**Quickstart: Build a Modern Data Stack with**

**dbt and Databricks V1.2**

Hands-on Workshop

| **Summary** | Quickstart: Build a modern data stack with dbt and Databricks |
| --- | --- |
| **ID** | Build-a-modern-data-stack-with-dbt-and-Databricks |
| **URL** | [Build-a-modern-data-stack-with-dbt-and-Databricks](https://pages.databricks.com/202207-AMER-VE-BuildaModernDataStackWithdbtandDatabricks_01-Registration-LP.html) |
| **Category** | Web |
| **Status** | Complete |
| **Feedback Link** |  |
| **Author** | Pradeep Anandapu, Fei Lang, Bobby Birstock |
| **Version** | 1.2 (09/2022) |
| **Change log** | * Added Beta UI Note * Added “Read Picker” steps; * Change “SQL endpoint” to “SQL warehouse” following Databricks official documentation. |

# Overview

To build an effective data organization, it helps to have the right data platform and the right framework to scale with your organization’s data and AI needs.

The [Databricks](https://databricks.com/product/data-lakehouse) Lakehouse Platform combines the best elements of data lakes and data warehouses to deliver the reliability, strong governance and performance of data warehouses with the openness, flexibility and machine learning support of data lakes.

[dbt](https://www.getdbt.com/) is a transformation workflow that lets teams quickly and collaboratively deploy analytics code following software engineering best practices like modularity, portability, CI/CD, and documentation. Now anyone who knows SQL can build production-grade data pipelines.

A modern data stack built on the lakehouse with Databricks and dbt greatly simplifies your data engineering to bring scale and performance to your data platform.

In this Quickstart, you will follow a step-by-step guide to using dbt Cloud with Databricks. You will build a scalable transformation pipeline from scratch.

Let's get started.

## What you will need during the lab

* A Databricks account with **ADMIN** access to the workspace:
  + For workshop participants, please follow the instructions provided by the instructors to get access to the lab workspace. Then move to the next section.
  + For post-workshop participants who do not have a Databricks account, please complete the two steps below following the ***Get Started with Databricks Guide*** on [AWS](https://docs.databricks.com/getting-started/index.html), or [Azure](https://docs.microsoft.com/en-us/azure/databricks/) before you can move to the next section:
    - Sign up for a Databricks free trial account.
    - Set up your workspace.
* Access to Databricks Partner Connect:
  + For workshop participants, the lab workspace already has access to Partner Connect.
  + For post-workshop participants, make sure your Databricks account, workspace, and the signed-in user all meet the requirements for Partner Connect on [AWS](https://docs.databricks.com/integrations/partner-connect/index.html#requirements) or [Azure](https://docs.microsoft.com/en-us/azure/databricks/integrations/partner-connect/#requirements).

## What you will learn

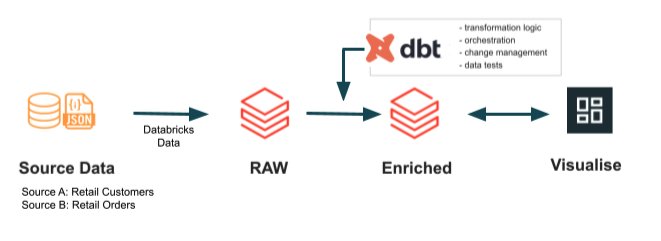
* How to build scalable data transformation pipelines using dbt & Databricks.
* How to leverage dbt transformed data in Databricks SQL to quickly find insights with the built-in SQL editor, visualizations and dashboards.

## What you will build

* A set of data analytics pipelines for Retail Organization data leveraging dbt and Databricks, making use of best practices like data quality tests and code promotion between environments
* A simple dashboard in Databricks SQL

# Architecture and use case overview

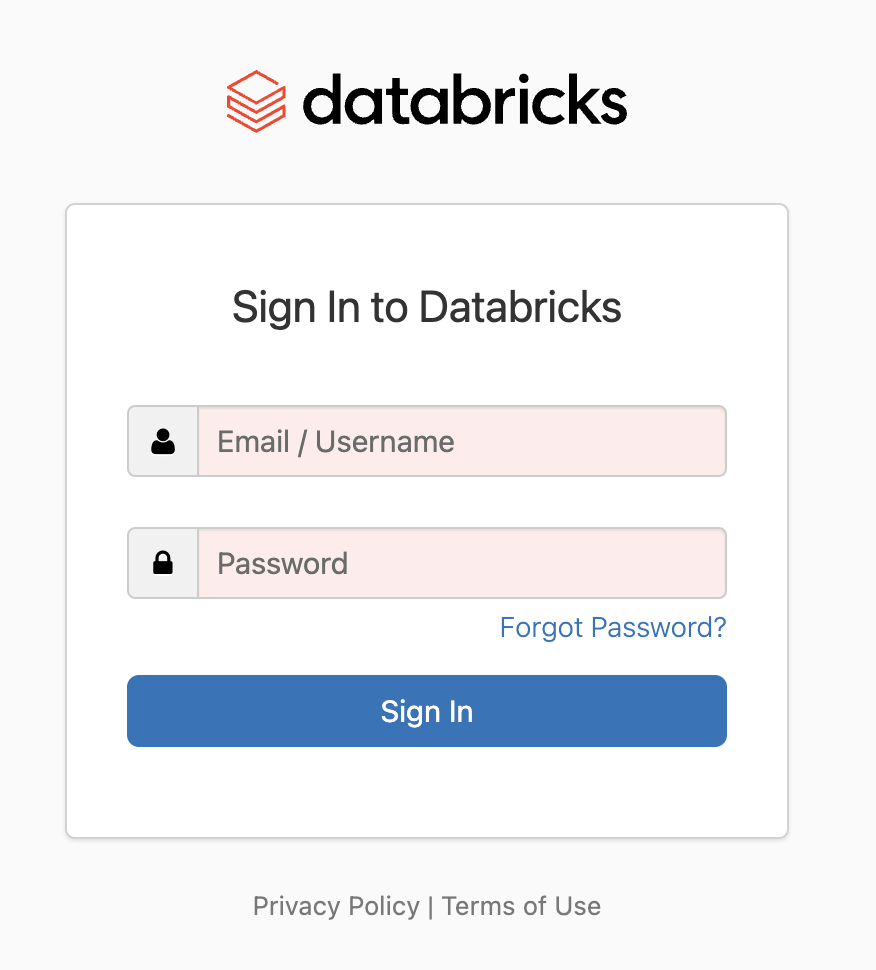
This workshop will walk you through how to take raw retail data sources from a Databricks dataset on [AWS](https://docs.databricks.com/data/databricks-datasets.html) or [Azure](https://docs.microsoft.com/en-us/azure/databricks/data/databricks-datasets) to create a cleaned final model of customer order data that you’ll be able to visualize. To accomplish this, we’re going to build a series of models that progressively clean and mold the data to our assumptions. We’ll also add documentation and testing at the end to make sure that our final transformation is well described and that it meets our assumptions.



# 

# Databricks Configuration

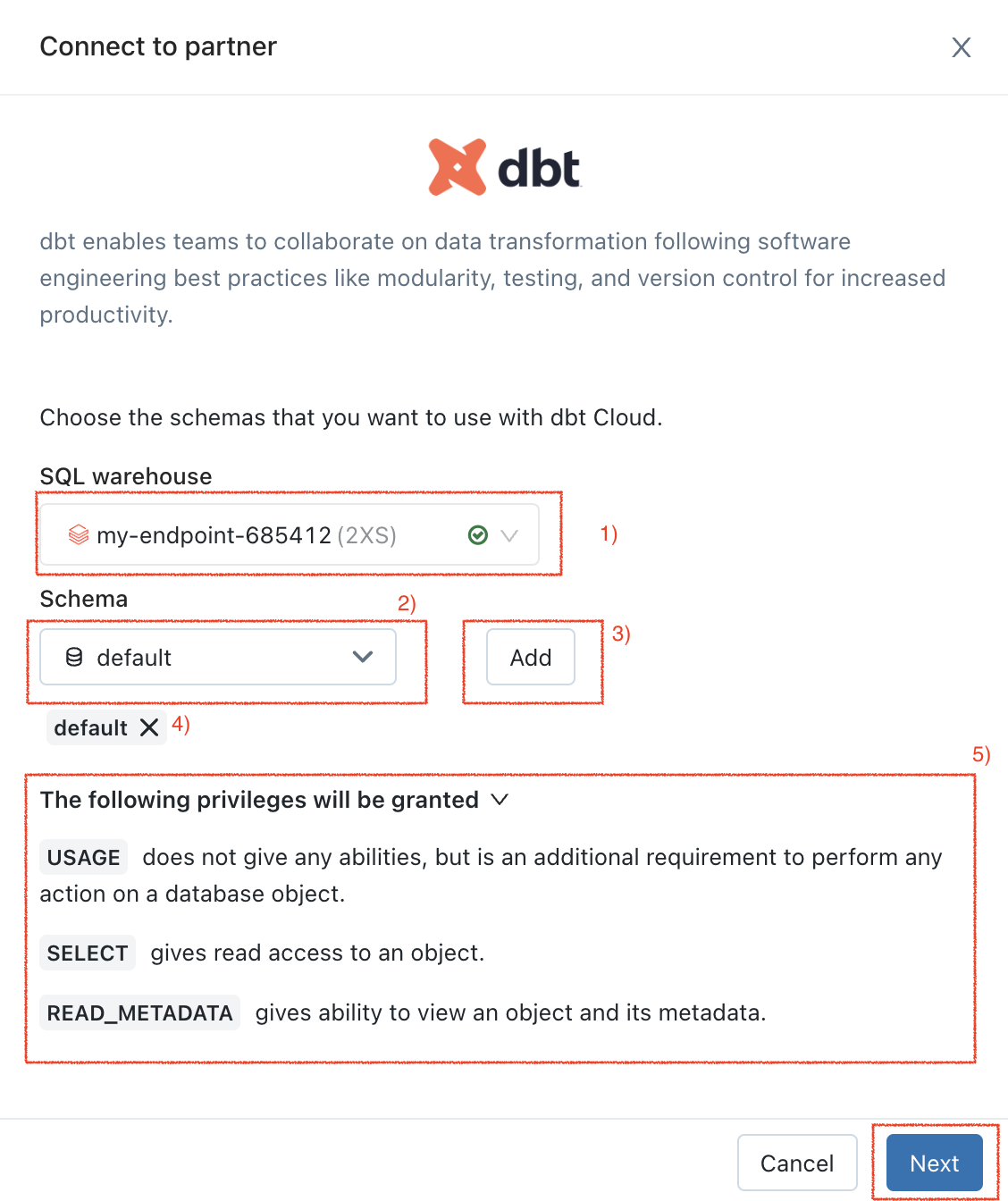
1. Login to your Databricks trial workspace.



1. Databricks workspace UI tour. Instructors will walk through this live during the workshop. For post-workshop participants, click here for a quick tour of the workspace UI on [AWS](https://docs.databricks.com/workspace/index.html) or [Azure](https://docs.microsoft.com/en-us/azure/databricks/workspace/).
2. In the Databricks workspace, click Partner Connect button **Partner Connect** in the sidebar on the left. It will take you to the Databricks Partner UI.

# dbt Configuration

1. In Partner Connect, find the dbt Tile and click on it.
2. You should now see a popup which says **Connect to partner** for dbt. You will be asked to choose the schemas that you want to use with dbt Cloud. This step is to grant read access to the data and metadata of the selected schemas for our dbt project later.



Complete the following steps on this popup:

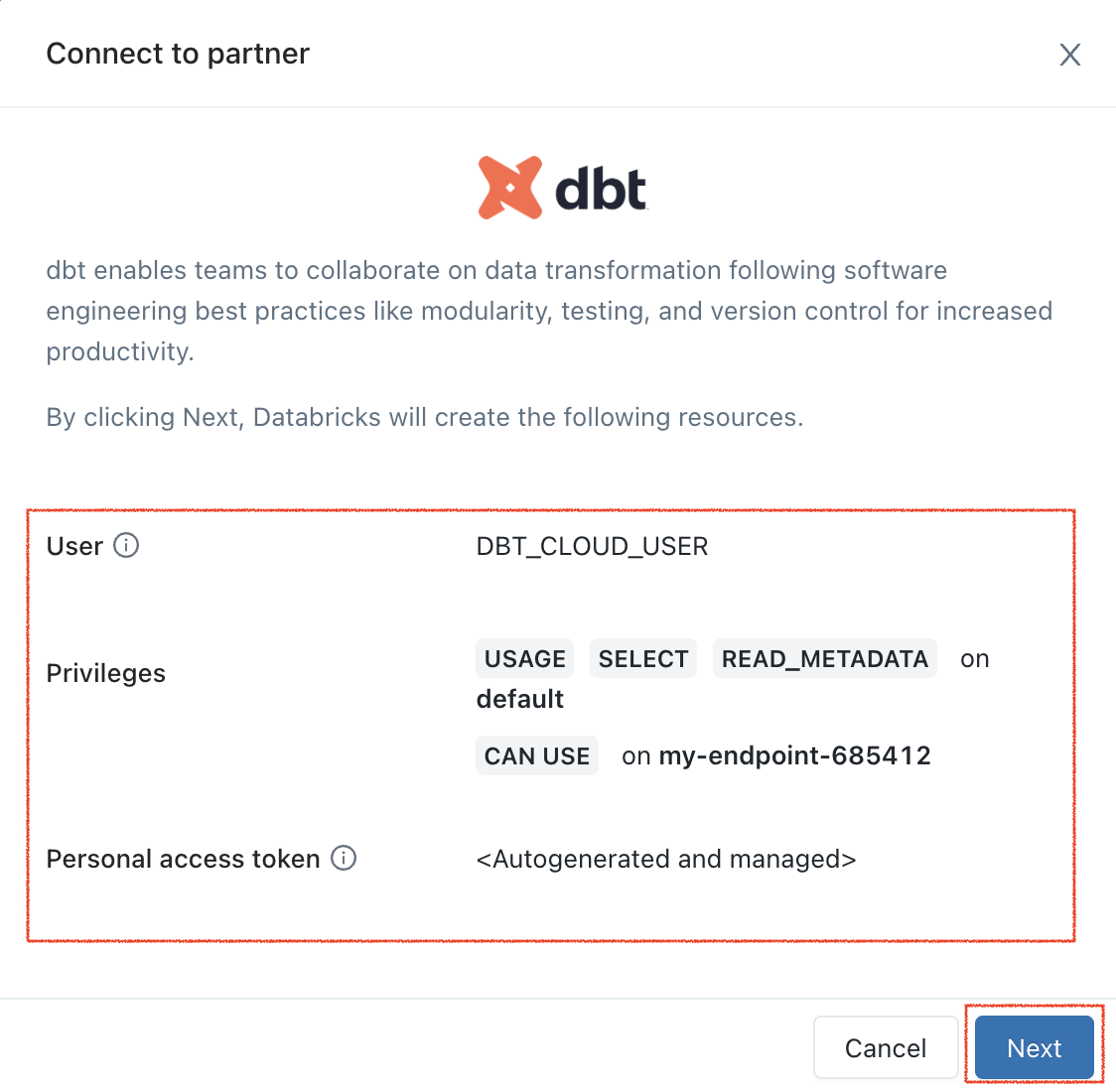
1. Select a SQL warehouse from the **SQL warehouse** dropdown list and start it. Wait until it’s running (when you see a green checkmark next to it);
2. Select a schema (database) from the **Schema** dropdown list. You can select multiple schemas here if needed.

\*If you are a workshop participant using the lab environment for the first time, you should only see one schema ***‘default’***. Select it.

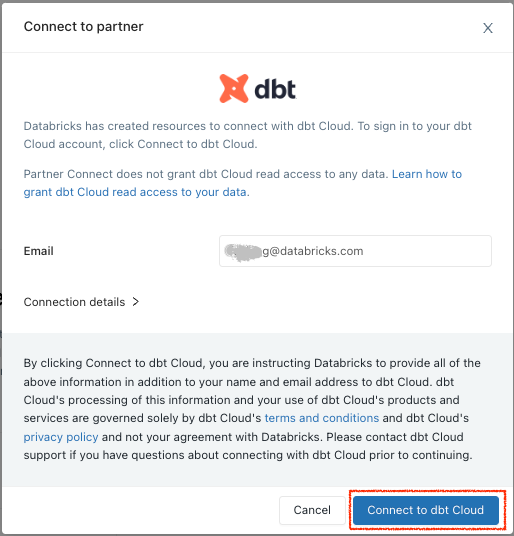
1. Click **Add** button;
2. You can see all the selected schemas listed here;
3. ***USAGE***, ***SELECT*** and ***READ\_METADATA*** privileges will be granted to the select schema(s).

Click on **Next**.

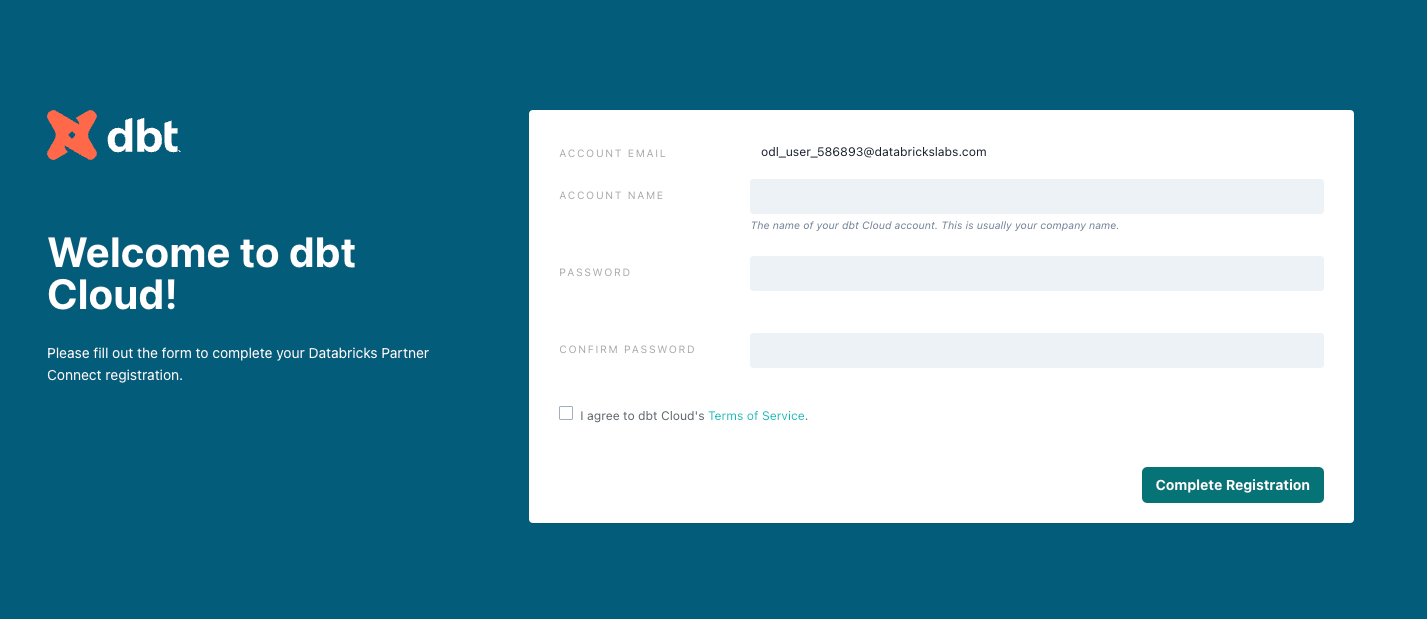
1. Click on **Next** on the next popup. This will create a dedicated ***dbt user,*** ***personal access token*** with the granted privileges to the select schema(s) and a ***Databricks SQL warehouse*** for your dbt Cloud trial for our lab later.



1. On the next popup, it prepopulates the email address tied with the workspace. This email address will be used for the signup for the dbt Cloud trial later. Click on **Connect to dbt Cloud**.



1. You should be redirected to a dbt Cloud signup page. Fill out the form. Make sure to save the password somewhere for login in the future.

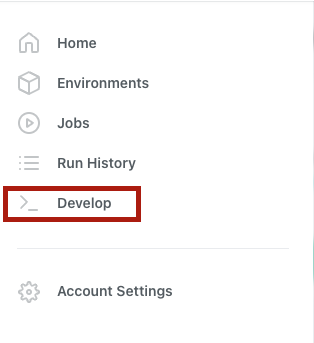
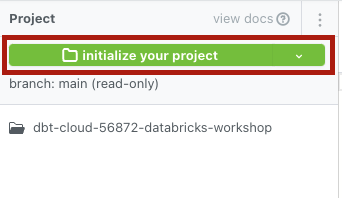
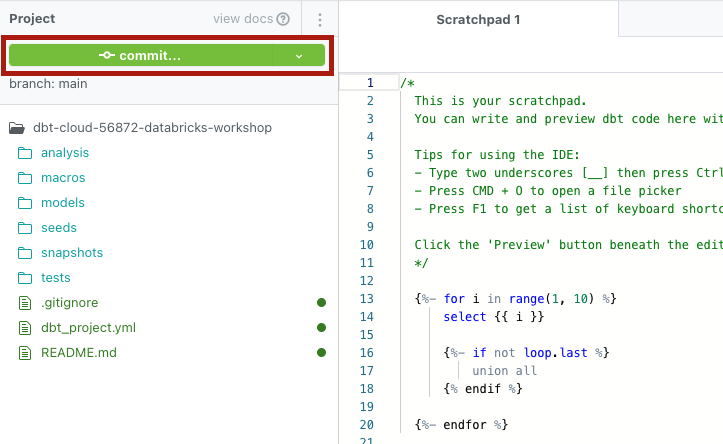
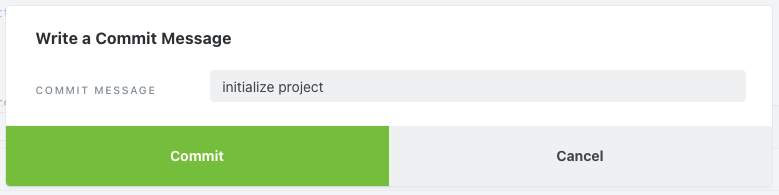
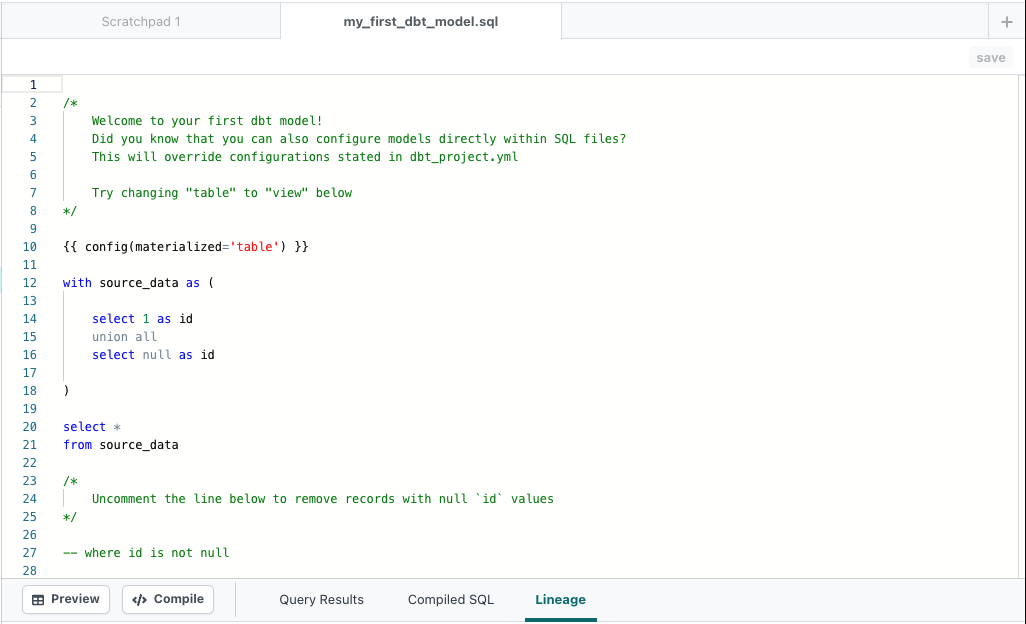
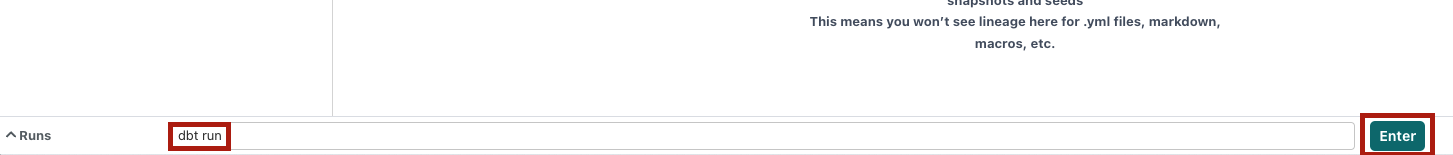
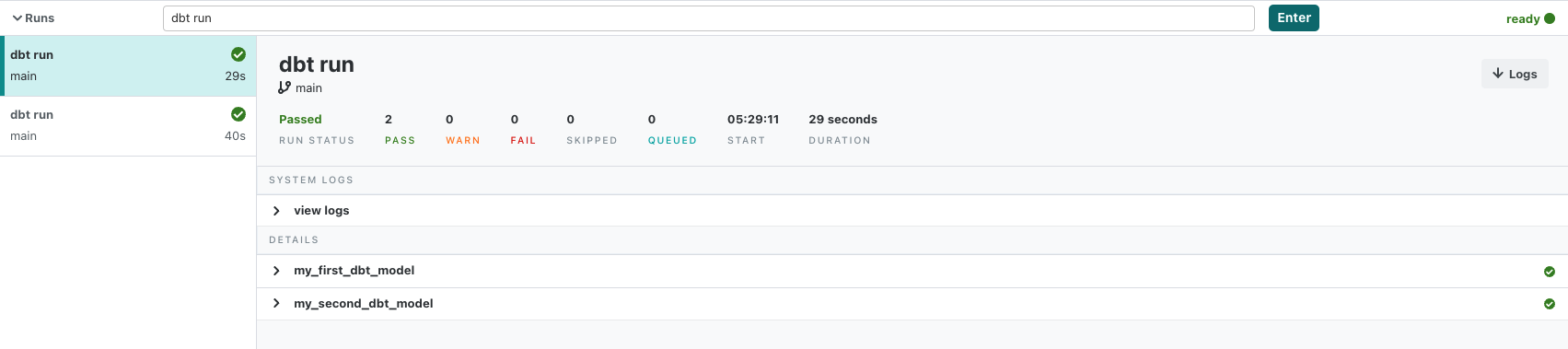


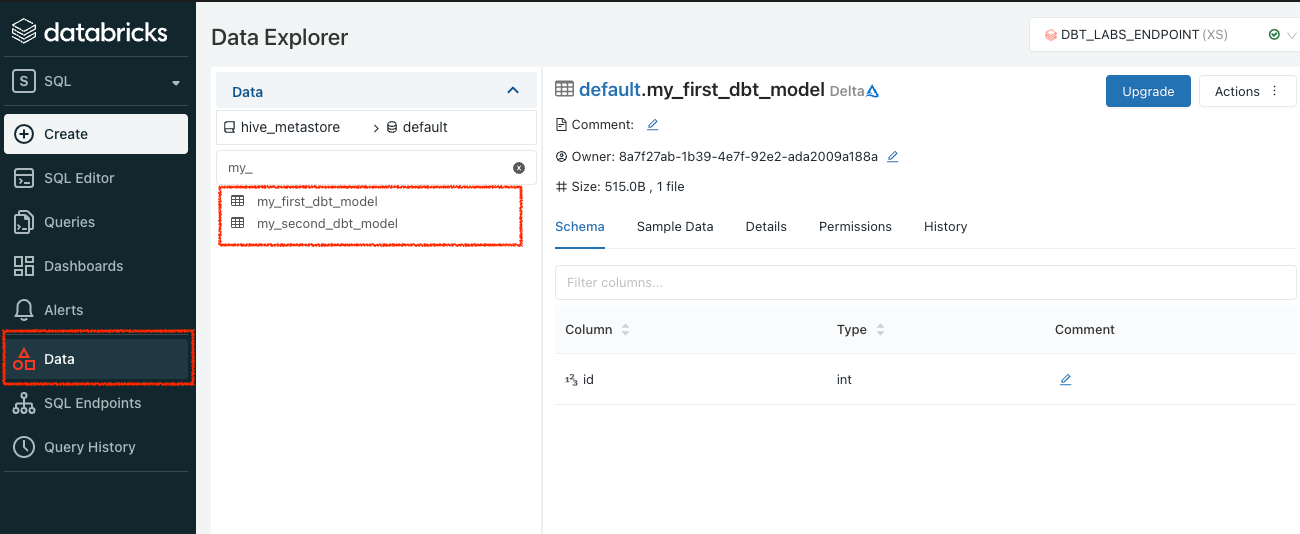
1. Click on Compete Registration. You should now be redirected to your dbt Cloud account, complete with a connection to your Databricks account, a deployment and a development environment, and even a sample job.

To help you version control your dbt project, we have connected it to a [managed repository](https://docs.getdbt.com/docs/dbt-cloud/cloud-configuring-dbt-cloud/cloud-using-a-managed-repository), which means that dbt Labs will be hosting your repository for you. This will give you access to a git workflow without you having to create and host the repository yourself. You will not need to know git for this workshop; dbt Cloud will help guide you through the workflow. In the future, when you're developing your own project, feel free to use [your own repository](https://docs.getdbt.com/docs/dbt-cloud/cloud-configuring-dbt-cloud/cloud-installing-the-github-application). This will allow you to play with features like [Slim CI](https://docs.getdbt.com/docs/dbt-cloud/using-dbt-cloud/cloud-enabling-continuous-integration-with-github) builds after this workshop.

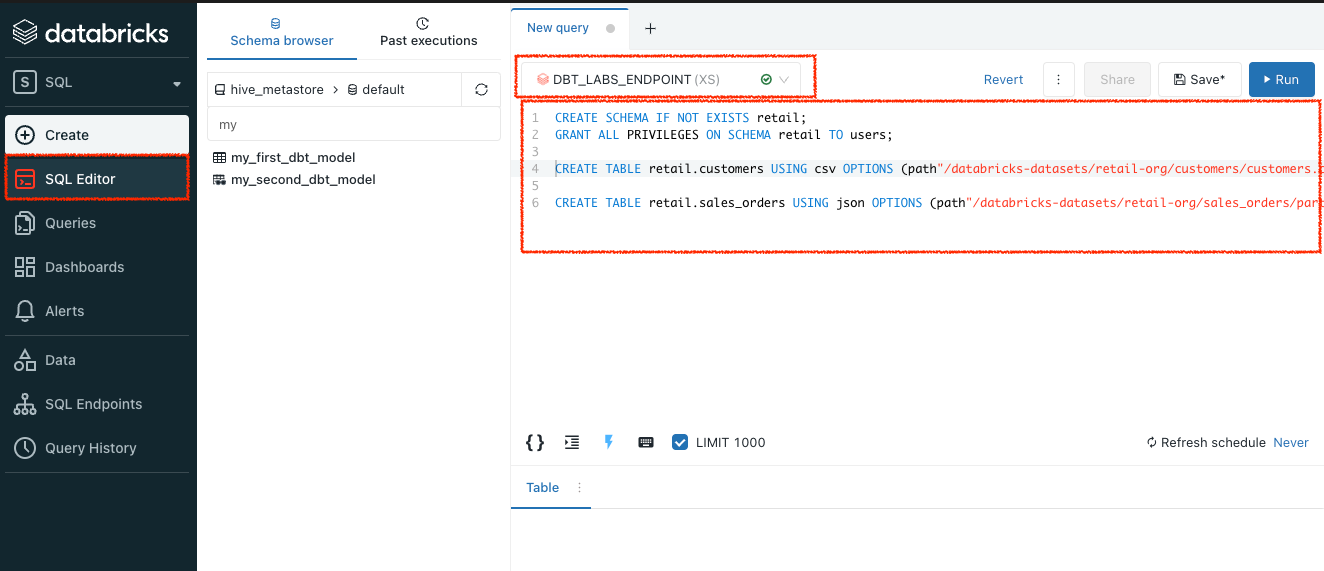
# dbt Project Configuration

*Note: If you land in dbt Cloud’s* ***Beta UI****, please switch to* ***Classic UI*** *in the drop down at the top right. Click on “****Go back to Classic UI****”.* ***Ignore any prompts that ask you to switch to Beta UI****.*

1. Before we start developing let’s do a quick walkthrough of our dbt project. Click on the hamburger menu on the top left side and click on **Develop**. This will open your IDE (Integrated Development Environment) where you will be building your dbt Project.  
   
2. When the IDE is finished loading, click the green **initialize your project** button in the upper left hand corner of the screen. This is the signal to create the dbt starter project with all of the core folders and files that you’ll need.   
   
3. After the initialization completes you should see a number of new files and folders, including everything that you’ll need to develop in your dbt project. As we move through the workshop, we’ll be sure to touch on these key pieces and work on them to fit the goals of our project.
4. Next click **commit…** in the upper left hand corner to make your first commit on the project with all of the new files and folders from the initialization. You always want to have a descriptive commit message anytime you save work, so our message can be something like **initialize project**.   
     
     
     
     
   [Committing](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) your work here will save it to the managed Git repository that was created during the Partner Connect signup. You’ll notice that this commit was made directly on our **main** branch and as the initial commit it will be the only commit directly to the main branch. From here on out we’ll be doing all of our work on a development branch to maintain separation between our production code and development code.
5. There’s a couple key pieces to point out about the dbt IDE before doing any work in it. It is a text editor, a SQL runner, and a CLI with git version control all in one place. You can edit your SQL files, preview the results with the SQL runner, build models at the command line in your Databricks workspace, and save all of your work with a git workflow when you’re done.
6. Alright, let’s run our first dbt models! The initial file structure includes a couple of example models that we’ll be able to run to demonstrate how dbt works and make sure everything was initialized correctly.   
     
     
   Type in **dbt run** into the command line at the bottom of the screen and hit **Enter**.   
     
     
   When the run bar expands you’ll be able to see the results of the run and they should look like this, confirming a successful first run!   
   
7. To confirm the example models created on the Databricks side, we’ll switch back to our Databricks SQL UI. Click the **Data** icon on the left sidebar which opens the **Data Explorer** UI. We can see both models run on the dbt side showing up here.



1. Now as we are here on Databricks SQL, let’s create the data sources that we will be using for the dbt project later. Click the **SQL Editor** icon on the left sidebar to navigate to the SQL Editor UI which opens a **New query** tab for us to create a new SQL query using the running SQL warehouse which was auto-provisioned by Partner Connect.



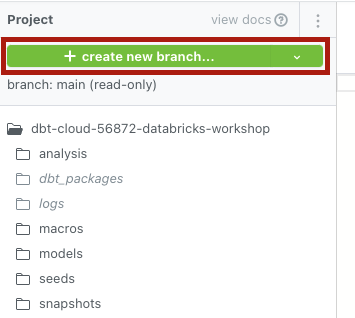
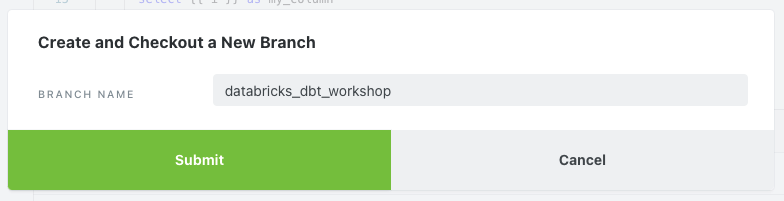
Copy and paste the following 4 SQL statements into the query editor, and run each of them.

| CREATE SCHEMA IF NOT EXISTS retail;  GRANT ALL PRIVILEGES ON SCHEMA retail TO users;  CREATE TABLE retail.customers USING csv OPTIONS (path"/databricks-datasets/retail-org/customers/customers.csv", header "true") ;  CREATE TABLE retail.sales\_orders USING json OPTIONS (path"/databricks-datasets/retail-org/sales\_orders/part-00000-tid-1771549084454148016-e2275afd-a5bb-40ed-b044-1774c0fdab2b-105592-1-c000.json", header "true") ; |
| --- |

After all the statements pass, click ‘**+**’ on the top to open another empty **New query** tab. Copy, paste and run the 2 following SQL Select statements to make sure you can see both tables’ data.

| SELECT \* FROM retail.customers; SELECT \* FROM retail.sales\_orders; |
| --- |

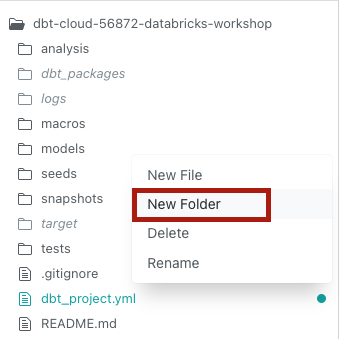
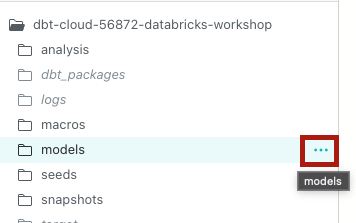
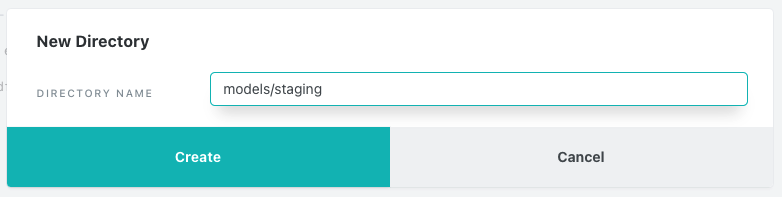
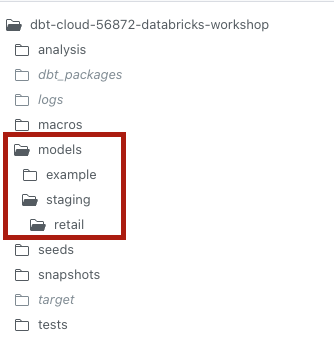
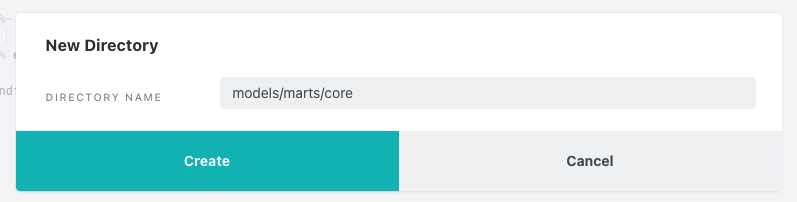
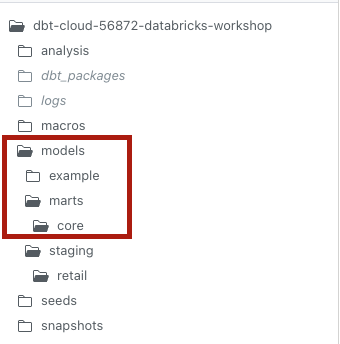
# Structure Setup

1. Alright, let’s get started with creating our dbt project. Before we start building we’ll need to create a new git branch for our work. Click on the green **create new branch** button in the upper left hand corner, call it something like **databricks\_dbt\_workshop,** and then click submit.   
     
     
   
2. The first thing we’ll do is update the **dbt\_project.yml** file, which is the file dbt looks for to recognize that the file directory we’re working in is a dbt project. Find the file in your file tree and click on it to open it up. You can replace the existing contents of the file by copying from below, pasting into the file, and then saving the file using the **save** button in the upper right hand corner of the screen:

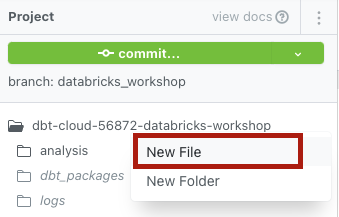
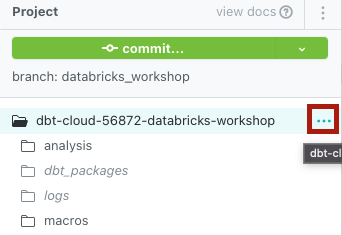
| name: 'databricks\_workshop' version: '1.0.0' config-version: 2  profile: 'default'  source-paths: ["models"] analysis-paths: ["analysis"] test-paths: ["tests"] seed-paths: ["seeds"] macro-paths: ["macros"] snapshot-paths: ["snapshots"]  target-path: "target"  clean-targets:   - "target"  - "dbt\_modules"  dispatch:  - macro\_namespace: dbt\_utils  search\_order: ['spark\_utils', 'dbt\_utils']  models:  databricks\_workshop:  example:  materialized: view  staging:  materialized: view  marts:  materialized: table |
| --- |

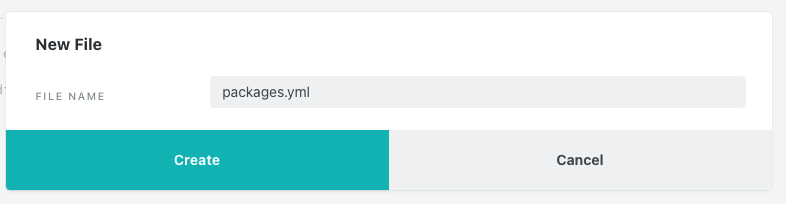


The important configurations to point out in the file with relation to the work we’re doing are in the models section. The gist of it is that we are defining configurations at the folder level that we want applied to all of the models within that folder. Here we’re demonstrating the **materialized** config, which tells dbt how to materialize models when compiling the code before it pushes it down to Databricks, and we’re telling dbt to materialize all of the models in the example and staging folders as views and all of the models in the marts folders as tables. It should be noted that configs such as this can be overridden at the model level if there’s ever a difference in how you’d like to configure a specific model.

1. Next we’ll set the stage for our development, so let’s create a staging directory within the models directory for our staging models. Hover over **models**, click on the 3 dots to the right of the name, then click **new folder**. Add **staging** to the end of the directory name in the popup window and click create. You should now see a new staging folder nested within the models directory.   
     
     
   
2. Within the staging folder, let’s follow the same pattern as above and create another folder called **retail** for our retail data source. Your folder structure should now look like this:  
   
3. We’ll also create one more folder (actually two folders at once!) where we’ll eventually place our transformed models. To do this, click the three dots when hovering over **models**, click **new folder**, and make sure the file path looks like this: **models/marts/core**. The end result here is that we’re creating a **marts** folder within **models** and then a **core** folder within **marts**.  
     
     
   
4. Let’s add some packages to our project! dbt packages are additional dbt projects that you can bring into your own project and use the code as if it was your own. We’ll be demonstrating how to use existing tests in the **dbt\_utils** package to up your testing game. We’ll also be installing **spark\_utils** alongside **dbt\_utils**, which contains spark compatible versions of the macros of the **dbt\_utils** package. To install the packages, first create a new file within your home directory (same level as your **dbt\_project.yml** file) and call it **packages.yml**. Then copy and paste the following code into it and click save:

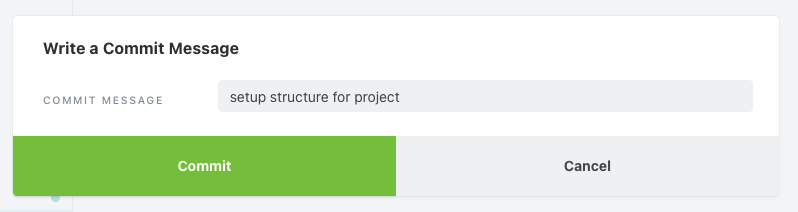
| packages:  - package: dbt-labs/dbt\_utils  version: 0.8.2  - package: dbt-labs/spark\_utils  version: 0.3.0 |
| --- |



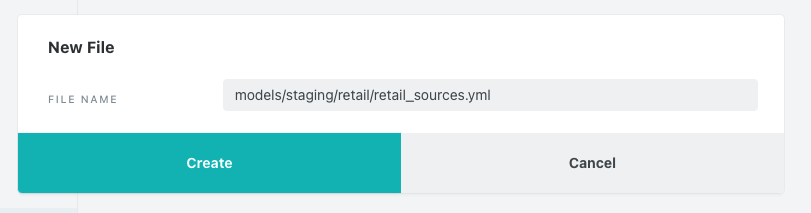


The last step to install the package is to run **dbt deps**, which tells dbt to install all of the packages defined in your **packages.yml** file. Enter in **dbt deps** into the command line, click **Enter**, and you should see a success message there when it completes. Your packages are installed!  
  
  
  
A quick note on the packages we’re installing: if you take a look back at the **dbt\_project.yml** file, you’ll notice there’s a config there called **dispatch**. This configuration tells dbt that whenever **dbt\_utils** is referenced, dbt should look in the **spark\_utils** package for a corresponding function first, and if it doesn’t find it there then it will look in **dbt\_utils**. This is important for maintaining compatibility with Spark SQL across this particular package.

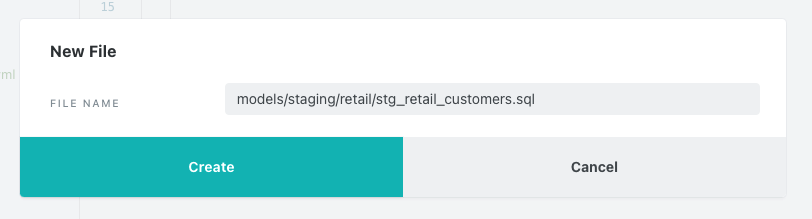
1. That does it for our initial setup, but before we move on let’s commit our work! Click the green **commit** button, write in a descriptive commit message like **setup structure for project** and click commit.



# Sources and Staging

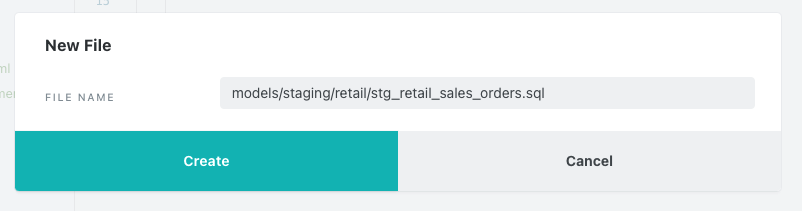
1. Let’s create a source file for the retail data we’ll be transforming. Sources in dbt are what allow you to name and describe raw data in your warehouse. When we declare sources we help define data lineage within our project, which we’ll check out in a bit when we’ve got some models created. Within the **models/staging/retail** folder create a new file called **retail\_sources.yml** and paste the following code into the file:   
     
   

| version: 2  sources:  - name: retail  schema: retail  tables:  - name: customers  - name: sales\_orders |
| --- |

1. The two table names are the names of the two tables we already created in the Databricks UI, a table with customer data and a table with corresponding orders data from retail sales.
2. Next we’ll set up staging models to clean up the raw data sources that we’re pulling in. Typically staging models have a one to one relationship with their corresponding source table and our **customers** data source fits that scenario. On the other hand, our **sales\_orders** source is pretty messy and contains foundational information at two granular levels (row level is at the order granularity while there’s a JSON column with order items embedded within it). We will want both to build the foundational logic for our project. In order to do this and stick with dbt best practices we’ll create a staging model for orders that will handle some deduping and then another staging model that will bring in the deduped data and transform it to order items. By handling this in two separate staging models we’ll ensure clean data at both levels of granularity while building a foundation for more complex transformations downstream.
3. The first staging model is going to be for the customers table. Create a new file in the **models/staging/retail** folder called **stg\_retail\_customers.sql** and paste the following select statement into it before saving the file:   
   

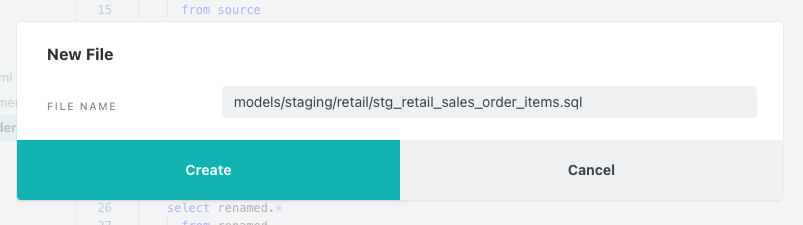
| with source as (  select \*   from {{ source('retail', 'customers') }} )  , renamed as (  select customer\_id as customer\_id  , cast(tax\_id as int) as tax\_id  , tax\_code as tax\_code  , customer\_name as customer\_name  , state as state  , city as city  , case when postcode like '%-%'  then cast(left(postcode,5) as int)  else cast(postcode as int)  end as postcode  , street as street  , case when number like '%.%'  then cast(number as int)  else number  end as number   , unit as unit  , region as region  , district as district  , cast(lon as double) as longitude  , cast(lat as double) as latitude  , ship\_to\_address as ship\_to\_address  , from\_unixtime(valid\_from,'yyyy-MM-dd') as valid\_from\_date  , from\_unixtime(valid\_to,'yyyy-MM-dd') as valid\_to\_date  , cast(units\_purchased as int) as units\_purchased  , loyalty\_segment as loyalty\_segment  from source  )  , de\_duped as (  select \*  from renamed  where valid\_to\_date is null )   select \*   from de\_duped |
| --- |

Make sure you save the file when you’re done copying and pasting!

1. There’s a couple ways to check your query in the dbt IDE before pushing the code down to your warehouse. The first is by clicking on the **Preview** button to see the results of the query in the IDE. You can also click on the **Compile** button, which will show you the compiled code that will be executed when the query is run against your warehouse.
2. To start building the second staging model, create a new file in the **models/staging/retail** folder called **stg\_retail\_sales\_orders.sql** and paste the following select statement into it before saving the file:   
     
   

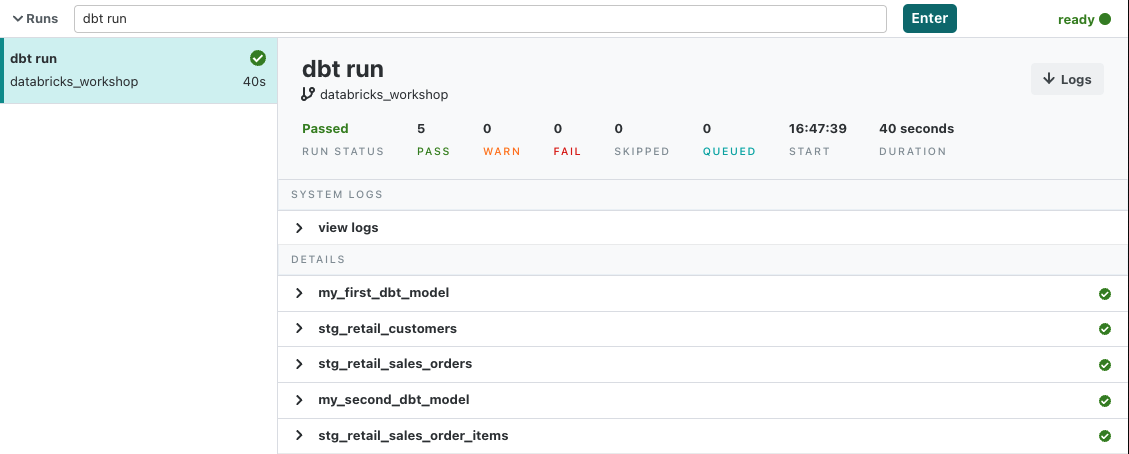
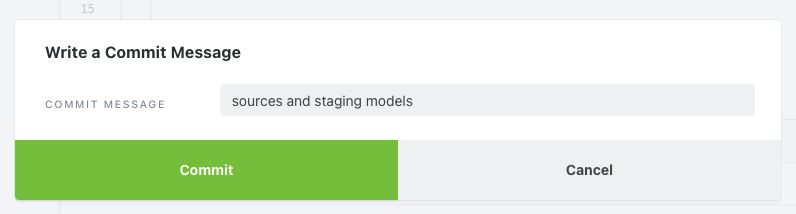
| with source as (  select \*   from {{ source('retail', 'sales\_orders') }} )  , renamed as (  select clicked\_items  , customer\_id  , customer\_name  , cast(number\_of\_line\_items as int) as number\_of\_line\_items  , from\_unixtime(order\_datetime,'yyyy-MM-dd') as order\_date  , order\_number  , ordered\_products as order\_items  , promo\_info  from source )  , duplicate\_orders as (  select order\_number  , max(number\_of\_line\_items) as max\_number\_of\_line\_items  from renamed  group by 1 )  , de\_duped\_orders as (  select renamed.\*  from renamed  join duplicate\_orders  on renamed.order\_number = duplicate\_orders.order\_number  and renamed.number\_of\_line\_items = duplicate\_orders.max\_number\_of\_line\_items )   select \*   from de\_duped\_orders |
| --- |

Again, make sure to save the file after you’re done copying and pasting!

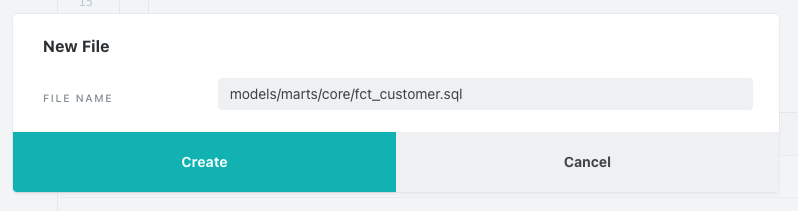
1. Last, we’ll create our order items staging model. Create one more new file in the **models/staging/retail** folder called **stg\_retail\_sales\_order\_items.sql** and paste the following code into the file before saving:   
     
   

| with sales\_orders as ( select \*  from {{ ref('stg\_retail\_sales\_orders') }} )  , explode\_orders as ( select order\_number  , customer\_id  , customer\_name  , order\_date  , number\_of\_line\_items  , explode(order\_items) as ordered\_items\_explode  from sales\_orders )  , order\_items as ( select order\_number  , customer\_id  , customer\_name  , order\_date  , number\_of\_line\_items  , ordered\_items\_explode.curr as currency  , ordered\_items\_explode.id as product\_id  , ordered\_items\_explode.name as product\_name  , ordered\_items\_explode.price as price  , ordered\_items\_explode.promotion\_info as promotion\_info  , ordered\_items\_explode.qty as quantity  , ordered\_items\_explode.unit as unit  from explode\_orders )  select \*  from order\_items |
| --- |

Last time, save that file!  
  
The array/JSON column that has the order items data that we’ll need for our final transformations is **order\_items**, which includes product names, prices, and quantities. And given that there are nested JSON objects within the outer array, we want to convert each one of those blocks into their own row in this model. To do this we’re using the **explode** function and the end result is a model where each **order\_number** has the same number of rows as there are nested objects in the source data. (ie. If an order number has 2 nested objects in the source array, the new model will have 2 rows for that order number, one row for each product ordered). Great! We now have order items.

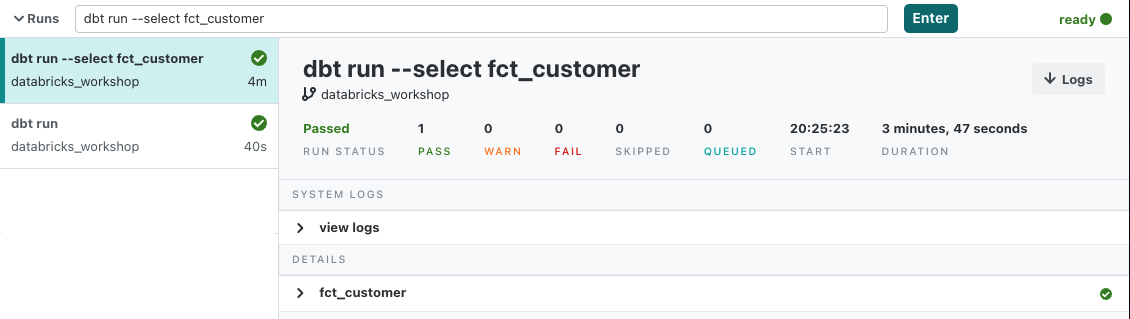
1. You’ll notice in the first CTE of each file we use a similar piece of code with a couple different dbt functions that are very important to dbt development. In the first two files we’re using the source function instead of a direct hardcoded database reference. This serves a couple different purposes, but the important thing to know here is that the source function creates a dependency between our source database object and our staging models. When we get to our documentation section we’ll be able to see that this dependency is really important for building our lineage graph.  
     
   In our last staging model you’ll notice that we’ve switched from using the source function to the ref function, which is arguably the most important function in dbt. Similar to the source function we’re using the ref function instead of a hardcoded database object, but here we’re doing so to reference another dbt model we’ve already created. We always want to use the ref function to reference other dbt models within our project for a number of reasons, like for building lineage as mentioned above as well as for enabling easy promotion of code through different environments, which you can read more about [here](https://docs.getdbt.com/docs/building-a-dbt-project/building-models#building-dependencies-between-models).
2. Ok, now that we’ve got our staging models built and saved, let’s create these models in Databricks! To do this, we’ll do another **dbt run** at the command line. This command will run all of the models in our project, which at the moment are the three staging models and the example models. Later we’ll look at how to run a specific subset of models so that you don’t have to run the entire project every time you create a new model or modify an existing one. Your run results should look something like this:  
     
   
3. And before we move on to the next section, be sure to commit your new models to your git branch by clicking on the commit button and giving your commit a descriptive message like **sources and staging**.  
   

# Fact Model

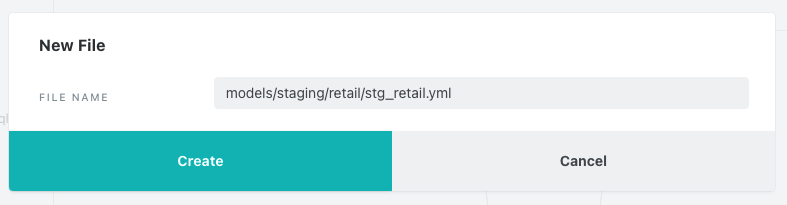
1. Now that we have our staging models built and our orders transformed to order items, we’re able to build our final model to show some key metrics by customer. This model will be what our end users use to build dashboards in their BI tool of choice, as well as to query directly if needed.
2. Start by creating a new file called **fct\_customer.sql** in the core folder.  
     
   
3. Next you can paste the following code block into your new model and click save:

| with customers as (  select \*  from {{ ref('stg\_retail\_customers') }} )  , orders as (  select \*  from {{ ref('stg\_retail\_sales\_order\_items') }} )  , customer\_calcs as (  select customer\_id  , sum(price \* quantity) as total\_order\_amount  , count(distinct order\_number) as total\_order\_count  , min(order\_date) as first\_order\_date  , max(order\_date) as last\_order\_date  from orders  group by 1 )  , final\_join as (  select customers.customer\_id  , customers.customer\_name  , customers.tax\_id  , customers.tax\_code  , customers.state  , customer\_calcs.total\_order\_amount  , customer\_calcs.total\_order\_count  , customer\_calcs.first\_order\_date  , customer\_calcs.last\_order\_date  from customers  join customer\_calcs  on customers.customer\_id = customer\_calcs.customer\_id )  select \*  from final\_join |
| --- |

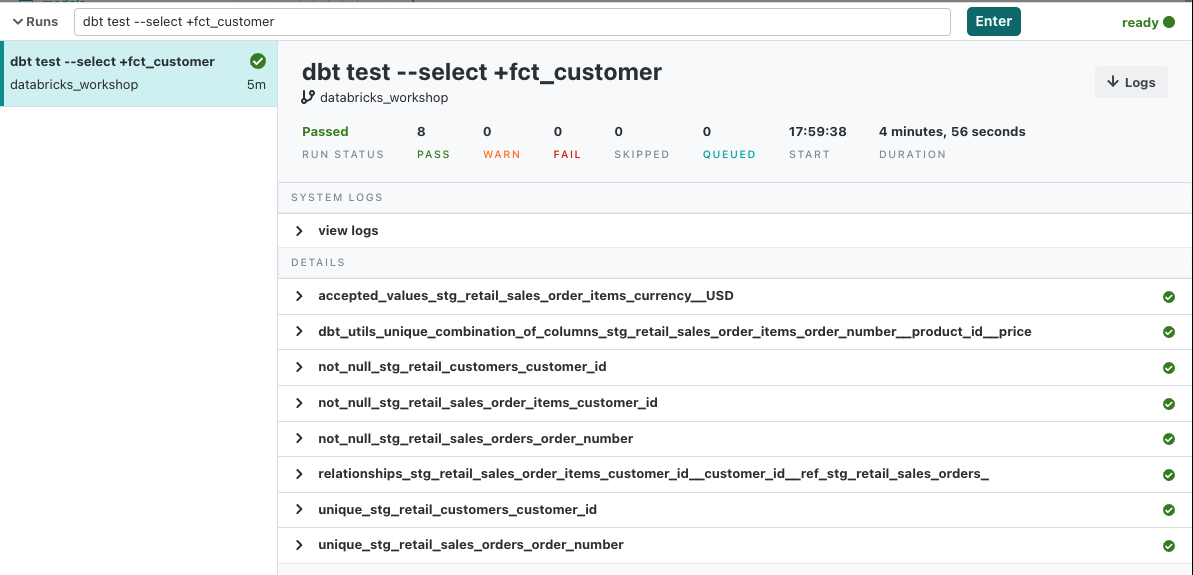
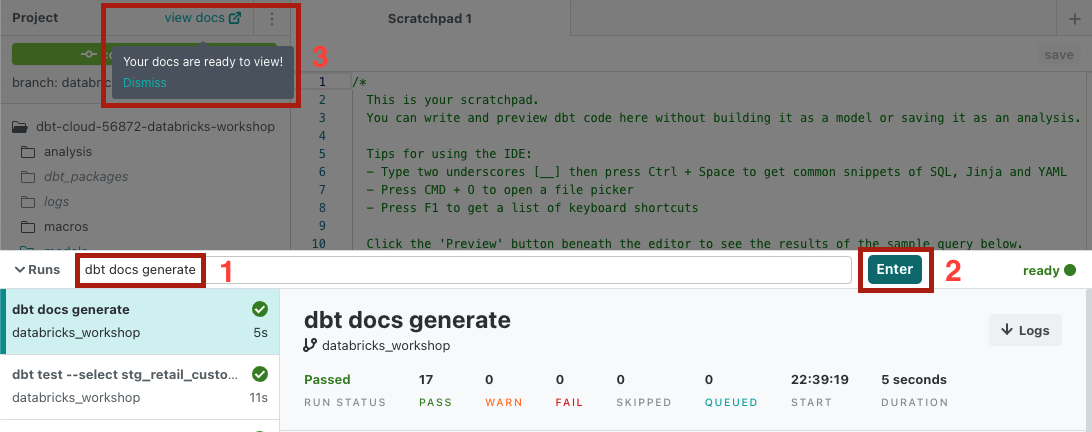
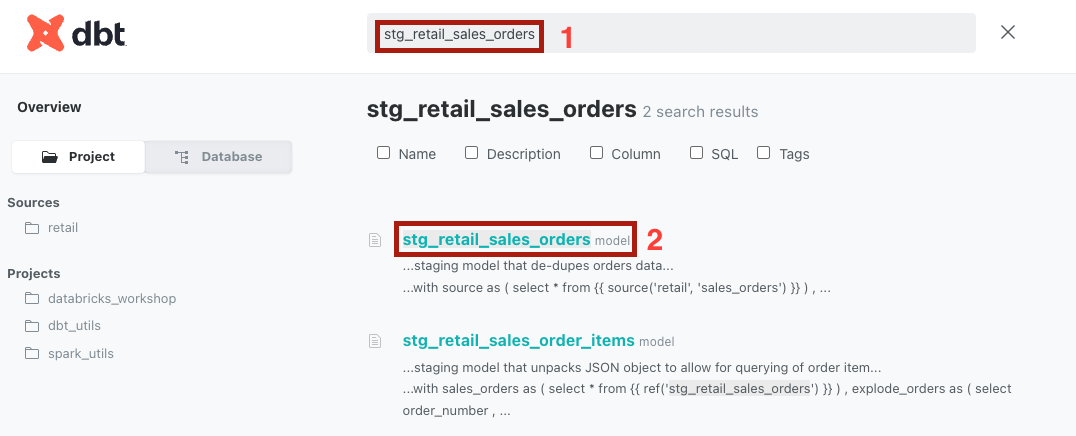
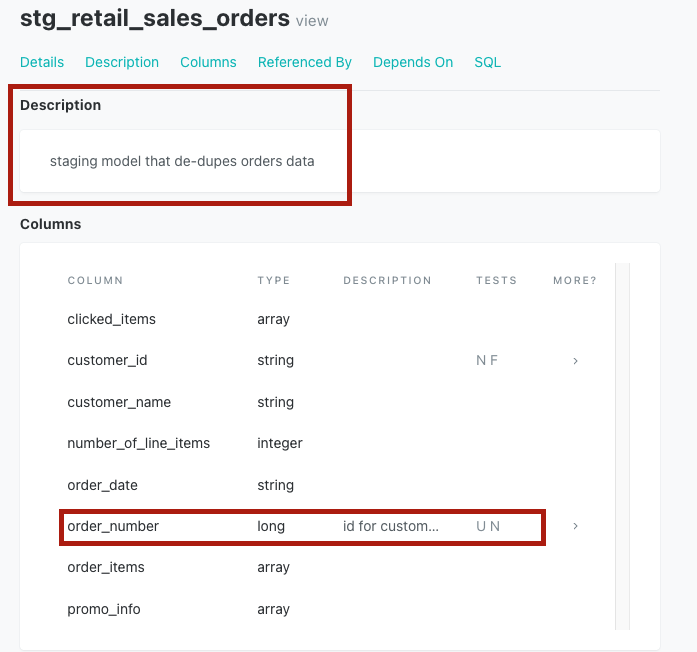
As you can see, the transformation here is pretty straightforward. We’re doing some simple aggregations on order items at the **customer\_id** level and then joining that back with the **customers** model to get a couple other pieces of customer information.

1. And let’s do another **dbt run**, but this time we’ll tell dbt to materialize just this model with the following command: **dbt run --select fct\_customer**. By using the **--select** argument we are telling dbt to specifically run the model or path that we provide after the argument, in this case just our **fct\_customer** model. The results should look like this:   
     
   
2. Now that we’ve built our complete series of models we can take full advantage of the lineage feature in the IDE to see how all of our models relate to each other. If you’re not already there, navigate to the **fct\_customer** model in the IDE and then click on the **Lineage** button in the lower window pane.You should be able to see the lineage graph below:  
     
     
     
   This lineage graph shows the relations between all of our models, from our green source nodes all to the way to our final **fct\_customer** model. In the sources and staging section we touched on the significance of the **ref** and **source** functions and the role that they play in creating this lineage graph. The usage of those functions to refer to raw data sources and other models is what allows dbt to create a relation between all of the nodes in this graph and create this visual. This is a powerful feature that gets more and more powerful as a project grows to hundreds of source nodes and models.
3. Before moving on let’s make sure that we commit our work, so take a moment to click the commit button and write a descriptive message for the work that will be added with this latest commit.

# Tests and Documentation

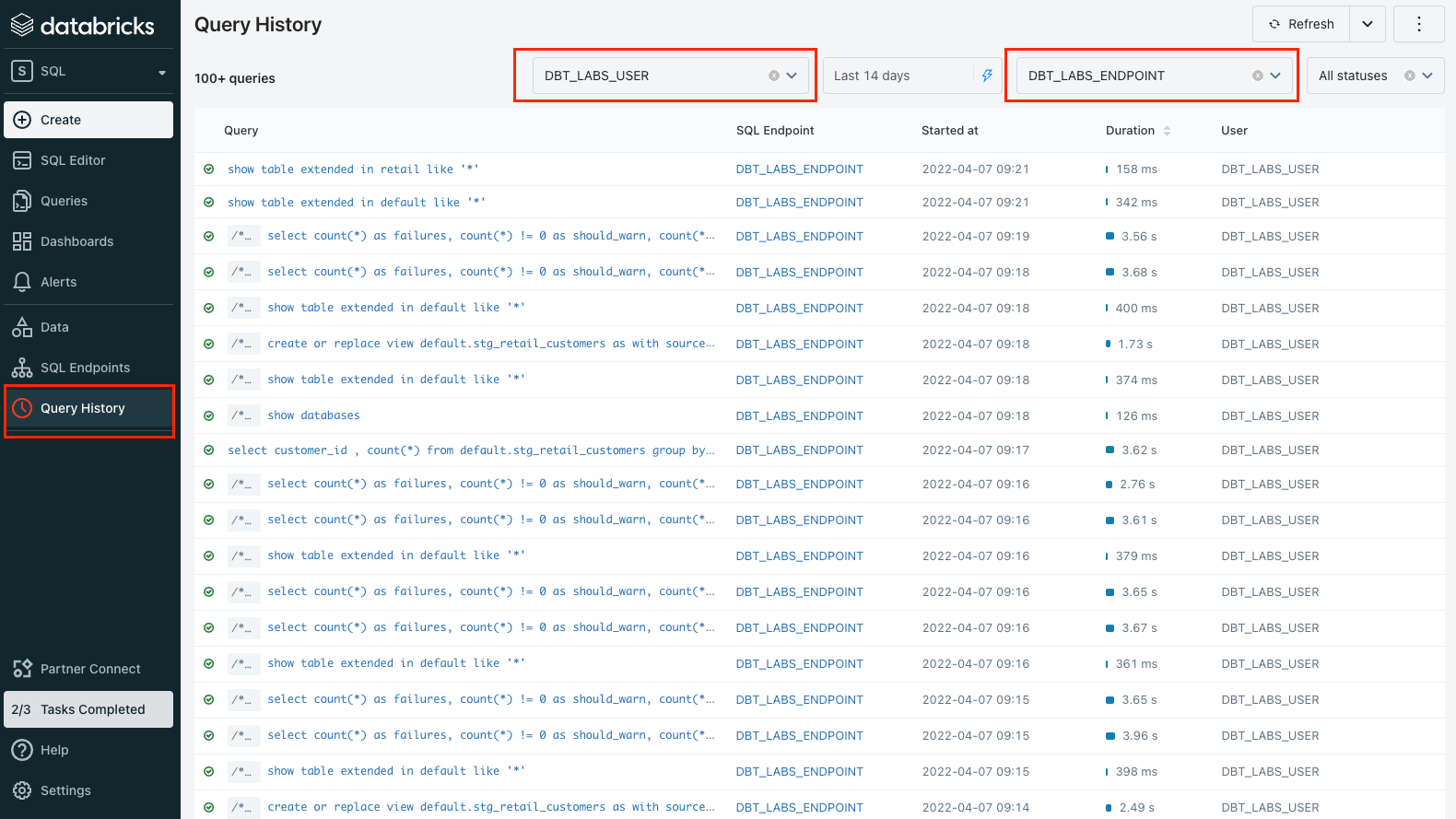
1. Now that we’ve built out our models and transformations, it’s incredibly important to test and document our transformations. This ensures that we catch any errors that may have flown under the radar, as well as provides a guide to anyone else that comes across our work and wants to know what is what when they take a look at our models. dbt’s native features include both a data testing and documentation framework to help cover all the bases here.
2. dbt comes with a set of four generic tests that work out of the box that we’re going to use here: **unique**, **not null**, **accepted values**, and **relationships**. We’ll also leverage pre-written tests from the package that we installed earlier to quickly expand our testing capabilities without having to write any more code. It should be mentioned that you can write your own custom tests in the form of SQL select statements that can be implemented throughout your project. While we recommend adding tests to all of the models and sources throughout your dbt project, we’re only going to implement them on our staging models in this workshop.
3. When it comes to documentation, dbt brings together information that you provide (model and column level descriptions) along with details from your Databricks workspace in a neat package for consumption by other data team members as well as stakeholders.
4. Tests and descriptions for documentation are both defined in YAML files, so to get started we’ll create a new file in the **models/staging/retail** directory called **stg\_retail.yml**.  
     
   
5. Next you can paste the following code block into your new model and click save:

| version: 2   models:  - name: stg\_retail\_customers  description: staging model that includes only current customer records  columns:  - name: customer\_id  description: unique id for each customer  tests:  - unique  - not\_null    - name: stg\_retail\_sales\_orders  description: staging model that de-dupes orders data  columns:  - name: order\_number  description: id for customer orders  tests:  - unique  - not\_null   - name: stg\_retail\_sales\_order\_items  description: staging model that unpacks JSON object to allow for querying of order items  tests:   - dbt\_utils.unique\_combination\_of\_columns:  combination\_of\_columns:  - order\_number  - product\_id  - price  columns:  - name: customer\_id  tests:  - not\_null  - relationships:  to: ref('stg\_retail\_sales\_orders')  field: customer\_id  - name: currency  description: currency of the product price  tests:  - accepted\_values:  values: ['USD'] |
| --- |

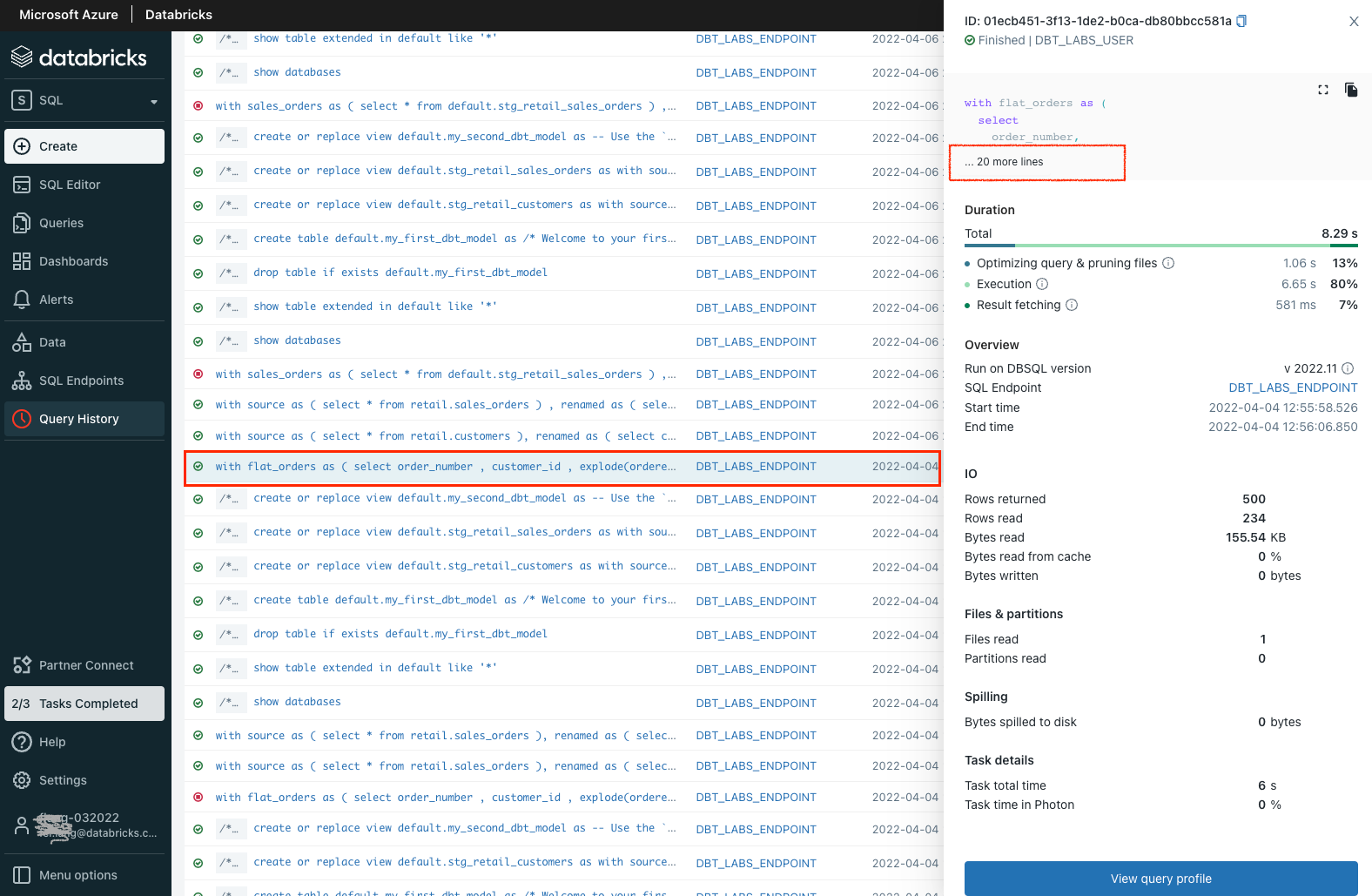
1. Before running our tests and creating our docs, let’s take a moment to describe what’s happening in the YAML file and how we’re defining everything. This YAML file is in the retail directory and contains tests and descriptions specifically for models in this directory. This is simply an organizational step and it is ultimately up to you how you’d like to organize your YAML files and the models they test and define, but we strongly recommend having at least one YAML file for testing and documentation per directory.   
     
   As far as formatting goes, we start at the top level by naming the model we’re going to describe and then providing any model level information we’d like to. In our case, we’ve provided descriptions for all three staging models and these will show as the descriptions at the model level when we open up the documentation site for our project. There are also some tests that can be performed at the model level and for the **stg\_retail\_sales\_order\_items** model we’re doing just that with a test from the **dbt\_utils** package. Given that this model fans out records on the primary key from the source, we now have three different columns that we must check to ensure uniqueness in a row. While there are a couple of different ways we can go about checking for uniqueness over a combination of columns, the **dbt\_utils** package provides a handy test right out of the box that we can use: **unique\_combination\_of\_columns**. This test allows us to name the combination of columns that we’d like to check for uniqueness and dbt takes care of the rest. This is just one example of many helpful tests and macros that exist in dbt packages and can be easily added to your project so that there’s no need to rewrite code that has already been written.   
     
   Once we’re done with model level descriptions and tests, we can move down to our column level descriptions and tests where we’ll be doing most of our work here. While it’s always best to provide as much documentation and testing as possible, a good rule of thumb is that every model should have at minimum a **unique** and a **not\_null** test. Here we’ve got a number of columns defined and tested, starting with the **customer\_id** column in the **stg\_retail\_customers** model that we expect to be **unique** and **not\_null**. For **stg\_retail\_sales\_order\_items** we’re using the **relationships** test to make sure that every **customer\_id** that exists in our order items model also exists in our initial orders model. And also in the **stg\_retail\_sales\_order\_items** model we’re using the **accepted values** test to check and make sure that the only currency we see is USD. To top it all off, we’ve got descriptions for each column we’re testing listed just under the column name that we’ll see when we launch the documentation site for the project.
2. Alright, it’s time to run these tests. There are also tests that come with the example files as part of your initialized dbt project, but we only want to run the tests we defined for the models we built. The command we’re going to use to do that is **dbt test --select +fct\_customer**. The syntax here is similar to what we used to build our fact model earlier, with a couple notable differences:  
     
   - **dbt test** is going to specifically run tests on the models that we tell dbt to look for  
   - We selected the **fct\_customer** model, but with a plus sign at the front of the model name. Why? The plus sign is a graph operator and if it’s placed at the front of a model name, it tells dbt to select that model and every model upstream of that model, and vice versa if it is placed at the end of a model name. You can read more about graph operators [here](https://docs.getdbt.com/reference/node-selection/graph-operators).  
     
   The end result is that dbt will look for tests on **fct\_customer** and every model and source upstream of **fct\_customer**, which will catch all of the associated tests on our staging models. Even though there aren’t any tests on **fct\_customer** right now, as soon as you add them it will run those as well. Here’s what your command will look like:  
     
     
     
   When we look at the results it appears that all of our tests have passed. Great!  
   
3. Now we’re ready to take a look at the documentation for our project. The command we’ll need to run to tell dbt to build our documentation is **dbt docs generate**. Run that command and after it’s finished you should see an icon pop up next to the **view docs** button in the upper left hand corner of your IDE just above the green git workflow button to notify you that your new docs are ready.  
     
     
     
   Clicking on **view docs** will launch the documentation site in a new tab. Once inside, type **stg\_retail\_sales\_orders** into the top search bar and click on the top result to take you to that model.  
     
   Once the model page loads you should be able to see both our model level descriptions and column level descriptions, as well as the tests we implemented next to the column names.  
     
   
4. And that’s it for our dbt features!

# Review dbt compiled queries in Databricks SQL

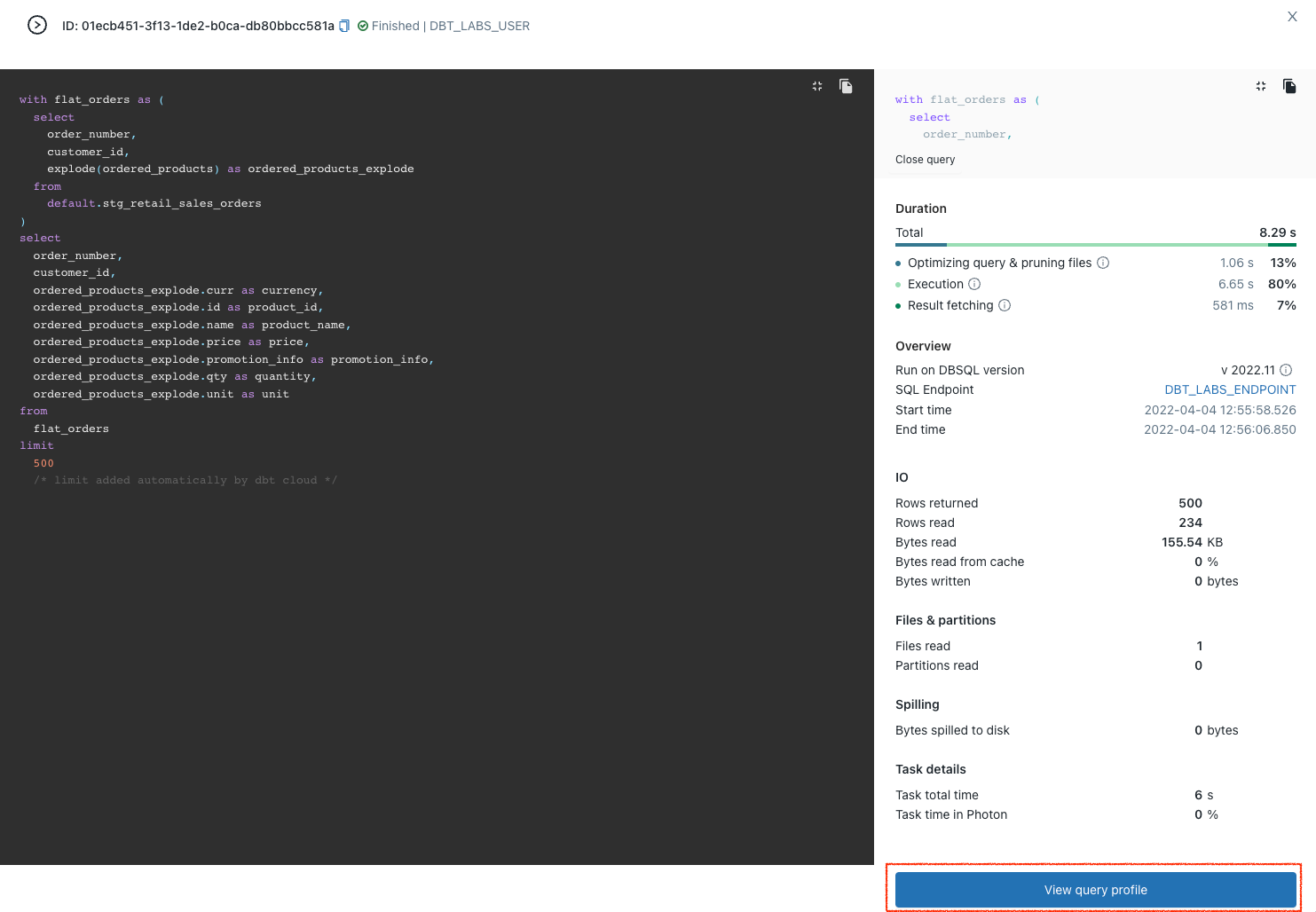
1. We’ve just completed the end to end data analytics pipeline leveraging dbt Cloud and Databricks. Now let’s go back to Databricks UI to have a look at the dbt compiled queries that run against Databricks SQL. Click the **Query History** icon on the left sidebar to navigate to the **Query History** screen. We can see a list of SQL queries performed using [SQL warehouse](https://docs.databricks.com/sql/admin/sql-endpoints.html) in this workspace. To find the queries which got pushed down from dbt, we can filter the list by user as “*DBT\_LABS\_USER*” and SQL warehouse as “*DBT\_LABS\_ENDPOINT*”. Remember those were automatically created in Databricks Partner Connect when creating the connection to dbt Cloud..



1. Click a query from the list to view the details of it, including its duration, I/O performance, and etc.



1. You can see the complete query by clicking “... x more lines” if the query is not shown completely. Like this, we can see the entire query got pushed down from dbt Cloud.

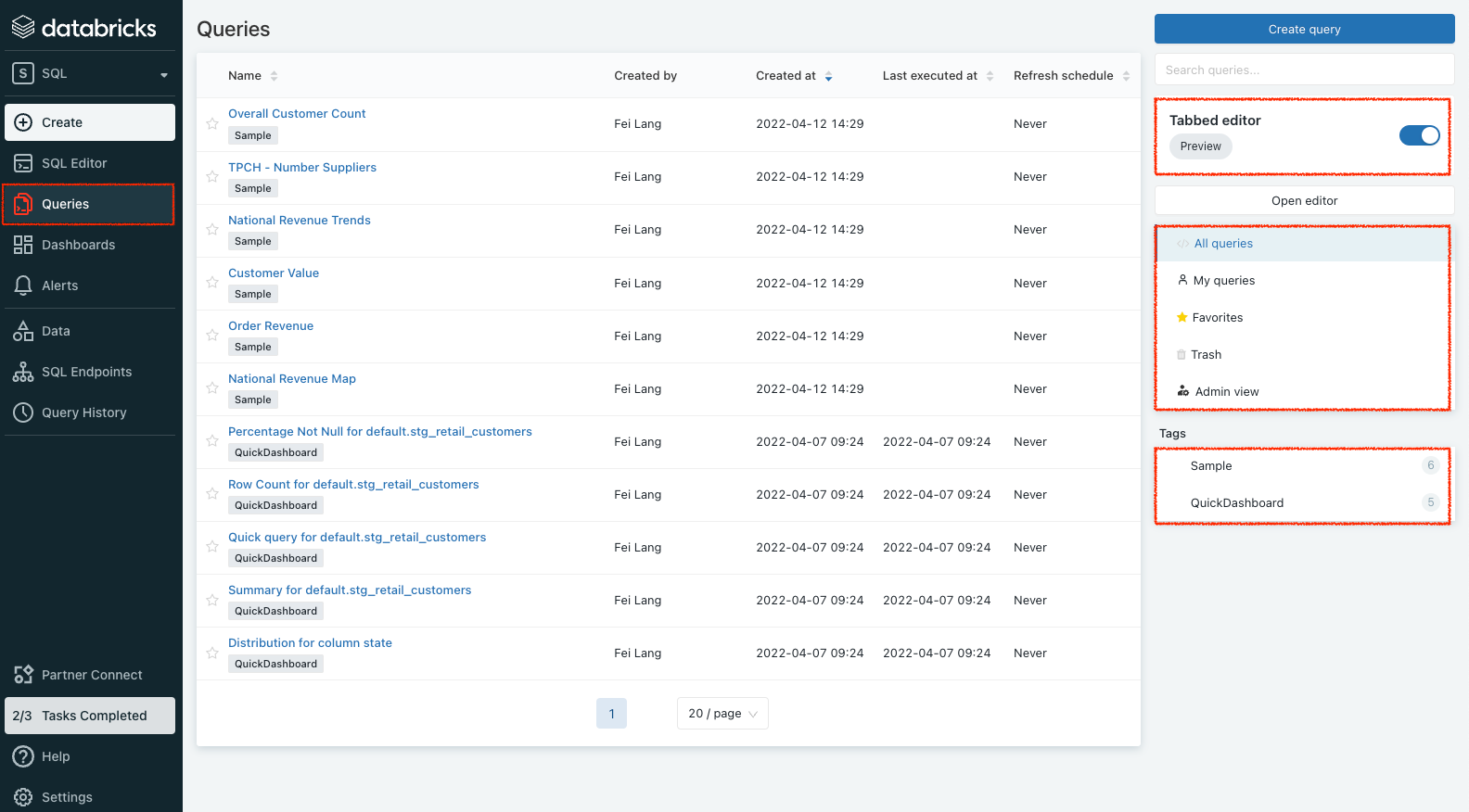


Click **View query profile** at the bottom for more detailed information about the query’s performance, such as its execution plan and so on.

# Ad-hoc data analysis and visualization in Databricks SQL

So far dbt has done the heavy lifting on the data transformation and has created a cleaned final model of customer data, next we are going to perform ad-hoc and exploratory data analysis on the data set and quickly develop a simple dashboard.

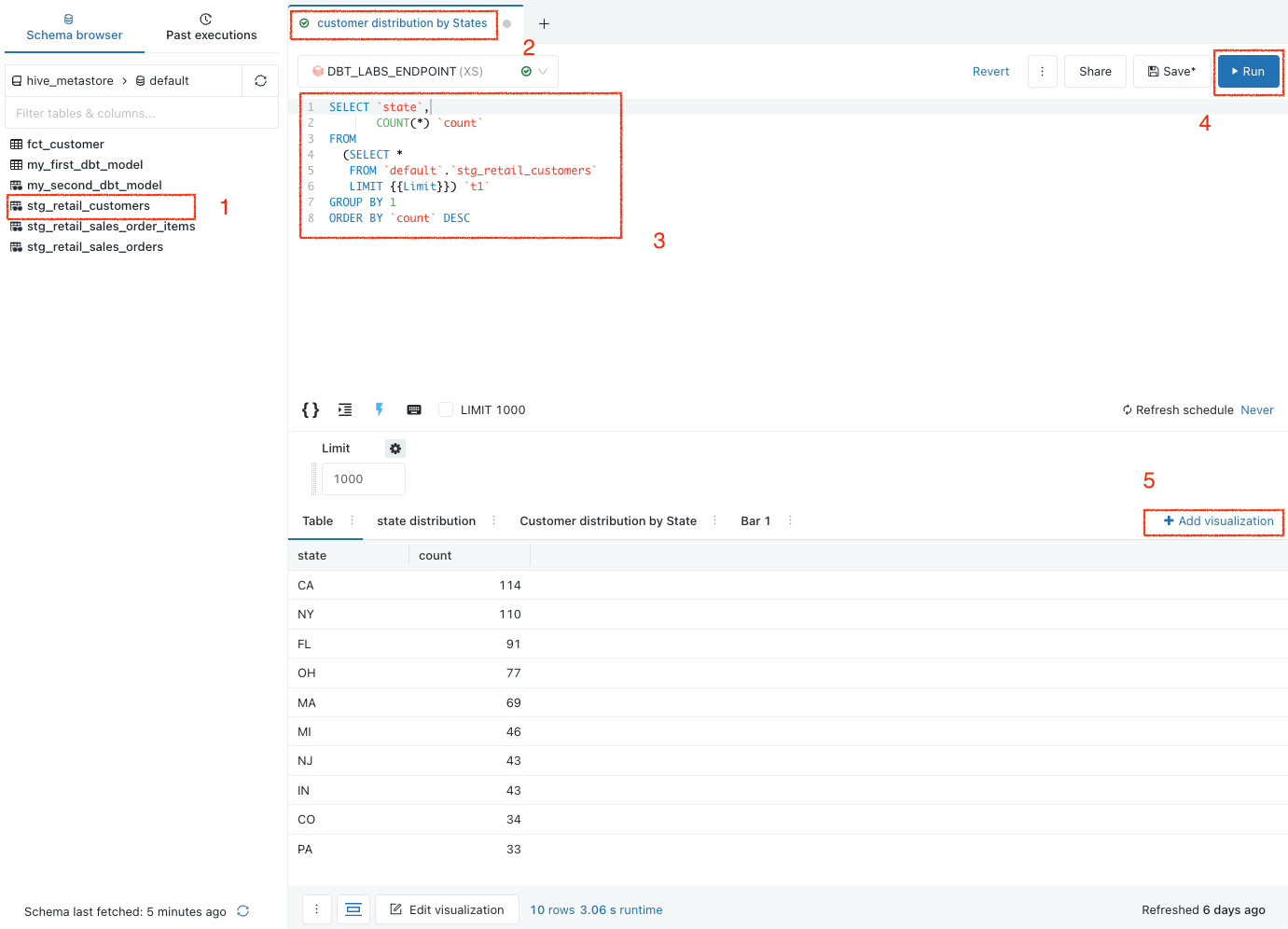
1. Click the **Queries** icon on the left sidebar to navigate to the **Queries** page. For each query listed, you can see who created it, when it was created, last executed or if there is a refresh schedule attached. Within this view, you can create a new query or search for existing queries. Different from the **Query History** page which we just went through earlier, the queries shown in the list on the **Queries** page here are saved ones which can be rerun manually or automatically.



Notice the **Tabbed** editor which is enabled by default. It is a preview feature that allows you to open multiple queries without opening a new query editor. And below this preview feature, You can view queries you’ve created by clicking **My queries** or **Favorites** or deleted queries which have been sent to **Trash**. The **Admin view** gives you access to see all queries created and delete queries in this workspace. However, an admin can’t edit a query if it is not shared with the admin. Lastly, you can filter the query view by tags.

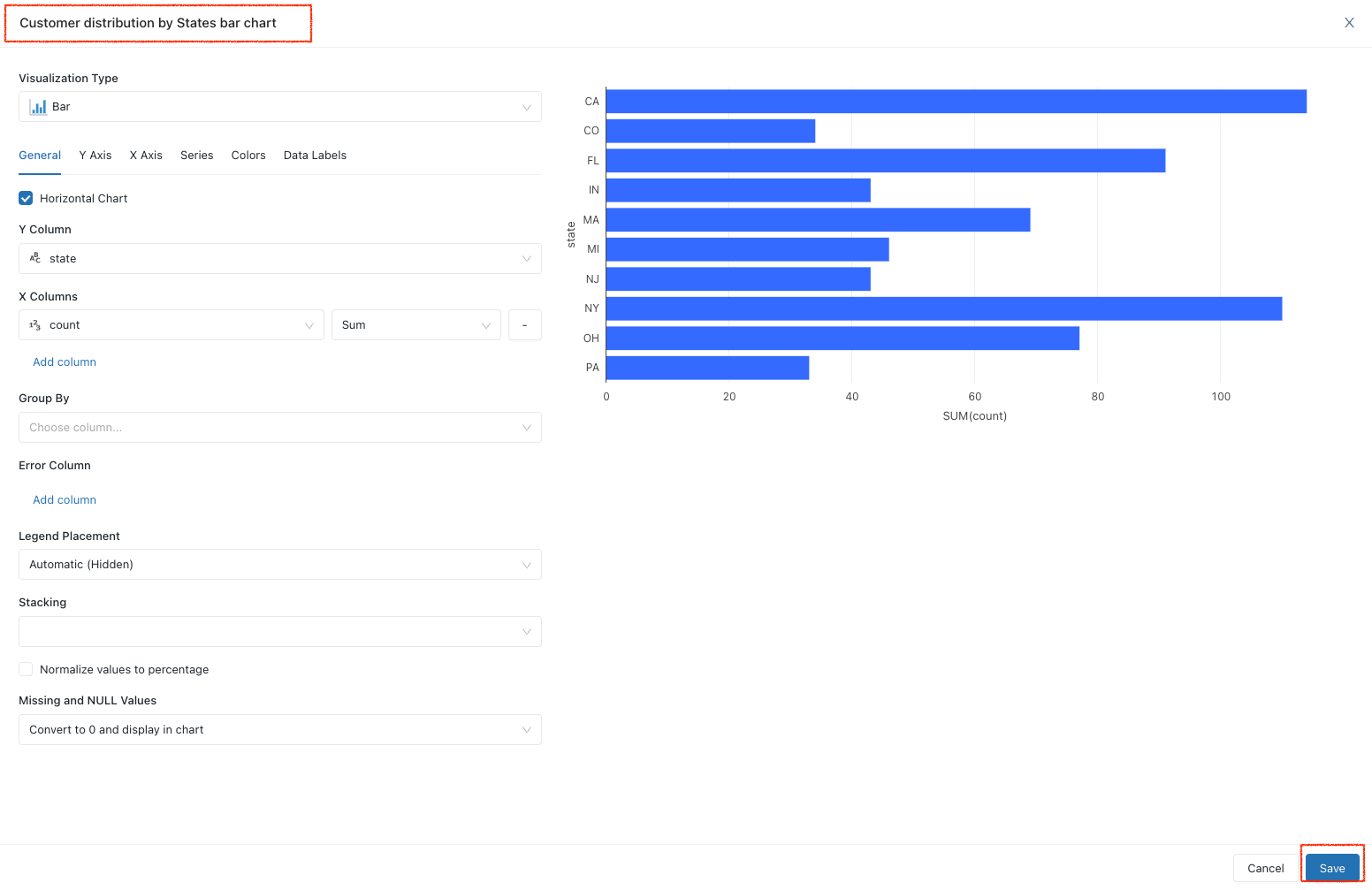
1. We want to create a simple customer distribution by States using dbt transformed dataset *fct\_customer*. To start, click **Create query** from the top right corner to open a new query editor. Then copy and paste the following SQL statement. Replace **New Query** in the tab with a meaningful name. For example, “customer distribution by States”.

| SELECT `state`,  COUNT(\*) `count` FROM  (SELECT \*  FROM `default`.`fct\_customer`  LIMIT 1000) `t1` GROUP BY 1 ORDER BY `count` DESC |
| --- |

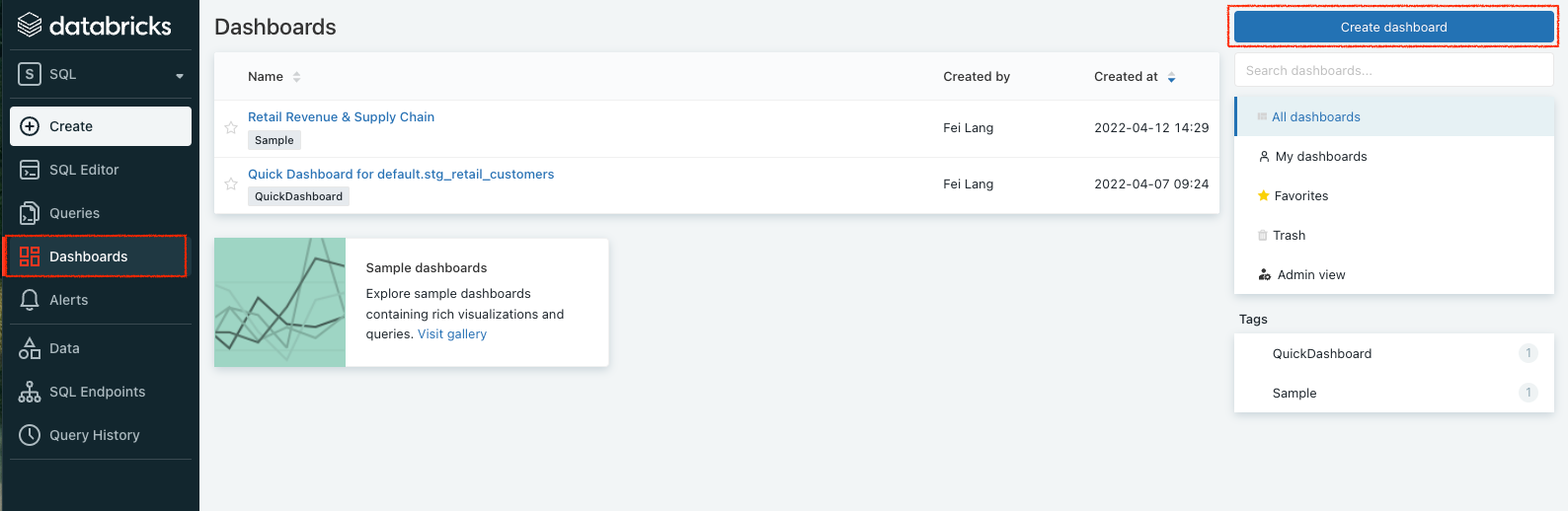


Click run until you see the query result at the bottom of the screen. Then, click **Add visualization** to visualize the result.

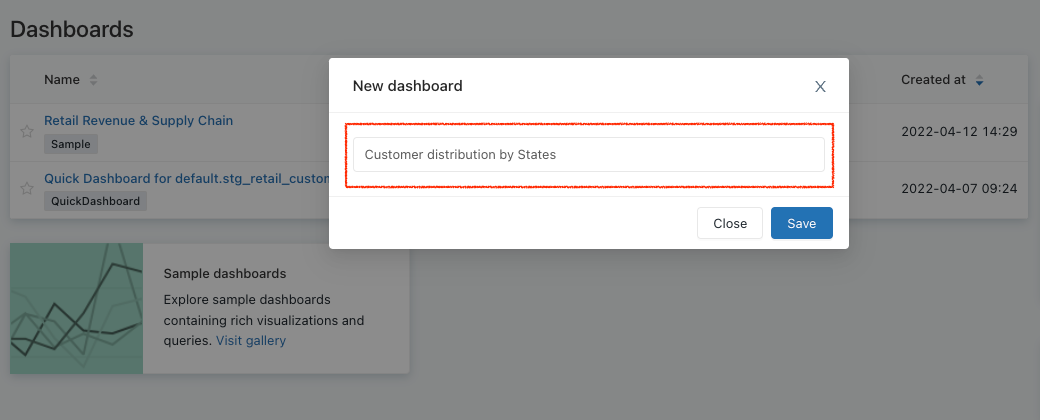
1. On the next screen, give the visualization a meaningful name and click the **Save** button at the bottom.



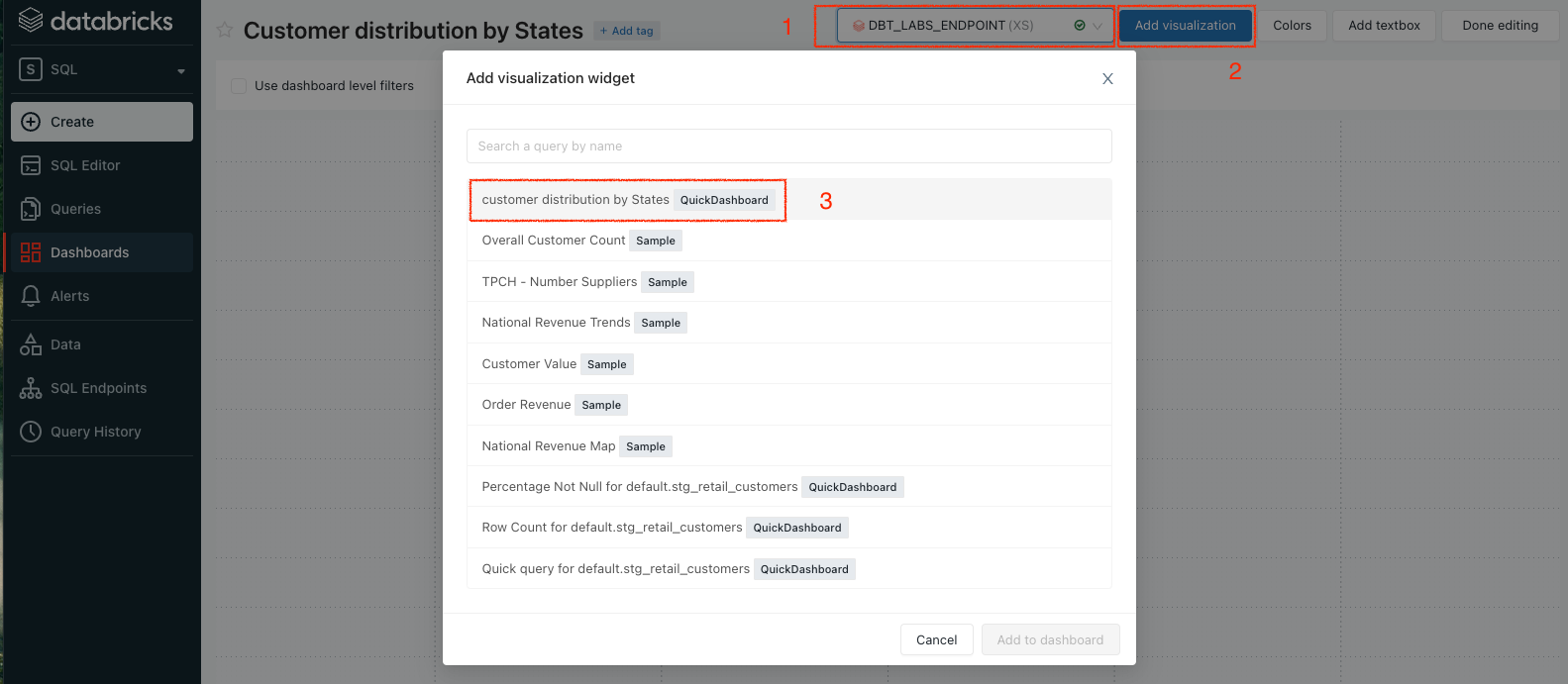
1. Now we go to the Dashboards UI by clicking the **Dashboards** icon on the left sidebar. Then click **Create dashboard**.



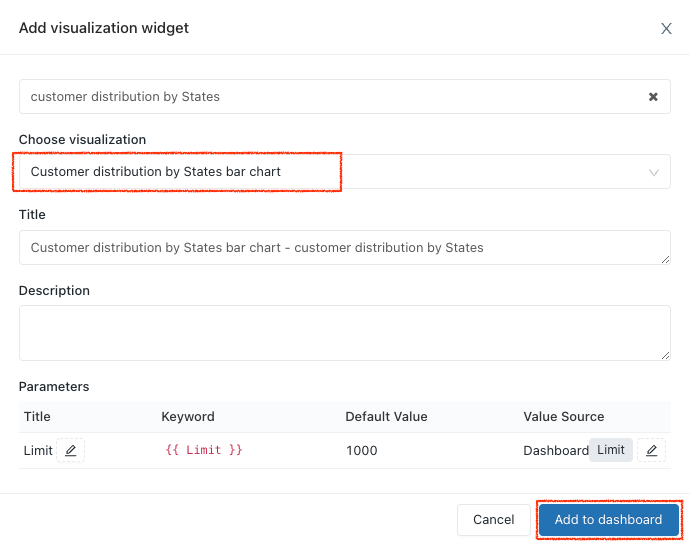
Give it a name on the next popout window. “Customer distribution by States” is what we used for this lab. Save it.



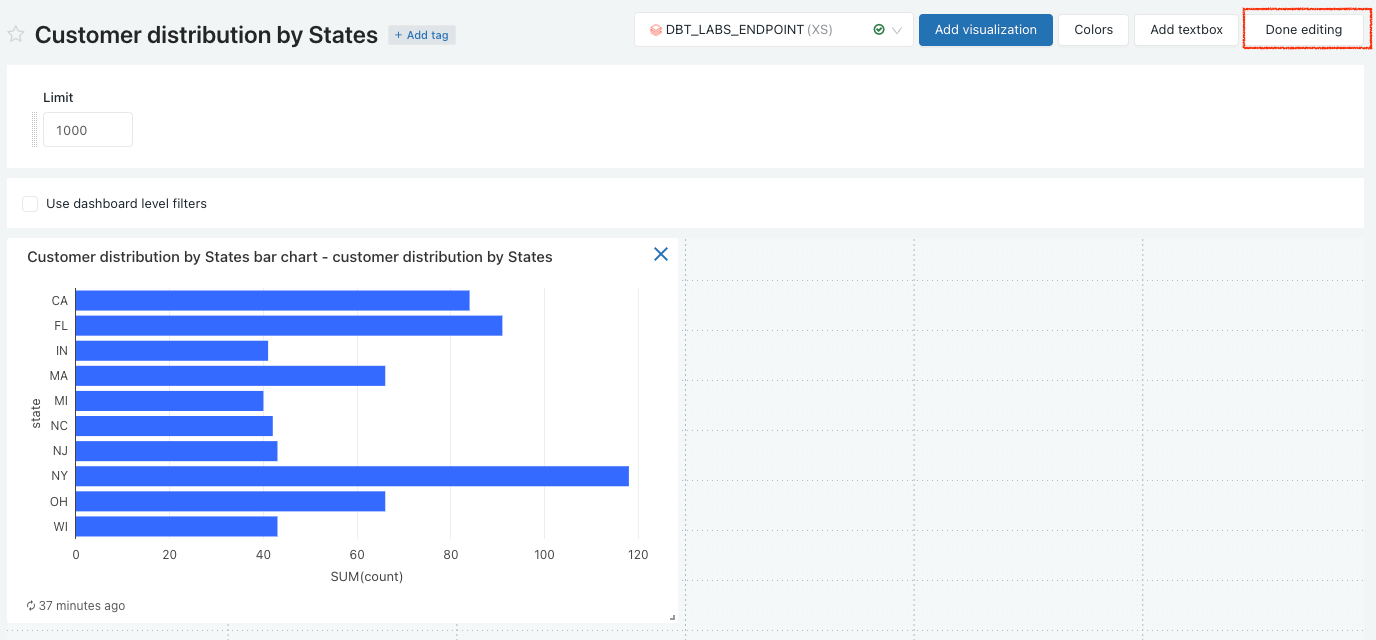
1. It creates an empty dashboard with the name that we just gave.



On this screen, pick the warehouse that we’ve been using for this lab which is “DBT\_LABS\_ENDPOINT”. Click **Add visualization**. You will be able to see Visualization “Customer distribution by States bar chart” which we created earlier for the same query saved. Select it and click **Add to dashboard**.

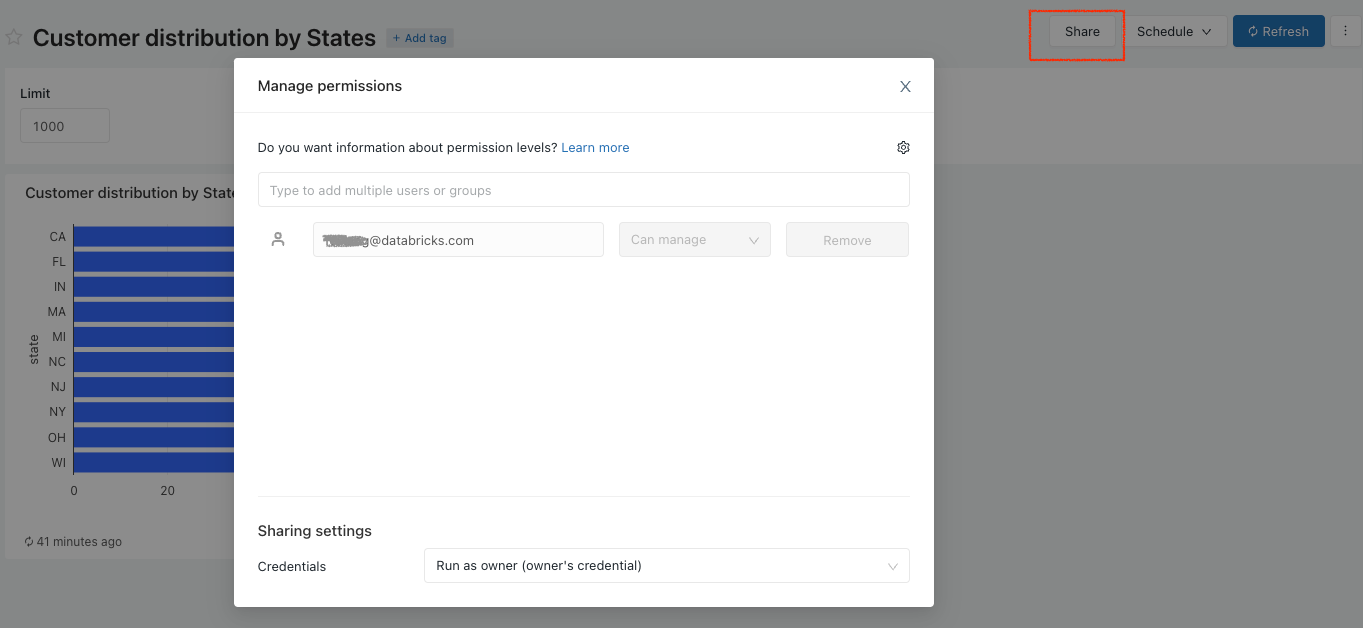


1. A simple dashboard is created!

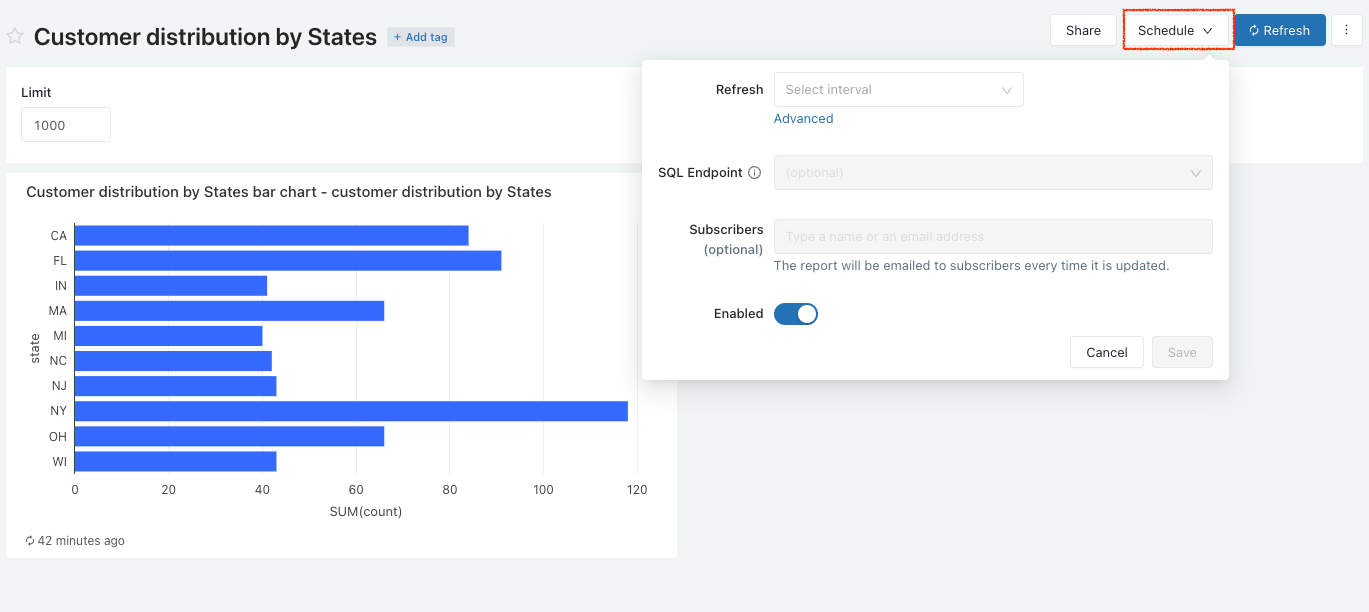


Click **Done editing**.

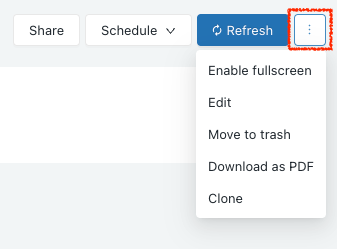
You can click **Share** to share this dashboard with someone in the same workspace with the corresponding access permissions.



You can also set a dashboard refresh schedule by clicking Schedule and entering the right settings.



And of course, you can enable fullscreen for the dashboard, edit, delete, clone or download it as PDF by clicking the three dots icon next to the **Refresh** button and find the right option.



# 

# Conclusion & Next Steps

Congratulations on completing the lab!

Today, you learned how to use dbt and Databricks to build a scalable transformation pipeline from scratch. You're now ready to apply these fundamentals to your own data. We encourage you to continue with your free trial by loading your own sample or production data, and by continuing to dive into some of the more advanced functionality of dbt Cloud and Databricks.

What we've covered:

* How to set up dbt via Databricks Partner Connect
* How to build scalable data transformation pipelines using dbt & Databricks
* How to leverage data in Databricks SQL to quickly find insights with the built-in SQL editor, visualizations and dashboards

# 

# Additional Resources

* dbt integration with Databricks
  + Join our [dbt community Slack](https://community.getdbt.com/) which contains more than 18,000 data practitioners today. We have a dedicated slack channel #db-databricks-and-spark for Databricks related content.
  + To continue to learn to use dbt more effectively, check out the [dbt Learn site](https://learn.getdbt.com/).
  + Contact the [dbt Cloud Sales team](https://www.getdbt.com/contact/) if you're interested in exploring dbt Cloud for your team or organization.
* Databricks SQL
  + To learn more about [Databricks SQL](https://databricks.com/product/databricks-sql).
  + On-demand hands-on workshop: [Using Databricks SQL for Analytics on Your Lakehouse](https://databricks.com/p/webinar/hands-on-workshop-using-databricks-sql-for-analytics-on-your-lakehouse)