DBT Hello World with DuckDB

Josh Patterson

Contents

Introduction

DBT (Data Build Tool) is an open-source command-line tool that allows data analysts and data engineers a way to manage a collection of data transformations written in SQL or Python for analytics and data science. It provides a framework for creating and executing complex SQL queries and transforms, and it helps manage the entire data transformation process.

More simply, DBT is focused on transforming data — the 'T' in ELT.

DuckDB is an embedded database (think "SQLite"), but designed for OLAP-style analytics instead of OLTP. DuckDB is exceptionally fast and lets you work directly with data stored in files such as CSV and Parquet files directly, without forcing you to do any load operations first.

Together, with the `dbt-duckdb` project, they form a "Modern Data Stack In A Box" — or a simple and powerful data lakehouse (sans Java and Scala).

In this tutorial you will:

- prepare some metrics from raw patient checkup visit data for use in a machine learning pipeline
- use DBT to produce the summary statistics, or metrics, from the raw data sitting in DuckDB
- use DuckDB here as a simple local option so as to not force the reader to set up any cloud infrastructure

`Editing Note: Metrics and Models are two distinctly different things in DBT --- need to clea`

Further, you'll learn:

- a gentle introduction to DBT and DuckDB
- an understanding of how DBT fits into modern data pipelines
- an understanding of how DuckDB and DBT together can create a simple data stack for local analysis

We'll build our DBT "Hello World" project with synthetic patient data.

Building Our DBT Hello World Project with Synthetic Patient Data

In our "hello world" scenario, we want to build some metrics about how often a set of patients get their yearly physical checkup. The hypothetical company here is an insurance company evaluating a set of customers and building forecasts with machine learning to predict how much the customer will potentially cost in terms of healthcare while under an insurance policy.

Patient Metrics

> "The most granular form of data, Metrics describe the exact numbers that make up the data. Put more simply, they are the raw ingredients that make analytics possible. On their own, they may not actually be very helpful, but studied in the right context (foreshadow alert – this is analysis), they can be used to give fact-based direction to your decision-making.""

Examples of metrics in the Marketing Data world:

- Number of Users
- Sessions
- Pageviews
- Event actions

> "All of these units of measurement report activity or results of very specific user interactions in your marketing efforts"

The Scenario

The data engineering team has requested that the analysts pull the data from the (cloud) data warehouse and provide the metrics per customer for use in their modeling experiments as a feature.

In this exercise you'll take synthetic patient checkup history data (which insurance companies are able to access) and use DBT to build data models and produce the patient metrics. We'll use DuckDB for our database as its a quick and easy database to use locally and keeps us from having to provision cloud infrastructure for a simple Hello World example.

The Synthetic Patient Doctor Visit Data is shown below:

```
Date,PatientID
2016-09-05,pid-001
2015-08-17,pid-002
2013-07-18,pid-003
2015-09-22,pid-003
2015-10-13,pid-003
2019-01-08,pid-003
2021-07-24,pid-003
```

So let's dig into build a local data stack.

Building a Local Data Stack with DBT and DuckDB

Our tasks for this exercise are listed below:

- Create a new conda environment
- install DuckDB and connector dbt-duckdb
- load raw synthetic patient checkup data into DuckDB
- Create a new DBT project
- Configure/Test the DBT project for access to DuckDB
- Run our DBT pipeline

Let's start off by building a new environemnt in Conda for our project.

Create Environment in Conda

To create an environment in Conda, you can follow these steps:

1. Open your terminal (or Anaconda Prompt on Windows).
2. Type the following command to create a new environment named my_env (you can replace my_env with your preferred name):

```
conda create --name my_env
```

3. Conda will ask you to confirm the installation of any additional packages that are required to create the environment. Type y and press Enter to proceed.
4. Once the environment is created, activate it by typing:

```
conda activate my_env
```

5. Your prompt should now show the name of the environment in parentheses, indicating that it's active.
6. You can now install any packages you need for your project using conda install or pip install.
7. When you're done working in the environment, you can deactivate it by typing:

```
conda deactivate
```

Now that we have an environment to work in, let's get started with using DuckDB.

Working with DuckDB

DuckDB sits in an interesting spot as it offers an excellent way to solve complex problems using SQL while keeping things simple. DuckDB is a columnar, in-memory SQL database designed for analytical workloads. It's key features are:

- Fast query performance with low memory overhead for large datasets
- SQL interface making it easy to use for the broadest audience
- Lightweight database that can be easily installed and run on a local machine, without the need for a separate server or cluster
- Open-source project, which means that it is free to use and can be modified and extended by the community

DuckDB is compelling because you only have to point it at the data file you'd like to work with, giving you a short-cut to running SQL against CSV, Parquet, and other data files. Ease of use and time to insight are two key properties of why DBT has become a key tool for experimenting in the data "lab".

In a conversation with me, JD Long commented that he prefers DuckDB "when the data fits on a single machine so that he doesn't have to stand up a cluster for Spark".

As stated by the DuckDB Website:

> DuckDB is an embedded database, similar to SQLite, but designed for OLAP-style analytics. It is crazy fast and allows you to read and write data stored in CSV and Parquet files directly, without requiring you to load them into the database first.

What is DuckDB Typically Used For?

A recent article at dlthub.com, "As DuckDB crosses 1M downloads / month, what do its users do?", had some interesting notes on how DuckDB is being used. Among the use cases, they were seeing DuckDB being used as a processing enging for local data workflows.

They also called out how many users ("Normies") enjoyed the simplicity of the DuckDB user experience . I found both of those usage trends worth of note.

With those points noted, let's move on and install DBT and the dbt-duckdb connector.

Install DBT and DuckDB Connector

As described on the website:

> dbt is the best way to manage a collection of data transformations written in SQL or Python for analytics and data science. dbt-duckdb is the project that ties DuckDB and dbt together, allowing you to create a Modern Data Stack In A Box or a simple and powerful data lakehouse- no Java or Scala required.

You can also read further about the connector on dbt's website as well:

https://docs.getdbt.com/reference/warehouse-setups/duckdb-setup

This project is hosted on PyPI so we can use pip to install the connector and dependencies:

```
pip3 install dbt-duckdb
```

Once that is installed, we'll need the DuckDB CLI to work directly with DuckDB from the local command line to load our dataset.

DuckDB CLI

The DuckDB CLI (Command Line Interface) is a single, dependency free executable.

Download it from:

https://duckdb.org/docs/api/cli.html

Make sure and get the version of the CLI that matches the version of DuckDB you have installed locally via pip.

Now let's load some data for analysis.

Load Some Synthetic Patient Data Into DuckDB

Let's crank up the local DuckDB CLI with the command:

```
./duckdb dbt_patient_visits.duckdb
```

Note: if you don't give it a database name at start, for some ODD REASON, DuckDB will not let you save the in-memory db later.

Once we're inside the CLI the first command you want to know about is how to quit:

```
.q [enter]
```

Typing .q or .quit will quit the CLI and get you back to your shell.

Now, once we're back inside the CLI, let's see if we can access our local patient data in CSV form with the command:

```
select * from './patient_checkup_logs.csv';
```

This should show:

```
    Date      Patient ID
    date       varchar

 2016-09-05   pid-001
 2015-08-17   pid-002
 2013-07-18   pid-003
 2015-09-22   pid-003
 2015-10-13   pid-003
     .           .
     .           .
     .           .
 2019-12-19   pid-1336
 2020-01-14   pid-1336
 2020-09-23   pid-1336

  6808 rows (40 shown)
```

The interesting part about the command we just executed is that we used a SQL command to query a raw file as we would a table. DuckDB is quite flexible like that.

Let's quickly load the checkup visit csv file into DuckDB and let DuckDB automatically infer the schema with the `read_csv_auto()` command:

```
CREATE TABLE patient_visits AS SELECT * FROM read_csv_auto ('./patient_checkup_logs.csv');
```

```
select * from patient_visits;

    Date      PatientID
    date       varchar

 2016-09-05   pid-001
 2015-08-17   pid-002
 2013-07-18   pid-003
 2015-09-22   pid-003
 2015-10-13   pid-003
     .           .
     .           .
     .           .
 2019-12-11   pid-1338
 2020-02-26   pid-1338
 2020-10-02   pid-1338

  6808 rows (40 shown)
```

We now have a table loaded into our DuckDB database, as we can see when we type the following `describe` command:

```
describe table patient_visits;
```

```
 column_name   column_type    null       key     default    extra
   varchar       varchar     varchar   varchar   varchar   varchar

 Date          DATE          YES
 PatientID     VARCHAR       YES
```

Now that we have our raw data loaded into DuckDB, we can start building our DBT pipeline.

Building Our First DBT Pipeline for Patient Metrics

We need to do 3 things before we can run our DBT pipeline:

1. Configure our DBT Profile (`profiles.yml`) so that the dbt-duckdb connector can talk to our duckdb database
2. Create an initial DBT project to hold our DBT pipeline SQL code
3. Write our specific SQL data modeling code in the DBT project

Once we have these 3 things done, we can run our project and have it produce the

materialized tables inside duckdb that hold the Patient checkup data metrics.

Let's dig into how to configure our DBT profile next.

Configure DBT Profile to Connect to DuckDB Database

To configure DBT to connect to a database, you need to create a configuration file called `profiles.yml`. This file should contain the necessary connection information for each database you want to connect to.

On OSX for example, the DBT profile.yml file lives at:

`~/.dbt/profiles.yml`

Let's now edit our `profiles.yml`

Configure the Profile for DuckDB

In the code listing below, we can see some sample contents of a type `profiles.yml` file:

```
dbt_duckdb_test:
  outputs:
    dev:
      path: /Users/josh/Documents/PattersonConsulting/workspaces/snowpark_demos/duckdb/test_
      type: duckdb
  target: dev

dbt_duckdb_patient_visits:
  outputs:
   dev:
     type: duckdb
     path: /Users/josh/Documents/PattersonConsulting/workspaces/dbt_demos/dbt_hello_world/pa
  target: dev
```

The connection we'll use for this demo is `dbt_duckdb_patient_visits` and we'll reference that in our dbt project in a moment. Let's now create a new DBT project.

Initialize DBT Project

The top level of a dbt workflow is the project. There is a dbt_project.yml file in the main project directory.

The project file tells dbt the project context, and the defined models let dbt know how to build a specific data set.

We can create a project with the `dbt init [name]` as shown below:

```
dbt init [project_name]
```

We'll use a project name of `dbt_hello_world`, and in the directory listing below we can see what was generated inside the directory (`./dbt_hello_world`) created:

```
ls -la

total 24
drwxr-xr-x  12 josh  staff   384 Jul 28 16:04 .
drwxr-xr-x   7 josh  staff   224 Jul 28 16:02 ..
-rw-r--r--   1 josh  staff    29 Jul 27 15:00 .gitignore
-rw-r--r--   1 josh  staff   571 Jul 27 15:00 README.md
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 analyses
-rw-r--r--   1 josh  staff  1349 Jul 28 16:02 dbt_project.yml
drwxr-xr-x   3 josh  staff    96 Jul 28 16:04 logs
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 macros
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 models
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 seeds
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 snapshots
drwxr-xr-x   3 josh  staff    96 Jul 27 15:00 tests
```

As DBT themselves state:

> A dbt project, at its core, is just a folder structure for organizing your individual SQL models. Within the `/models/` folder of a project, any .sql files you publish will be materialized as tables or views to your data warehouse.

In your DBT project, you can now reference the profile name (`dbt_duckdb_patient_visits`) in your `dbt_project.yml` file. In our example you can reference it like this:

```
# dbt_project.yml

...
profile: dbt_duckdb_patient_visits
...
```

Now let's move on to testing our dbt-duckdb connection.

Confirm DBT Profile Connection Works

Try the following command:

```
dbt debug
```

we should see:

```
18:45:48  Running with dbt=1.4.5
dbt version: 1.4.5
python version: 3.9.2
python path: /usr/local/opt/python@3.9/bin/python3.9
```

```
os info: macOS-10.16-x86_64-i386-64bit
Using profiles.yml file at /Users/josh/.dbt/profiles.yml
Using dbt_project.yml file at /Users/josh/Documents/PattersonConsulting/workspaces/dbt_demos

Configuration:
  profiles.yml file [OK found and valid]
  dbt_project.yml file [OK found and valid]

Required dependencies:
 - git [OK found]

Connection:
  database: dbt_patient_visits
  schema: main
  path: /tmp/dbt_patient_visits.duckdb
  Connection test: [OK connection ok]

All checks passed!
```

If your output looks similar to above, then your `dbt-duckdb` connector was configured correctly and DBT can communicate with DuckDB. Now let's work on writing our first DBT data model.

Write Our First DBT Data Model

In dbt (Data Build Tool), a data model is a logical representation of a specific type of data that you want to analyze or work with in your database. It describes the structure, relationships, and constraints of the data in a way that can be easily understood by both humans and computers.

A data model in dbt is typically defined as a SQL query that defines the relationships between tables or other data sources. It specifies how data should be transformed and aggregated to create a particular view of the data. This view can then be used as a source for further analysis or reporting.

In dbt, a data model is created using a "model" statement in a SQL file. This statement defines the columns of the model, any relationships with other models or tables, and any transformations that should be applied to the data. Once defined, a data model can be used as a building block for creating more complex data structures and analyses.

Data modeling in DBT is meant to replace a large, monolithic SQL file with a DAG of smaller SQL operations that can be used by group of people and orchestrated properly.

https://www.getdbt.com/analytics-engineering/modular-data-modeling-technique/

Moving to this method over the monolithic SQL file style allows us to standardize

on transforms for analytics between people or teams and bring consistency to the analytical data output. This allows producers or consumers of data models to start from the foundational data modeling work others have already done.

Some Quick Notes on Data Modeling Naming Conventions

The data modeling naming conventions in DBT can take some work to figure out, especially depending on what part of the data universe you come from (In another converation with JD Long, he off-handed commented: "all of DBT would make more sene to me if they just used terms like *data flow with cached intermediate data* or something"). Yet, I digress.

Data modeling conventions vary, but 3 common ideas are:

1. Sources: raw tables in the database
2. Staging models: clean up and standardize the raw data coming from the warehouse
3. Intermediate models: where we start applying more complex transformations

In dbt, a staging model is typically used to extract and load data from source systems into a data warehouse or other target system. Staging models often involve basic data transformations, such as renaming columns, filtering rows, or casting data types, but they are primarily focused on moving data from the source to the target.

On the other hand, an intermediate model in dbt is typically used for more complex data transformations that are necessary to create analytical models or other downstream data models. Intermediate models often involve complex SQL queries, joins, and other operations that combine data from multiple sources, perform calculations, or reshape data structures.

In general, the flow of data through a dbt project might involve the following steps: extract data from source systems using staging models, transform the data into a format suitable for analysis using intermediate models, and finally, load the transformed data into analytical models that provide insights and value to end-users.

We can also refer to the raw data as a "source" model, as well, that will feed into the a "staging" model.

With all of that in mind, let's take a quick look at the generated `dbt_project.yml` file for our dbt project.

Configuring the Models to Materialize in 'dbt_project.yml'

In dbt (Data Build Tool), the `dbt_project.yml` file is a configuration file that defines the settings and parameters for a dbt project. This file is typically located in the root directory of a dbt project, and it controls how dbt operates and interacts with your data sources.

In the code listing below we see the `dbt_project.yml` file for our hello world project:

```
# Name your project! Project names should contain only lowercase characters
# and underscores. A good package name should reflect your organization's
# name or the intended use of these models
name: 'dbt_hello_world'
version: '1.0.0'
config-version: 2

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

# These configurations specify where dbt should look for different types of files.
# The `model-paths` config, for example, states that models in this project can be
# found in the "models/" directory. You probably won't need to change these!
model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

target-path: "target"  # directory which will store compiled SQL files
clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_packages"


# Configuring models
# Full documentation: https://docs.getdbt.com/docs/configuring-models

# In this example config, we tell dbt to build all models in the example/
# directory as views. These settings can be overridden in the individual model
# files using the `{{ config(...) }}` macro.
models:
  dbt_hello_world:
    # Config indicated by + and applies to all files under models/example/
    patient_visits_models:
      +materialized: view
```

where:

- `dbt_hello_world`: this is the name of the project
- `patient_visits_models`: maps to a subdirectory named 'patient_visits_models' under the 'models' subdirectory (e.g., "dbt_hello_world/models/patient_visits_mo

and inside the `patient_visits_models/` subdirectory, there should be at least:

- `schema.yml`: contains the names of the models represented by .sql files in the same subdirectory
- 1 or more .sql files referenced in `schema.yml`

Files with a .sql extension contain SQL code that defines tables, views, and other database objects. These files are used to define the data models that make up a dbt project.

Schema.yml

In dbt (Data Build Tool), the schema.yml file is used to define the structure and constraints of tables and columns in a database schema. This file is typically located in the same directory as the SQL file that defines a particular table or model, and it provides a way to document and validate the schema of your data.

In the code listing below, we see the file `schema.yml`'s contents:

```
version: 2

sources:
  - name: patient_visits
    schema: main
    description: "The main patient visits raw data table"
    tables:
      - name: patient_visits
        columns:
          - name: PatientID
          - name: Date

models:
  - name: patient_visits_summed_model
    description: "A starter duckdb dbt model"
    columns:
      - name: PatientID
        description: "The primary key for this table"
        tests:
          - unique
          - not_null
      - name: visits
        description: "the summed visits across time for this patient"
  - name: patient_visits_summed_over_5
    description: "Only Patients who made at least 5 visits"
    columns:
      - name: PatientID
```

```
        description: "The primary key for this table"
        tests:
          - unique
          - not_null
      - name: visits
        description: "the summed visits across time for this patient"
```

We can see 2 models listed ("patient_visits_summed_model" and "patient_visits_summed_over_5"), along with a "source"; This source represents the raw tables in our database, DuckDB, and we are listing our table `patient_visits` as a source to work with in our DBT workflow. We'll see that used in a DBT model in a moment.

Overall, the schema.yml file provides a way to define and document the schema of your data in a structured and consistent way, making it easier to understand, maintain, and validate your data over time.

Now let's take a look at the first model, `patient_visits_summed_model`.

patient_visits_summed_model.sql

In the code listing below, we see the contents of the data model file `patient_visits_summed_model.sql`:

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

with patient_data as (

    select count(Date) as visits, PatientID from {{source('patient_visits', 'patient_visits'

)

select *
from patient_data
```

We can see it is to be materialized as a table in target database (DuckDB), and we're reading data `from {{source('patient_visits', 'patient_visits')}}`. Since this is a simple Hello World example, we're going ahead and building the aggregation in this "intermediate" model with a `group by` and `count()` operation. The results of this data modeling operation in our DBT workflow give us a list of all patients with how many times total they got a checkup. This intermediate aggregation can be used by other data models in the rest of our DBT workflow, as we'll see next.

patient_visits_summed_over_5.sql

Now that we have an aggregation of the total checkups for each patient as a data model, we'll create another data model that references the previuos intermediate stage with a new data model that shows only Patients that had more than 5 visits, as seen in the code listing below:

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

with patient_data_summed as (

    select * from {{ref('patient_visits_summed_model')}} where visits >= 5

)

select *
from patient_data_summed
```

if you'll notice in the second model (patient_visits_summed_over_5.sql), we are referring to the first model (patient_visits_summed_model.sql) not by file name — but by logical data model name, `{{ref('patient_visits_summed_model')}}`.

You'll notice the logical name of the data model is the filename without the .sql extension.

We refer to other logical data models in DBT with the `ref` function, as explained in the dbt docs:

https://docs.getdbt.com/reference/dbt-jinja-functions/ref

> The most important function in dbt is ref(); it's impossible to build even moderately complex models without it. ref() is how you reference one model within another. This is a very common behavior, as typically models are built to be "stacked" on top of one another.

These models are stacked on top of one another, linked with the ref() function, creating a DAG that can be executed on different type of backend data warehouse systems (e.g., cloud, MPP, embedded, etc).

Now, let's run our full DBT workflow DAG locally.

Run DBT Pipeline Locally

Let's now crank up our DBT workflow.

It's worth noting that you are running a model that will transform your data without that data ever leaving your warehouse when you run a DBT workflow.

We can run our current DBT Hello World project by running the command below inside the main directory of our project:

```
dbt run
```

and we should see:

```
19:44:11  Running with dbt=1.4.5
19:44:11  Found 2 models, 4 tests, 0 snapshots, 0 analyses, 296 macros, 0 operations, 0 seed
19:44:11
19:44:11  Concurrency: 1 threads (target='dev')
19:44:11
19:44:11  1 of 2 START sql table model main.patient_visits_summed_model .................. [
19:44:11  1 of 2 OK created sql table model main.patient_visits_summed_model ............. [
19:44:11  2 of 2 START sql table model main.patient_visits_summed_over_5 ................. [
19:44:11  2 of 2 OK created sql table model main.patient_visits_summed_over_5 ............ [
19:44:11
19:44:11  Finished running 2 table models in 0 hours 0 minutes and 0.20 seconds (0.20s).
19:44:11
19:44:11  Completed successfully
19:44:11
19:44:11  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

We can confirm that DBT read from 1 source and then created 2 models, as we expected. Now let's confirm that is what happened inside the DuckDB database locally.

Check DuckDB Materialized Tables

If we open our previous DuckDB database up with the CLI command:

```
./duckdb dbt_patient_visits.duckdb
```

We can then check out the models materialed with the SQL:

```
select * from patient_visits_summed_over_5;
```

visits	PatientID
int64	varchar
5	pid-003
7	pid-004
7	pid-005
8	pid-006
8	pid-008
5	pid-009
5	pid-010
7	pid-012
6	pid-013
5	pid-015
10	pid-016
8	pid-017
6	pid-018

```
 6  pid-019
10  pid-020
 5  pid-021
 5  pid-022
10  pid-023
 8  pid-025
10  pid-027
 .     .
 .     .
 .     .
 7  pid-1271
 7  pid-254
 8  pid-795
 7  pid-1126
 5  pid-940
 7  pid-986
 8  pid-1135
 6  pid-1141
10  pid-1208
 7  pid-1235
10  pid-1302
10  pid-556
10  pid-695
 9  pid-1123
 6  pid-1332
 9  pid-617
 6  pid-1112
10  pid-1308
 8  pid-979
 6  pid-889

 758 rows
(40 shown)
```

Confirming that DBT did run correctly.

Summary

In this blog post we gave you an overview of DBT and then showed how to build a simple "Hello World"-style of application. For more information on DBT, check out their documentation. For more information on cloud infrastructure, check out the rest of the articles in our blog.

We also offer private workshops for companies on topics such as creating and running DBT pipelines (on multiple platforms such as Snowflake and Fivetran),

please feel free to reach out if you'd like to discuss attending one of our workshops.