model nyc_bikes.nyc_male_2014 ..... [RUN]
19:08:26  4 of 4 OK created sql view model nyc_bikes.nyc_male_2014 ..... [CREAT
19:08:26
19:08:26  Finished running 3 view models, 1 table model in 0 hours 0 minutes and 29.78 seconds (2
19:08:26
19:08:26  Completed successfully
19:08:26
19:08:26  Done. PASS=4 WARN=0 ERROR=0 SKIP=0 TOTAL=4
```

Now that we have already created a table of `nyc_bikes_timestamp`, we would also want to reference it in downstream models. As you read in an earlier chapter, dbt sources are what make models to be referenced in other queries using the `source()` function. Therefore in the `sources/sources_bikes.yml`, add the following:

```
- name: nyc_bikes_nyc_bikes_nyc_bikes2014
        schema: nyc_bikes_nyc_bikes_nyc_bikes2014
        tables:
          - name: nyc_bikes_timestamp
            description: ''
```

Now is the time to create a dbt snapshot relying on the `timestamp` strategy.

Create a `timestamp_snapshot` in the `snapshots` directory with the following SQL contents.

```
{% snapshot timestamp_snapshot %}

{{
  config(
    target_schema='snapshots',
    strategy='timestamp',
    unique_key='_id',
    updated_at='updated_at'
  )
}}

SELECT * FROM `dbt-project-437116`.`nyc_bikes_nyc_bikes_nyc_bikes2014`.`nyc_bikes_timestamp`

{% endsnapshot %}
```

You may wonder why we are not using something like `{{ source("schema", "table") }}` in the SELECT statement. We had initially run that with

`nyc_bikes_nyc_bikes_nyc_bikes2014` and `nyc_bikes_timestamp` as the *schema* and *table* names respectively. However, dbt kept throwing an error that it couldn't find such a table therefore we resulted in the unorthodox way of hardcoding the entire dataset-schema-table namespace.

Now run `dbt snapshot` to create the `nyc_bikes_timestamp` table.

```
19:31:54  Concurrency: 1 threads (target='dev')
19:31:54
19:31:54  1 of 2 START snapshot snapshots.timestamp_snapshot .....................
19:32:02  1 of 2 OK snapshotted snapshots.timestamp_snapshot ......................
19:32:02  2 of 2 START snapshot snapshots.tripdata_snapshot .......................
19:32:15  2 of 2 OK snapshotted snapshots.tripdata_snapshot .......................
19:32:15
19:32:15  Finished running 2 snapshots in 0 hours 0 minutes and 28.86 seconds (28.86s)
19:32:15
19:32:15  Completed successfully
19:32:15
19:32:15  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

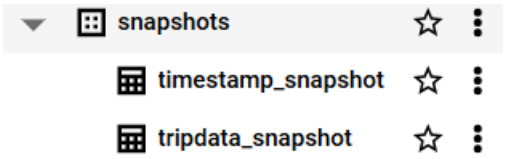


Figure 13.5: Timestamp snapshot

Now to check if our timestamp table can snapshot changes, let's insert a new row and make some changes to it. Paste the following in a SQL tab in BigQuery.

```
INSERT INTO `dbt-project-437116`.`nyc_bikes_nyc_bikes_nyc_bikes2014`.`nyc_bikes_timesta
VALUES (21001995, 2000, 'Sumwhere Near Station', '2024-10-25 23:09:47.169668 UTC');
```

Note the `updated_at` column. Unlike when working with the `check` strategy which could still work well with several fields as `null`, omitting the `updated_at` column in the `timestamp` strategy is costly as dbt will be unable to track any change. All you will get is just a new field but with several `null` values in the snapshot table.

Now run `dbt snapshot` and when it succesfully runs, check `timestamp_snapshot` table using this SELECT statement in BigQuery.

```
SELECT * FROM `dbt-project-437116`.`snapshots`.`timestamp_snapshot`
WHERE `_id` = 21001995;
```

Now change the station name from ‘Sumwhere Here Station‘ to ’Somewhere Near Station’ to demonstrate tracking a change.

```
UPDATE `dbt-project-437116`.`nyc_bikes_nyc_bikes_nyc_bikes2014`.`nyc_bikes_timestamp`
SET `start station name` = 'Somewhere Here Station', `updated_at` = '2024-10-25 23:14:47.169668 U
WHERE `_id` = 21001995;
```

Run `dbt snapshot`.

Now check if dbt has been able to track changes. We expect that the row with the station name ‘Sumwhere Near Station’ was valid for a short period (see the `dbt_valid_from` and `dbt_valid_to` columns) while the ‘Somewhere Here Station’ is the most current.

```
SELECT * FROM `dbt-project-437116`.`snapshots`.`timestamp_snapshot`
WHERE `_id` = 21001995;
```

You should see we’ve been able to track changes.

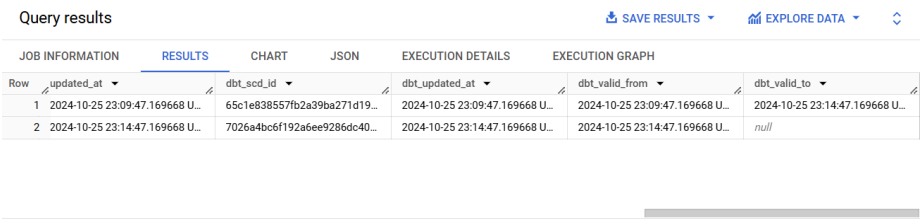


Figure 13.6: Timestamp tracking

The downside of using the `timestamp` strategy is that you have to use the `updated_at` column or whatever timestamp column you defined. Nevertheless, based on our exercise so far, the `check` strategy is far much better and less taxing.

Chapter 14

Analyses

In dbt, analyses are those kind of queries that might not exactly be very much needed as a model but can be used for data exploration and visualization. Typically, any SQL model that is for analytical rather than modeling purposes is stored within the `analyses` directory. Thereafter, running the code `dbt compile` will create the compiled SQL file inside the `target/compiled/{project name}/analyses/sql_file_name.sql` directory. The code inside this directory can be pasted in a data visualization tool but it will not appear in the data warehouse. You read that well: it will not.

14.1 Creating an `analysis`

As mentioned earlier, analyses queries are stored within the `analysis` folder. To begin with, we shall create a SQL query that performs a join. The below query will join the rows in `nyc_bikes_nyc_bikes2014.2014-tripdata` with those in `nyc_bikes.citi_trips_round` based on station id. The aforementioned contents are found within the `start_join_bikes` SQL within the `analyses` folder.

```
WITH `2014-tripdata` AS (
    SELECT * FROM {{ source('nyc_bikes_nyc_bikes2014', '2014-tripdata') }}
),

WITH citi_trips_round AS (
    SELECT * FROM {{ source('nyc_bikes', 'citi_trips_round') }}
)

SELECT cs.`start station id`, cs.`start station name`,
ct.bikeid, ct.start_station_id, ct.trip_min_round
FROM `2014-tripdata` cs
```

```
JOIN citi_trips_round ct
ON cs.`start station id` = ct.start_station_id
WHERE ct.trip_min_round > 50000
```

Thereafter type and hit `dbt compile` on the terminal. The results will appear in the `dbt_book/target/compiled/dbt_book/analyses` directory. The query is actually the same, as you can see below. The only exception is that the references within the `source` file have been expanded to contain the full table path.

```
WITH `2014-tripdata` AS (
    SELECT * FROM `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata`
),

WITH citi_trips_round AS (
    SELECT * FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_round`
)

SELECT cs.`start station id`, cs.`start station name`,
ct.bikeid, ct.start_station_id, ct.trip_min_round
FROM `2014-tripdata` cs
JOIN citi_trips_round ct
ON cs.`start station id` = ct.start_station_id
WHERE ct.trip_min_round > 50000
```

However, pasting this on BigQuery results in the below error.

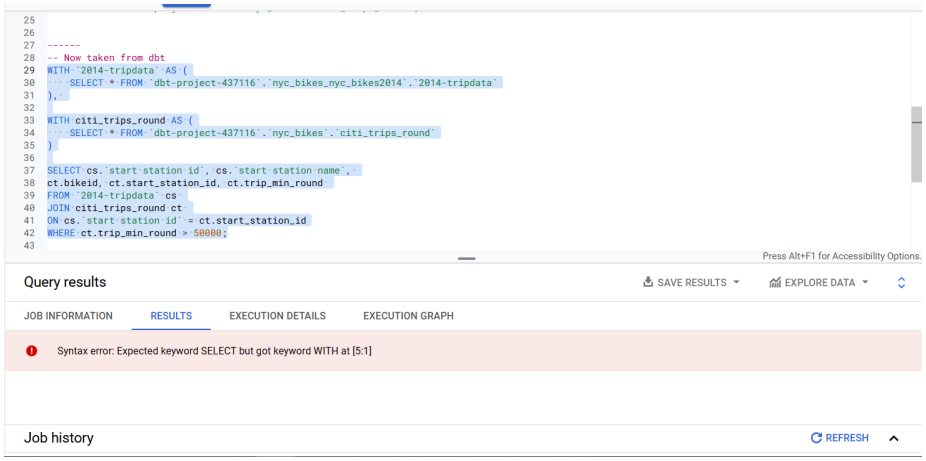


Figure 14.1: Analyses error

Therefore, we created a second model which does the same work but without the `WITH` statement. These are the contents of `station_join_bikes_snd.sql`.

```
SELECT cs.`start station id`, cs.`start station name`,
ct.bikeid, ct.start_station_id, ct.trip_min_round
FROM {{ source('nyc_bikes_nyc_bikes2014', '2014-tripdata') }} cs
JOIN {{ source('nyc_bikes', 'citi_trips_round') }} ct
ON cs.`start station id` = ct.start_station_id
WHERE ct.trip_min_round > 50000;
```

The contents of the SQL file inside the `dbt_book/target/compiled/dbt_book/analyses` directory are as follows:

```
SELECT cs.`start station id`, cs.`start station name`,
ct.bikeid, ct.start_station_id, ct.trip_min_round
FROM `dbt-project-437116`.`nyc_bikes_nyc_bikes2014`.`2014-tripdata` cs
JOIN `dbt-project-437116`.`nyc_bikes`.`citi_trips_round` ct
ON cs.`start station id` = ct.start_station_id
WHERE ct.trip_min_round > 50000;
```

The full table path was expanded and pasting this query into BigQuery produces the results without a fuss.

14.2 Definitions for `analyses`

The definitions for `analyses` are created the same way as the other YAML files we have created for other models, only that this time they are within the `analyses` folder.

Chapter 15

Exposures

Imagine a soldier dropping a piece of paper containing their camp location, trails and military equipment only for it to be picked up by a wandering enemy. That would be catastrophic, right? That is akin to exposing them into the line of fire. However, exposures in dbt serve a good purpose. They show how your data is used by downstream projects, be they be notebooks, a dashboard or another data pipeline. They only thing that sets apart exposures from other models is that this time round you are the one who defines which projects will be used downstream. For example, if you want to show your CEO which models were used to create the dashboard, instead of showing the ten models you sifted through, you only show the three that made it to the dashboard.

15.1 Creating an exposure

Exposures are written in YAML files but nested under the `exposures:` key. Below is an exposure created in the `exposure/exposures.yml` path.

```
version: 2

exposures:

  - name: station_bikes_exposure
    label: A join of station tables and bike rides
    type: dashboard
    maturity: high
    url: https://public-toilets-in-australia-infomap.onrender.com/
    description: '{{ doc("citi_trips_round") }}'

    depends_on:
```

```
      - ref('citi_trips_round')
      - ref('citi_trips_minutes') # Added this just to increase complexity of lineage g
      - source('nyc_bikes_nyc_bikes2014', '2014-tripdata')

    owner:
      name: Mr Fantastic
      email: mrfantastic@unlike.com
```

Below is the definition of each property used above.

Required

- `name`: a unique exposure name written in snake case
- `type`: one of dashboard, notebook, analysis, ml or application
- `owner`: name or email required; additional properties allowed

Expected

`depends_on`: list of nodes, including `metric`, `ref`, and `source`. While possible, it is highly unlikely you will ever need an `exposure` to depend on a `source` directly.

Optional

- `label`: May contain spaces, capital letters, or special characters.
- `url`: Activates and populates the link to View this exposure in the upper right corner of the generated documentation site
- `maturity`: Indicates the level of confidence or stability in the exposure. One of high, medium, or low. For example, you could use high maturity for a well-established dashboard, widely used and trusted within your organization. Use low maturity for a new or experimental analysis.

General properties (optional)

- description
- tags
- meta

15.2 Running an exposure

To run the exposure we just created, we use the following one liner. The plus sign is there to indicate to dbt to include all models used to feed into the `station_bikes_exposure` exposure.

```
dbt run --select +exposure:station_bikes_exposure
```

It is noteworthy to mention that you run the `name` value under the `exposures` key. The name of the YAML is not used when running exposures.

Here is the output:

```
20:31:06  Concurrency: 1 threads (target='dev')
20:31:06
20:31:06  1 of 2 START sql view model nyc_bikes.citi_trips_minutes ..... [RUN]
20:31:09  1 of 2 OK created sql view model nyc_bikes.citi_trips_minutes .... [CREAT
20:31:09  2 of 2 START sql view model nyc_bikes.citi_trips_round ..... [RUN]
20:31:12  2 of 2 OK created sql view model nyc_bikes.citi_trips_round ..... [CREAT
20:31:12
20:31:12  Finished running 2 view models in 0 hours 0 minutes and 13.36 seconds (13.36s).
20:31:12
20:31:12  Completed successfully
20:31:12
20:31:12  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

One can also decide to test all the upstream models for our exposure. The below code will display test results for all the three models defined by the `depends_on` key.

```
19:37:44  Concurrency: 1 threads (target='dev')
19:37:44
19:37:44  1 of 23 START test dbt_expectations_expect_column_max_to_be_between_citi_trips_minutes_
19:37:49  1 of 23 PASS dbt_expectations_expect_column_max_to_be_between_citi_trips_minutes_trip_d
19:37:49  2 of 23 START test dbt_expectations_expect_column_max_to_be_between_citi_trips_round_tr
19:37:53  2 of 23 PASS
-- snip --
```

Nevertheless, we remain with visualizing our exposure.

15.3 Visualizing the exposure

Visualizing exposures is as simple as just generating your dbt documentation. This is actually the default way of displaying exposures. It begins with `dbt docs`

`generate` and `dbt docs serve`. Afterwards, open the dbt documentation static web page in the port number provided. Ours is `localhost:/8080`.

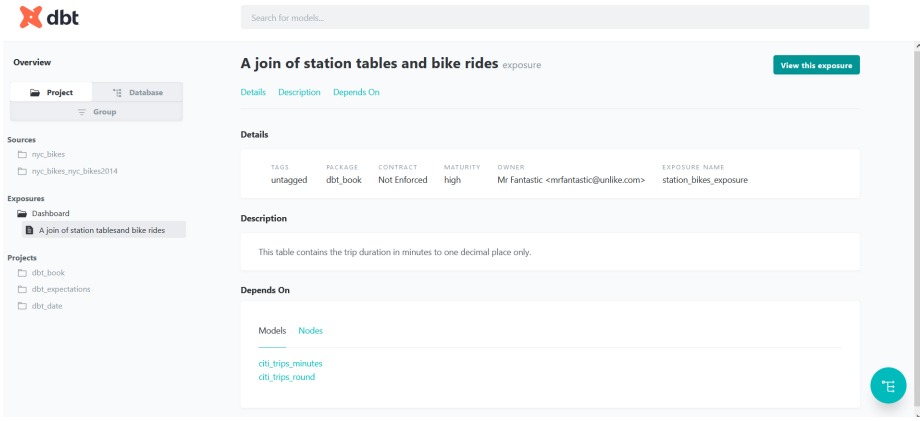


Figure 15.1: Exposure

You will notice that there is a dedicated section for exposures called `Exposures`. The exposure title is `Dashboard` and the label for the exposure is the `label` value provided in the YAML.

If you click on the blue lineage graph button, you will see the upstream models that feed into our exposure.

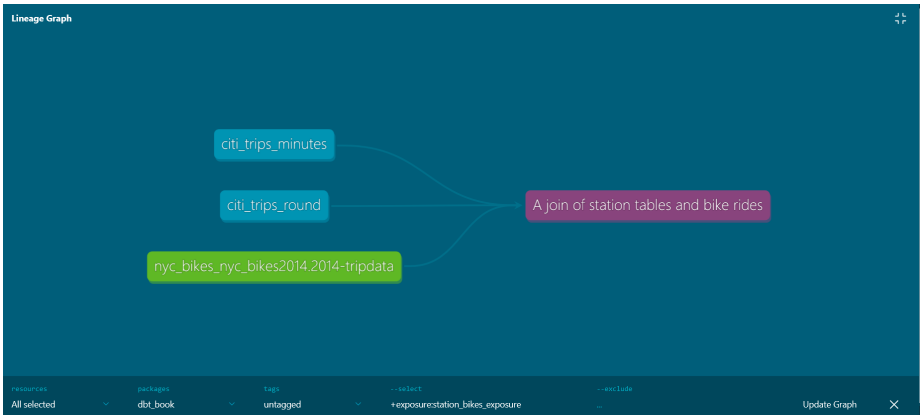


Figure 15.2: Exposure lineage graph

Still at the bottom of the webpage, the **Depends on** section contains links to all the upstream models and references for your exposure. Clicking on any takes you to the dbt documentation site for that model.

Finally, there is the **View this exposure** button. Clicking on it will take you

to the url you specified in the `url:` key of the exposures file. In our case, the url leads to a Dashboard showing all the public sanitation facilities in Australia. Obviously there is no relation between bikes and sanitation facilities. This was just for demonstration purposes only!

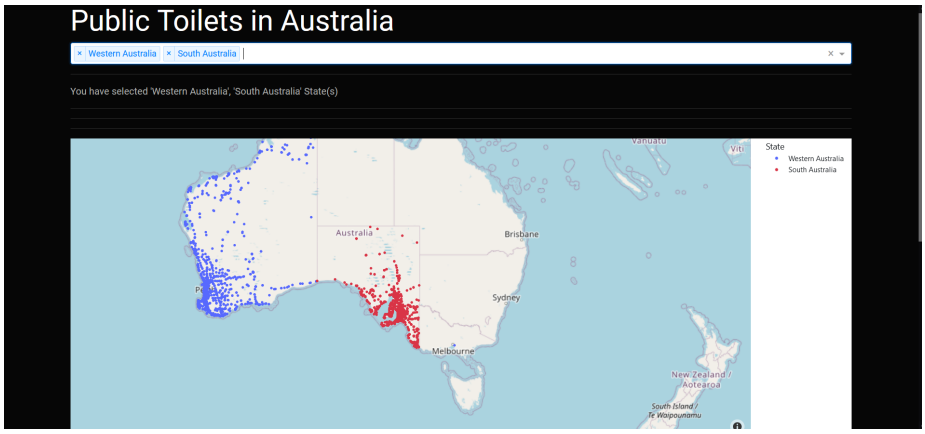


Figure 15.3: Exposure

Working with exposures can be fun. You can add as many exposures as you wish. Below we extended the `exposures` YAML to also include the following `station_bikes_application` exposure.

```
-- snip --
- name: station_bikes_application
    label: An app of stations and bike rides
    type: application
    maturity: high
    url: https://data-visualization-for-diarrhoea-deaths.onrender.com/
    description: '{{ doc("citi_trips_round") }}'

    depends_on:
      - ref('citi_trips_long')
      - ref('citi_trips_minutes') # Added this just to increase complexity of lineage graph
      - source('nyc_bikes_nyc_bikes2014', '2014-tripdata')

    owner:
      name: Mr Fantastic
      email: mrfantastic@unlike.com
```

Once again, to include this exposure, we run `dbt run --select +exposure:station_bikes_application`. Thereafter create a documentation using the two sesame magic characters of `dbt docs generate` and `dbt docs serve`.

The above exposure of `station_bikes_application` falls under the **Application** section as specified in the `type` key.

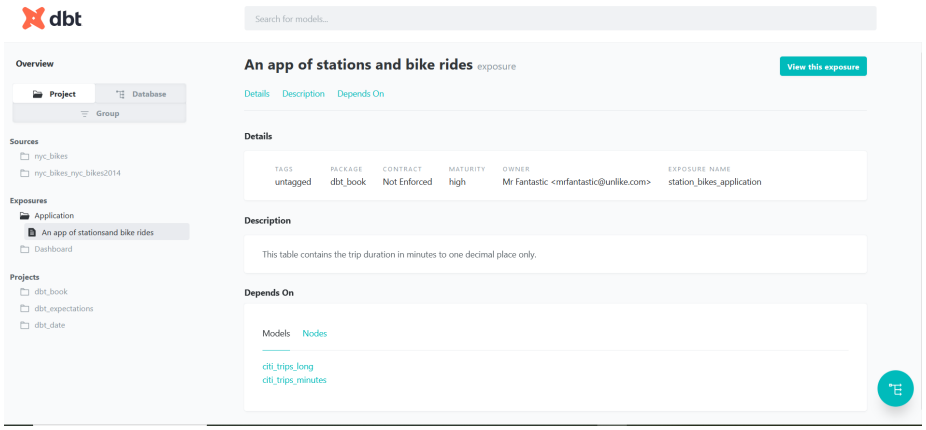


Figure 15.4: Exposure application

The lineage graph and the **View this exposure** buttons work for this exposure as well. In fact this lineage graph is the most complex we've encountered in the course so far. The exposure took into consideration that the `citi_trips_long` model is dependent on the `citi_trips_minutes` and `citi_trips_round` models!

Figure 15.5: Exposure lineage graph

The exposure button also leads to a dashboard showing rates of some sanitation related disease. Again, this dashboard is not related to bikes and stations but serves the purpose of demonstration.

Chapter 16

Jinja

Jinja in dbt is used to perform functions that regular SQL is unable to do, such as iterating over columns using a `for` loop and also `if` statements. Jinja is also used to hold environment variables which can be (re)used all over your project. A simple example of jinja use in dbt is when calling the `ref()` function. If you see anything with double curly brackets (`{{ }}`) or with brackets and percentage sign(s) in them (`{% %}`) then you are dealing with jinja.

Let's start by explaining some jinja concepts.

- **Expressions {{ ... }}**: Expressions are used when you want to output a string. You can use expressions to reference variables and call macros.
- **Statements {% ... %}**: Statements don't output a string. They are used for control flow, for example, to set up for loops and if statements, to set or modify variables, or to define macros.
- **Comments {# ... #}**: Jinja comments are used to prevent the text within the comment from executing or outputing a string.

Let's start with explaining a *for* loop. This is the skeleton of a `for` loop in jinja.

```
{% for item in sequence %}
-- SQL Code for {{ item }}
{% endfor %}
```

16.1 A simple jinja statement

Going by the above cue, let's create a jinja statement that will select all those rows whose bike riders' birth years are any of the following: 1995, 1997 and 2002.

Nothing special about these years, just that they stem from the unprofitable notion that they correspond to my birth year and those of my siblings!

Create a new folder called `jinja` under the `models` directory and within it create a `years` SQL file. Copy paste the following contents into `years.sql`.

```
{% set years = [1995, 1997, 2002] %}

{% for year in years %}
SELECT *
FROM {{ ref('citi_trips_long') }}
WHERE birth_year = {{ year }}

{% endfor %}
```

Let's go through the above line by line.

- `{% set years = [1995, 1997, 2002] %}` - this sets the variables we will use to extract some data from our tables. The `years` variable consists the years that we will use to filter our tables.
- `{% for year in years %}` - the SQL statement that will be repeated is placed inside the `for ...` loop. In this statement, for every year in the `years` variable list, we will repeat the below sql statement, where `{{ year }}` is each year in the `years` variable:

```
SELECT *
FROM {{ ref('citi_trips_long') }}
WHERE birth_year = {{ year }}
```

- `{% endfor %}` - nothing more than marking the end of the for loop.

Now, if we run the code `dbt compile --select jinja`, you will see a new `years.sql` appear under the `target/compiled/dbt_book/models/jinja/` directory.

Here is how the code looks like:

```
SELECT *
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year = 1995



SELECT *
```

```
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year = 1997



SELECT *
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year = 2002
```

What the `for` loop did was to avoid the redundancy of selecting each `birth_year` in its own `SELECT` statement. Instead it put the three `SELECT` statements inside one `for` loop statement. Going a step further, also the redundancy of hardcoding the `year` is removed. This formula, though a bit complicated, toes in line with the Do not Repeat Yourself (DRY) principle in programming.

16.2 A more complex jinja query

One can also create more complex jinja queries that leverage other SQL functionalities such as aggregation and CASE WHEN statements. Now suppose, for purely selfish reasons, this author wants to compare the bike ride durations against those of other people who were born in the years 1995, 1997 and 2002. Below is an `age.sql` file that creates a table that has a column showing the trip duration for each value in the `years` variable.

```
{% set years = [1995, 1997, 2002] %}

SELECT birth_year,
{% for year in years %}
SUM (CASE WHEN birth_year = {{ year }} THEN trip_min_round ELSE 0 END) AS trip_min_round_{{ year
{% endfor %}
SUM(trip_min_round) AS totals_trip_min_round
FROM {{ ref('citi_trips_long') }}
GROUP BY birth_year
```

In the corresponding `age.sql` in the `target` directory, this is the compiled SQL query result.

```
SELECT birth_year,

SUM(CASE WHEN birth_year = 1995 THEN trip_min_round ELSE 0 END) AS trip_min_round_1995,

SUM(CASE WHEN birth_year = 1997 THEN trip_min_round ELSE 0 END) AS trip_min_round_1997,
```

```
SUM(CASE WHEN birth_year = 2002 THEN trip_min_round ELSE 0 END) AS trip_min_round_2002

SUM(trip_min_round) AS totals_trip_min_round
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year IN (

    1995,

    1997,

    2002

)
GROUP BY birth_year
```

Pasting this query in BigQuery gives us the below table:

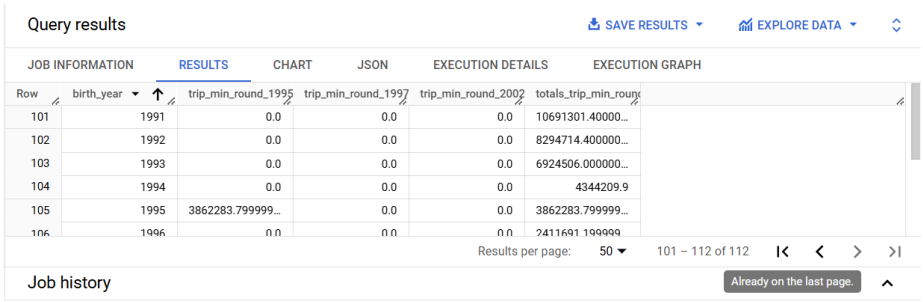


Figure 16.1: Age query

We can see a column for trip duration in minutes for each of the three select years of 1995, 1997 and 2002. However, other years not set in our `years` variable are included as well, but with the value `0` in each of the three columns. Before one starts getting confused and blaming the universe for not wanting us to succeed, there is a way we can sort this pesky issue: enter the `if not loop.last` statement!

In our previous SQL query, you saw that one can sort the issue of excluding unnecessary years using the `WHERE` clause. For example, we would have used the clause `WHERE birth_year IN (1995, 1997, 2002)`. However, we frowned on this approach because it is breaking the DRY principle by hardcoding the years by hand. Taking a more complex approach to fulfil the DRY principle

sounds like we are exhibiting Obsessive Compulsive Disorder (OCD) but being obsessed in doing things in a higher way is not all too bad in programming.

The `if not loop.last` statement separates the values of interest with a comma, thus effectively fulfilling the work the `WHERE` clause where it failed. The `age2.sql` shows this in action.

```
{% set years = [1995, 1997, 2002] %}

SELECT birth_year,
{% for year in years %}
SUM(CASE WHEN birth_year = {{ year }} THEN trip_min_round ELSE 0 END) AS trip_min_round_{{ year }
{% endfor %}
SUM(trip_min_round) AS totals_trip_min_round
FROM {{ ref('citi_trips_long') }}
WHERE birth_year IN (
  {% for year in years %}
  {# this will separate the years 1995, 1997 and 2002 with a comma, nothing out of this world #}
    {{ year }}{% if not loop.last %}, {% endif %}
  {% endfor %}
)
GROUP BY birth_year
```

Running the `dbt compile --select jinja` code will compile the `age2.sql` inside the `target` directory. It's contents are as follows. Notice the effect of the `if not loop.last` statement at the end and how it is a replicate of hardcoding `WHERE birth_year IN (1995, 1997, 2002)`.

```
SELECT birth_year,

SUM(CASE WHEN birth_year = 1995 THEN trip_min_round ELSE 0 END) AS trip_min_round_1995,

SUM(CASE WHEN birth_year = 1997 THEN trip_min_round ELSE 0 END) AS trip_min_round_1997,

SUM(CASE WHEN birth_year = 2002 THEN trip_min_round ELSE 0 END) AS trip_min_round_2002,

SUM(trip_min_round) AS totals_trip_min_round
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year IN (


    1995,


    1997,
```

```
    2002

)
GROUP BY birth_year
```

Copy pasting the above compiled SQL into BigQuery you get a cleaner table with all the other birth years left out.

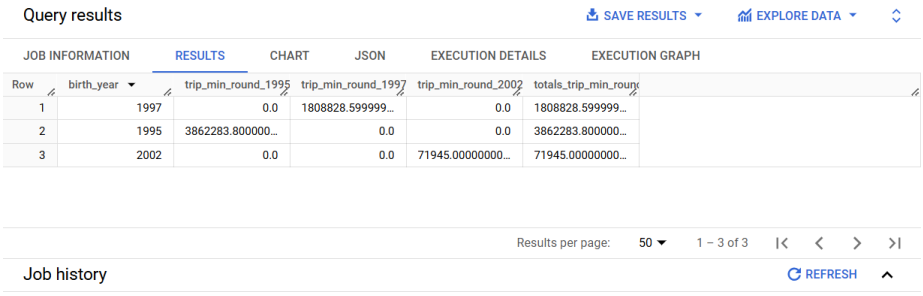


Figure 16.2: Age2 query

16.3 Improvising using DRY Principle

We can go a step further and make our table leaner, by eliminating all the `trip_min_round_<year>` columns and having just one `trip_min_round` summation column for the three years 1995, 1997 and 2002. The `ages3.sql` exemplifies this.

```
{% set years = [1995, 1997, 2002] %}

SELECT birth_year,
SUM(trip_min_round) AS totals_trip_min_round
FROM {{ ref('citi_trips_long') }}
WHERE birth_year IN (
  {% for year in years %}
  {# this will separate the years 1995, 1997 and 2002 with a comma, nothing out of this
    {{ year }}{% if not loop.last %}, {% endif %}
  {% endfor %}
)
GROUP BY birth_year
```

After running `dbt compile --select jinja`, the compiled `age3.sql` in the target directory is as follows:

```
SELECT birth_year,
SUM(trip_min_round) AS totals_trip_min_round
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year IN (

    1995,

    1997,

    2002

)
GROUP BY birth_year
```

For sure you get a leaner table which is far less verbose.

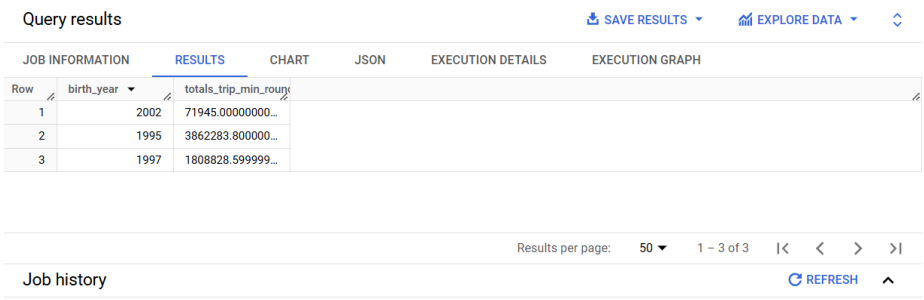


Figure 16.3: Age3 query

One more thing, you can create views from the SQL jinja queries by simply running the trusty `dbt run --select jinja`. This is the resulting output.

```
--snip--
10:37:54  1 of 4 START sql view model nyc_bikes.age ..................................... [RUN]
10:37:59  1 of 4 OK created sql view model nyc_bikes.age ................................ [CREAT
10:37:59  2 of 4 START sql view model nyc_bikes.age2 .................................... [RUN]
10:38:04  2 of 4 OK created sql view model nyc_bikes.age2 ............................... [CREAT
10:38:04  3 of 4 START sql view model nyc_bikes.age3 .................................... [RUN]
10:38:06  3 of 4 OK created sql view model nyc_bikes.age3 ............................... [CREAT
10:38:06  4 of 4 START sql view model nyc_bikes.years ................................... [RUN]
10:38:09  BigQuery adapter: https://console.cloud.google.com/bigquery?project=dbt-project-437116&
10:38:09  4 of 4 ERROR creating sql view model nyc_bikes.years .......................... [ERROR
```

```
10:38:09
10:38:09  Finished running 4 view models in 0 hours 0 minutes and 23.66 seconds (23.66s
10:38:09
10:38:09  Completed with 1 error and 0 warnings:
10:38:09
10:38:09    Database Error in model years (models/jinja/years.sql)
  Syntax error: Expected end of input but got keyword SELECT at [15:1]
  compiled code at target/run/dbt_book/models/jinja/years.sql

--snip--
```

The same views can be called out in BigQuery like the `age3` view shown below.

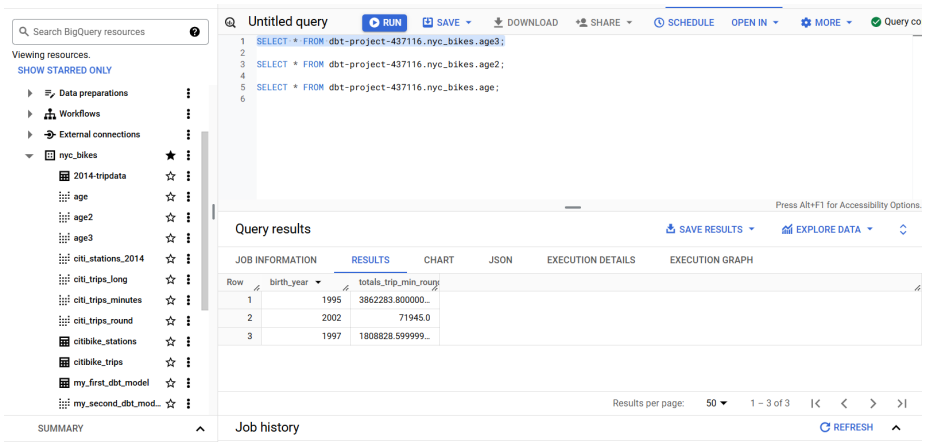


Figure 16.4: Age3 view in bigquery

Now that the `{% if not loop.last %}` and its closing `{% endif %}` statements have ceased being cryptic, we can now sort out why the `years` view could not be created as seen in this `dbt run --select jinja` error.

```
10:38:09  4 of 4 ERROR creating sql view model nyc_bikes.years .....................
```

If you look at the compiled SQL for the `years.sql` in the target directory, there are multiple SELECT statements each for a single year in the `years` variable. All those three SELECT statements cannot be run simoultaneously to produce a single table in BigQuery. But tweaking the `jinja/years.sql` a bit and using the `loop.last` statement will make all the difference. Here is the `jinja/years2.sql`.

```
{% set years = [1995, 1997, 2002] %}
```

```
SELECT *
FROM {{ ref('citi_trips_long') }}

WHERE birth_year IN (
    {% for year in years %}

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

)
```

The `for` loop begins when we want to specify the years, the `if not loop` programmatically adds a comma until the last one and finally the `endfor` statement tells our computer to break out of the looping. Here is the corresponding compiled SQL in the `years2.sql` in the target directory. When this query is pasted, into BigQuery, it only filters the rows matching to the specified birth years.

```
SELECT *
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`

WHERE birth_year IN (


    1995
    ,


    1997
    ,


    2002



)
```

When we run `dbt run --select jinja`, this time round the `years2` view is created.

```
--snip--
11:45:10  5 of 5 START sql view model nyc_bikes.years2 ................................ [RUN]
```

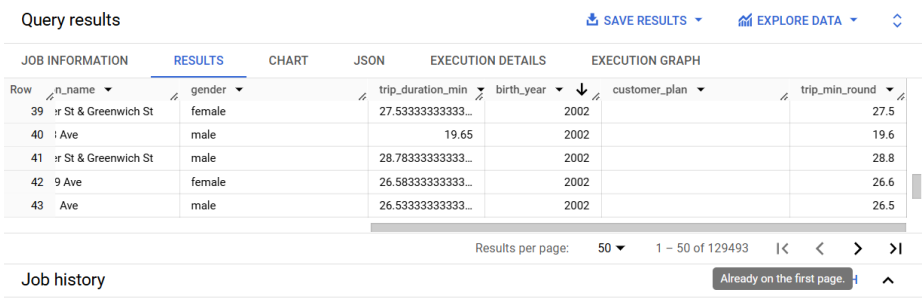


Figure 16.5: Years 2 query

```
11:45:12  5 of 5 OK created sql view model nyc_bikes.years2 ........................
--snip--
```

Chapter 17

Macros

A macro in dbt is a reusable piece of code. That's it. A macro in dbt is what a function is to Python or JavaScript. The building block of a macro is the jinja template. Below is the structure of a dbt macro.

```
{% macro macro_name(arg1, arg2, ..., argN) %}

    SQL logic here, using the parameters as needed.

{% endmacro %}
```

You begin a macro with the name `macro` and end it with `endmacro`. Macros are defined in SQL files and stored inside the already shipped `macros` folder in dbt.

17.1 Invoking a macro

There are three ways to call a macro. They are:

1. Using expression blocks
2. Call blocks
3. Run operation command

17.1.1 Invoking a macro using expression blocks

If the macro does not have any parameters, it can be invoked as a solo object like so:

```
{{ macro_name() }}
```

But if it has parameters, we have to call it with it's entire entourage.

```
{{ macro_name(arg1, arg2, argN) }}
```

17.1.2 Invoke a macro using call blocks

In this method, one can invoke a macro inside another macro. Here is the template.

```
{% call called_macro( arg1, arg2, . . argN) %}

    Code to be accessed by the macro called_macro

{% endcall %}
```

The above code calls a macro called `called_macro` and everything in between the `{ %call% }` and `{% endcall %}` statements can be accessed using the `caller()` method.

Here is an example of a macro.

```
{% macro select_all_columns_macro(table_name) %}

    SELECT *
    FROM {{ table_name }}
    WHERE {{ caller() }}

{% endmacro %}
```

Now, call the macro using call blocks.

```
{% call select_all_columns_macro('EVENT_TABLE') %}

CREATE_DATE >= '2020-02-18'::DATE

{%- endcall %}
```

When it is called it would render:

```
SELECT *
FROM EVENT_TABLE
WHERE CREATE_DATE >= '2020-02-18'::DATE
```

17.1.3 Invoke a macro from the Command Line Interface (CLI)

To run a macro from the CLI or terminal, we use the `dbt run-operation {macro} --args '{args}'{macro}:` command. The macro will run with the arguments provided. The below macro being run from the CLI selects all columns from a table called `my_table`.

```
dbt run-operation select_all_columns --args '{table_name: my_table}'
```

The above are ways to invoke a macro but for simplicity purposes, we shall rely on the first method of invoking macros using expressions.

17.2 Simple macro

Having known that a macro acts like a function, let's create a macro that calculates the age of a bike rider. We already have the `birth_year` column, so calculating one's age should be as simple as subtracting their birth year from the current year.

As earlier mentioned, macros should go into the `macros` folder. Create a `calculate_age` SQL file and inside it paste the following contents.

```
{% macro calculate_age (year) %}

(EXTRACT( YEAR FROM CURRENT_DATE() ) - {{ year }})

{% endmacro %}
```

Remember, a macro is a function and thus what goes within the `{% macro %}` and `{% endmacro %}` expressions is the function itself! This explains why our `calculate_age` macro is so succinct. The `(EXTRACT( YEAR FROM CURRENT_DATE() ) - {{ year }})` is just SQL's way of extracting the current year and subtracting an earlier year being referenced by the variable `{{ year }}`. Of course the specified `{{ year }}` column has to be numeric.

Alright. To see the `calculate_age` macro in action, create a `biker_age` SQL file inside the `jinja` directory. Paste the following content.

```
SELECT *,
{{ calculate_age("birth_year")}} AS AGE
FROM
{{ ref('citi_trips_long') }}
```

In the sub-chapter of Invoking a Macro, we saw that a macro is invoked in the following format: `{{ macro_name(arg1, arg2, argN) }}`. The macro name goes first, followed by the arguments in brackets. We have essentially done this in the `biker_age` SQL file. The `calclate_age()` macro has been provided the column to calculate on, the `birth_year` column. We use the `AS` keyword to create a new column with the alias `AGE`. Thereafter, we run the open sesame command `dbt compile --select macros`.

After compilation, dbt created a `biker_age` SQL file containing the below code:

```
SELECT *,


(EXTRACT( YEAR FROM CURRENT_DATE() ) - birth_year)

 AS AGE
FROM
`dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
```

The above should definitely result in a table with the `AGE` column at the far end.

17.3 Complex macro

The above was a simple macro that neatly drove the point home. How about a more complex macro, like one that works on an entire table, transforms it, has more than one argument and oh my… one in which you can specify the parameters? We want to write our functions on sand, not stone, so that we can change at will. That's the kind of macro we need.

Let's go. Create a SQL called `age_trips` with the following code:

```
{% macro age_trips (column, duration, table_name, years = [1995, 1997, 2002]) %}

SELECT {{ column }},
SUM({{ duration }}) AS totals_trip_min_round
FROM {{ ref( table_name ) }}
WHERE {{ column }} IN (
  {% for year in years %}
  {# this will separate the years 1995, 1997 and 2002 with a comma, nothing out of this
    {{ year }}{% if not loop.last %}, {% endif %}
  {% endfor %}
)
GROUP BY {{ column }}
```

```
ORDER BY {{ column }} DESC

{% endmacro %}
```

Our intelligent mind (no pun intended) created a macro that selects a column, sums the time duration in that column and additionally, aggregates the sum based on certain numerical column values. It will not sum everything in the entire set but just certain values specified by the `years` variable. Additionally, we have preset the values to go into the `years` variables which are `1995`, `1997` and `2002`. Finally, we order the table by arranging the values of the specified columns in descending order.

Now is time to test our macro. Under the `jinja` directory, create a SQL file called `age_trip_totals`. It should have the below minutiae code.

```
{{ age_trips(column='birth_year', duration='trip_min_round',
table_name='citi_trips_long', years = [1990, 1996, 1998, 2001, 2002]) }}
```

What on earth just happened here? There was no `SELECT` statement as in the `biker_age` file? Yes, there wasn't, and for a good reason. We had already specifed our `SELECT` blueprint in the macros file. Therefore, calling the `age_trips` function from within `age_trips_totals` file will invoke the SQL statement encapsulated by the `age_trips` function. If you run `dbt compile --select macros`, the following SQL file will be compiled in the target directory.

```
SELECT birth_year,
SUM(trip_min_round) AS totals_trip_min_round
FROM `dbt-project-437116`.`nyc_bikes`.`citi_trips_long`
WHERE birth_year IN (


    1990,


    1996,


    1998,


    2001,


    2002
```

```
)
GROUP BY birth_year
ORDER BY birth_year DESC
```

Pasting this query on BigQuery will give us a table which aggregates the total trip duration for bikers born in specific years only. That is, those bikers born in any of the following years: 1990, 1996, 1998, 2001 and 2002.

Query results

JOB INFORMATION | RESULTS | CHART

Row	birth_year	totals_trip_min_roun
1	2002	71945.00000000...
2	2001	173807.3
3	1998	1600363.299999...
4	1996	2411691.199999...
5	1990	13084004.19999...

One can also create a view of this macro model using `dbt run --select age_trip_totals`. A `biker_age` view should appear under the `nyc_bikes` dataset.

By doing the above, we not only created a macros with default parameters, but we could also change them and get valid results. We can do this by tweaking the `years` argument in our `age_trips()` function. As a matter of fact, the `years` parameter doesn't have to take *years* per se, it can actually work with any numerical column. But we just specified the name `years` as a clue. In the `age_trip_totals2` SQL file, we specified the ages of interest from within the `AGE` column of our `biker_age` view.

```
{{ age_trips(column='AGE', duration='trip_min_round',
table_name='biker_age', years = [22, 27, 29]) }}
```

Now let's compile this model and see the result:

```
dbt compile --select age_trip_totals2
```

We get the following output in our terminal and by extension, the `age_trip_totals2` SQL file under the `target` directory.

```
19:53:44  Concurrency: 1 threads (target='dev')
19:53:44
19:53:44  Compiled node 'age_trip_totals2' is:


SELECT AGE,
SUM(trip_min_round) AS totals_trip_min_round
FROM `dbt-project-437116`.`nyc_bikes`.`biker_age`
WHERE AGE IN (


    22,


    27,


    29

)
GROUP BY AGE
ORDER BY AGE DESC
```

Pasting the above in BigQuery gives us an aggregation of the total trip duration for people aged 29, 27 and 22.

Query results

JOB INFORMATION | RESULTS | CHART | JSON | EXECUTION DETAILS | EXECUTION GRAPH

Row	AGE	totals_trip_min_round
1	29	3862283.8000000012
2	27	1808828.5999999992
3	22	71945.000000000015

Figure 17.1: Age trip total 2 view

As mentioned earlier, and as show with a quick example of `age_trip_totals` model, we can create views of each of our macro models. Since they are all within the jinja directory, the following does the trick: `dbt run --select jinja`. This should create a view of each of our models created in this chapter.

Below is a snippet of the creation of views.

```
-- snip --
```

```
20:15:05  8 of 8 START sql view model nyc_bikes.age_trip_totals2 .....................
20:15:07  8 of 8 OK created sql view model nyc_bikes.age_trip_totals2 ................
-- snip --
```

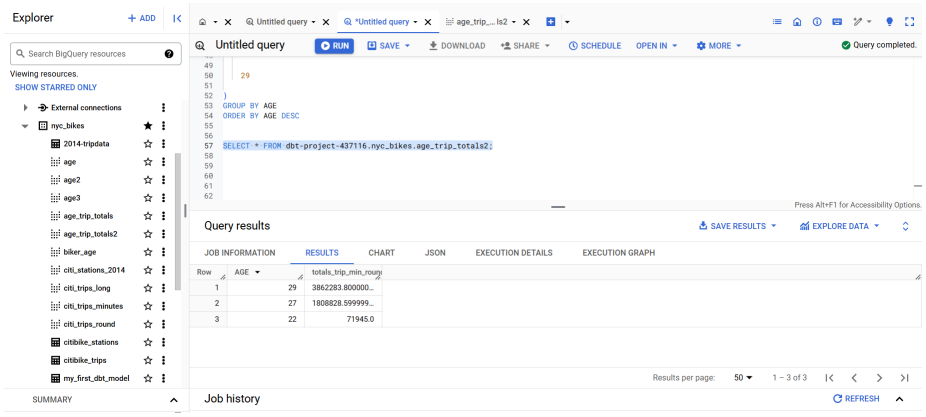


Figure 17.2: All macro reliant models

Chapter 18

Hooks

At one time, I posed a question to a well-versed data engineer of why the technological space seems to be awash with exotic outlandish names. Hooks in dbt seem to fit into the calibre of these outlandish names… for there is nothing regarding its purpose in dbt which seems to suggest it will lure and capture your data.

You could take hooks as customized SQL models that are out of the box when it comes to dbt. There are various forms of hooks, namely:

1. `pre-hook` - executed before a model, snapshot or seed is built.
2. `post-hook` - executed after a model, snapshot or seed is built.
3. `on-run-start` - executed at the start of implementing the following code executions: `dbt build`, `dbt compile`, `dbt docs generate`, `dbt run`, `dbt seed`, `dbt snapshot` or `dbt test`.
4. `on-run-end` - executed at the end of the following code executions: `dbt build`, `dbt compile`, `dbt docs generate`, `dbt run`, `dbt seed`, `dbt snapshot` or `dbt test`.

A confession to make: since I have minimal experience using hooks, I shall play it safe, thus the reason why this chapter is quite short.

18.1 Post-hooks

The format of writing a hook is:

```
{{ config(
  post_hook=[
    "<Place your SQl query here>"
  ]
) }}

SELECT * FROM raw_table
```

We tried creating a simple post-hook but no matter what we tried, `dbt` kept throwing back errors. Nevertheless, for more on how to set up hooks, see here and here.

Chapter 19

Hosting dbt generated documentation

Imagine you have put blood, sweat and tears into your book, you have said everything you wanted to say, divulged what was considered secret, and unearthed what was incomprehensible. Except for one thing: you can't publish it. That would be a disaster, a mockery of your efforts, yet that would be our portion if we had not published the dbt documentation we had created in Chapter 7. After all is said and done, it has to be printed somewhere, and obviously the world wide web is our playground.

There are various ways to publish your dbt generated documentation, such as in Github, Google Cloud and Azure Devops. We would have loved to host our dbt generated documentation in Google cloud as that would have put us in the league of astute developers, but the process and costs were a bit too much. Therefore, we slid back to Github which is totally free.

19.1 Preparations

19.1.1 Creating a `gh-pages` branch

If hosting dbt documentation were easy, then we would have simply done so from the `main` branch. However, taking into account the process of hosting on a more sophisticated platform such as Google Cloud, the entire process seems like a thing for the top-tier tech gurus. It is for this reason we create a separate branch by the name of `gh-pages`. Github can autodetect the `index.html` file under this branch automatically compared to a branch by any other name.

To create a new branch run `git checkout --orphan gh-pages`.

The purpose of the `--orphan` command is to create a branch from a clean slate; it has no connection to previous commits.

19.1.2 Tracking the `target` folder

Now, in order to create the dependency files for your dbt documentation within the `target` folder, run `dbt docs generate`.

Once the `catalog.json` is rewritten, we must add this branch for tracking. By default, in the `.gitignore` file, all files under the `target` folder are set to not be tracked. Therefore, when adding this folder, we introduce the `-f` keyword when preparing our files for commit like so:

```
git add -f target
```

Now let's commit our `target` folder contents within our local `gh-pages` branch.

```
git commit -m 'hosting dbt docs in github' target
```

Our terminal did get a huge list of outputs!

19.1.3 Pushing to Github

Next, we only want to take the contents of our `target` folder and push them into the `gh_pages` branch. We use the following code: `git subtree push --prefix target origin gh-pages`. A subtree is like a sub-folder in Github. We use `subtree` when we want to save certain directories from one repository into another. In our case we only want the contents of the `target` folder and this we specified using the `--prefix` keyword. Sometimes, one could have been saving their work at a higher folder level, such as yours truly. What do we mean here? We mean that perhaps you have been commiting from `folder1/myfiles/commit-here` but you work was initialized by git from within `folder1`. In this case just push your files from `folder1` or else you will get the below error when performing a subtree push:

```
You need to run this command from the toplevel of the working tree.
```

In this case, just specify the top-level folder and the target folder, separated with slash(es) like so:

```
git subtree push --prefix dbt_book/target origin gh-pages
```

Once you receive a message that the push operation was successful, you can checkout to the main branch like so: `git checkout main` or for any other branch for that matter.

19.2 Hosting on Github

If you go to your specific repository on Github, such as `dbt_book_codes` in our case, you can use the dropdown right under the repository name to move into a different branch.

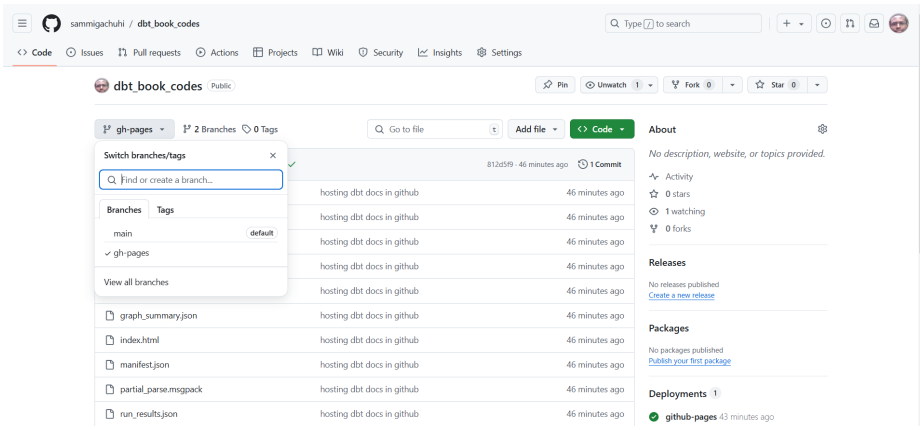


Figure 19.1: gh-pages branch

Once you are within the `gh-pages` branch, go to the **Settings** tab, and click on the **Pages** menu. You will find that Github already auto-detected the `index.html` file and proceeded to create a hosted webpage going by the name of your repository.

Click on the link to go to your hosted dbt documentation. Access the dbt generated documentation for this course from here.

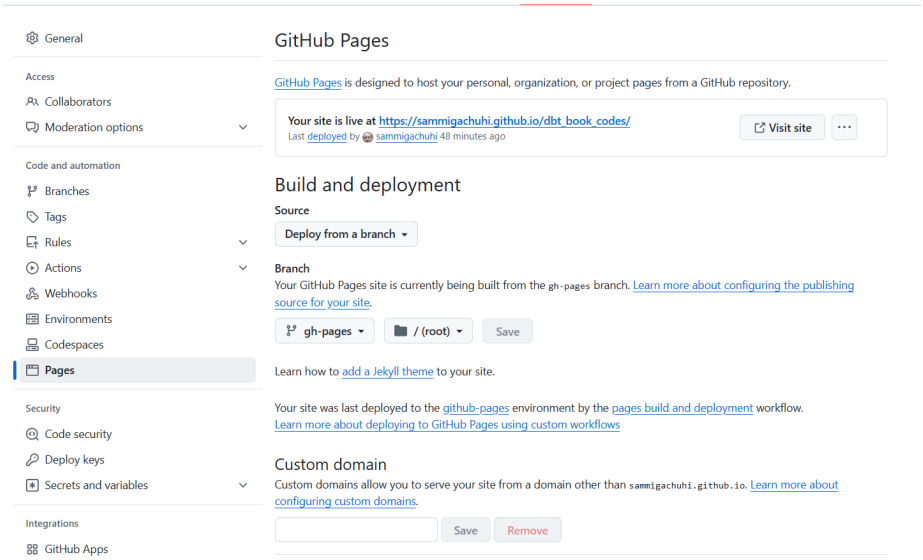


Figure 19.2: Hosting gh-pages

Chapter 20

Conclusion

Our journey of learning data build tool (dbt) has taken us from how to set it up, creating models, setting up seeds, taking snapshots of data, and using jinja and macros to create custom functions. Finally, as the final nail on the coffin, we saw how to set up an externally hosted dbt website on Github. Except for insufficient knowledge on hooks and how to host a dbt website on Google cloud, it is our wish that this book will enable you to come to speed with using dbt in your data. With the growth of using data warehouses and exploiting big data already happening in various institutions, we are sure dbt will always be one of the key tools for data handling.

FINIS