

A. DBT Use Cases

1. Data Warehousing & Modeling

Transforms raw data into structured models for analysis, ML, or querying. This supports ETL pipelines, and does the incremental processing (focussing only on changed data). Helps prepare structured training data, supports feature engineering. And also supports micro batch transformations for fast changing data. I can schedule runs as well.

2. Analytics & Reporting

Helps us to prepare clean and reusable data for BI tools. Prepare models using dbt and then connect that database to the BI tools.

3. Data Quality

Runs automated tests and tracks the data quality and we can connect with the observability platform.

4. Governance & Documentation

Documents models, columns, and owners. Tracks dependencies as well.

5. Workflow & Automation

Version Control & CI/CD

B. Python for DBT

Dbt introduced python models in extension of the SQL models for the transformation. Python allows models to write transformations using pandas and other libraries.

As of now, dbt supports Python models on five platforms

1. amazon anthena
2. big query
3. Databricks
4. DuckDB
5. SnowFlake

Now, I am choosing DuckDBT. I chose DuckDB as my database for this demonstration of the python models in dbt because it's lightweight, easy to set up, and runs entirely locally without

any cloud infrastructure. DBT runs according to the dependencies and it does not prioritise SQL or python.

C. Setup

```
python -m pip install dbt-core dbt-duckdb
```

D. Exploration

1. Data Warehousing Use Case

Step 1: Initialize Your Project and configuration

Use these commands in the cmd to initialise the project

```
dbt init dw_project  
cd dw_project
```

During setup make sure to use duckdb as the adapter as we are using that as our data platform. Configure the dbt profile for this project in the profiles.yml file to make sure that we are using DuckDB as our database.

```
C:\Users\megha\Documents\ResearchWork\DBTProjects\DWUseCase>dbt init dw_project
18:50:55  Running with dbt=1.8.9
18:50:55
Your new dbt project "dw_project" was created!

For more information on how to configure the profiles.yml file,
please consult the dbt documentation here:

  https://docs.getdbt.com/docs/configure-your-profile

One more thing:

Need help? Don't hesitate to reach out to us via GitHub issues or on Slack:

  https://community.getdbt.com/

Happy modeling!

18:50:55  Setting up your profile.
Which database would you like to use?
[1] duckdb
[2] fabric
[3] sqlserver

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

Enter a number: cd dw_project
Error: 'cd dw_project' is not a valid integer.
Which database would you like to use?
[1] duckdb
[2] fabric
[3] sqlserver

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

Enter a number: 1

C:\Users\megha\Documents\ResearchWork\DBTProjects\DWUseCase>cd dw_project
```

```

! profiles.yml X
C: > Users > megha > .dbt > ! profiles.yml
14 sample_dbt_project:
15   outputs:
16     dev:
23       trust_cert: true
24       type: sqlserver
25       windows_login: true
26     target: dev
27
28   dw_project:
29     outputs:
30       dev:
31         type: duckdb
32         path: dw_project.duckdb
33         threads: 1
34         extensions:
35           - parquet
36           - httpfs
37
38       prod:
39         type: duckdb
40         path: prod.duckdb
41         threads: 4
42
43     target: dev
44
45

```

Step 2: Add seed data

Add your datafiles in the seeds folder. And then run the **dbt seed** command in the terminal. In SQL Server, the data already exists in the database, so you don't need dbt seed. But in DuckDB, which starts from local files, dbt seed is used to load CSVs as tables so dbt can reference them in models. It's essential for initializing your data in a file based setup.

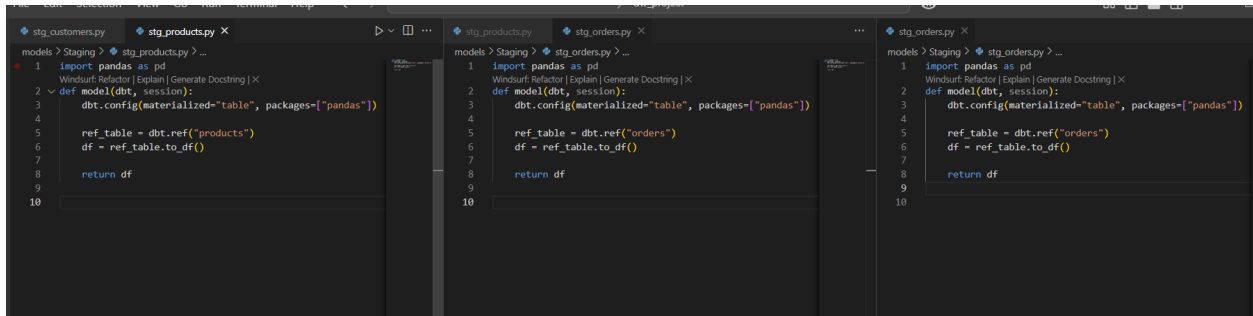
```

C:\Users\megha\Documents\ResearchWork\DBTProjects\DWUseCase\dw_project>dbt seed
18:57:00 Running with dbt=1.8.9
18:57:01 Registered adapter: duckdb=1.9.3
18:57:01 Unable to do partial parsing because saved manifest not found. Starting full parse.
18:57:03 Found 2 models, 3 seeds, 4 data tests, 428 macros
18:57:03
18:57:04 Concurrency: 1 threads (target='dev')
18:57:04
18:57:04 1 of 3 START seed file main.customers ..... [RUN]
18:57:04 1 of 3 OK loaded seed file main.customers ..... [INSERT 3 in 0.12s]
18:57:04 2 of 3 START seed file main.orders ..... [RUN]
18:57:04 2 of 3 OK loaded seed file main.orders ..... [INSERT 4 in 0.03s]
18:57:04 3 of 3 START seed file main.products ..... [RUN]
18:57:04 3 of 3 OK loaded seed file main.products ..... [INSERT 3 in 0.03s]
18:57:04
18:57:04 Finished running 3 seeds in 0 hours 0 minutes and 0.56 seconds (0.56s).
18:57:04
18:57:04 Completed successfully
18:57:04
18:57:04 Done. PASS=3 WARN=0 ERROR=0 SKIP=0 TOTAL=3

```

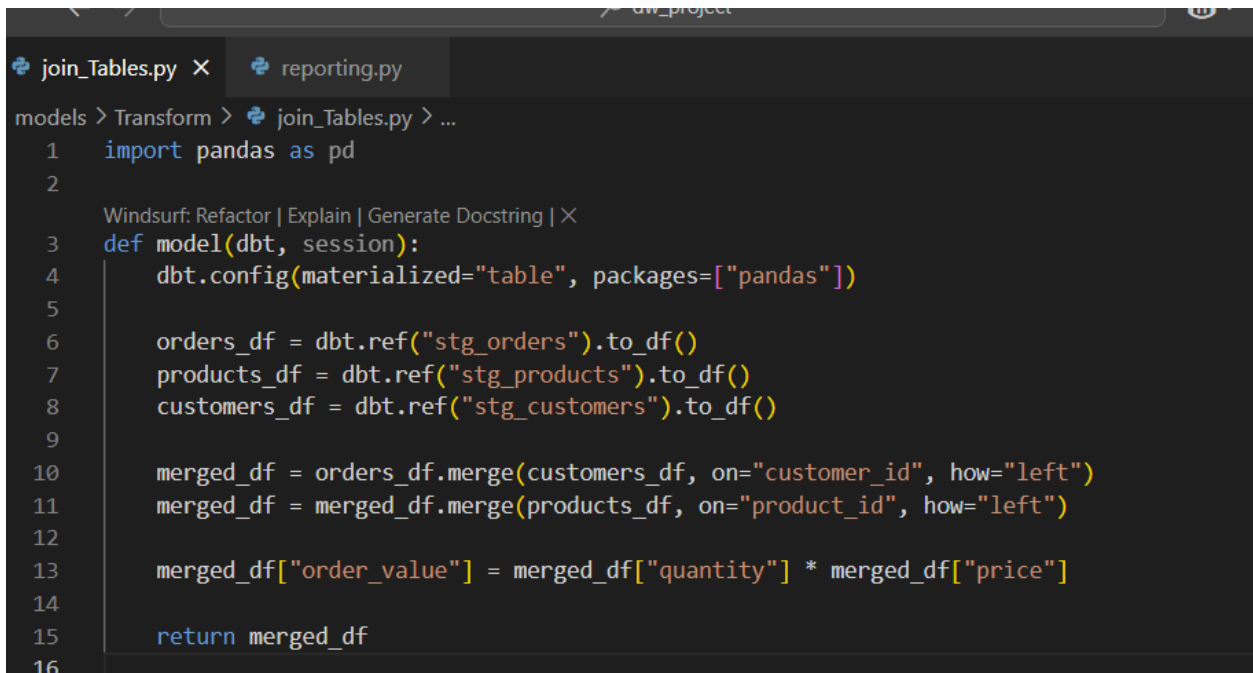
Step 3: Create python staging models

These will cleanly load each table into DuckDB as a dbt-managed table. Staging models are mainly used to load raw data into dbt using `ref()` and return it as DataFrames. They are useful for the future transformations and help us in modularizing the project.

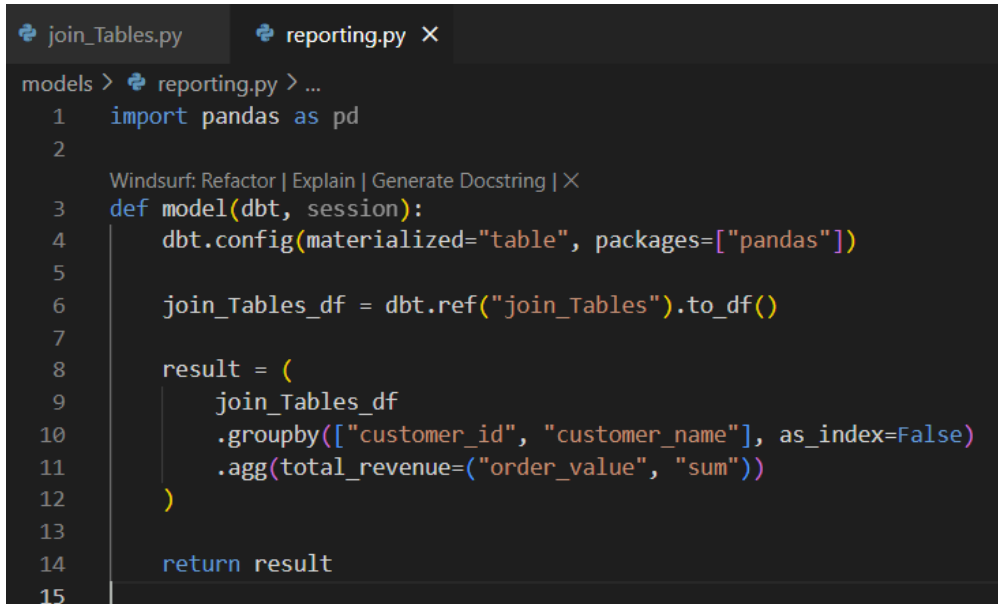


In dbt Python models with DuckDB, `dbt.ref("table_name")` returns a DuckDB relation object, not just a table name. This is different from the previous Microsoft SQL server as the database because that returns a table name string. This object supports `.to_df()`, allowing direct conversion to a pandas DataFrame. It works because DuckDB executes Python models locally in the same process.

Step 4: Applying some transformations



Here we are simply merging all the tables on keys and then creating a custom column named `order_value`. We can see, once it's converted into a pandas df. We can simply do whatever we want. Just like we can do it in a python file or else in a jupyter notebook.



```
models > reporting.py > ...
1  import pandas as pd
2
3  def model(dbt, session):
4      dbt.config(materialized="table", packages=["pandas"])
5
6      join_Tables_df = dbt.ref("join_Tables").to_df()
7
8      result = (
9          join_Tables_df
10         .groupby(["customer_id", "customer_name"], as_index=False)
11         .agg(total_revenue=("order_value", "sum"))
12     )
13
14     return result
15
```

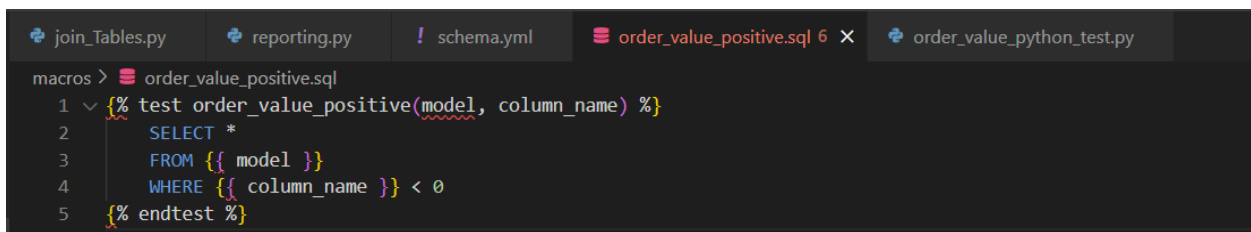
This file uses the previously joined table and aggregates the values of the `order_value`.

Step 5: Custom Tests

In dbt, you can define custom SQL tests using macros by writing Jinja-based SQL logic and referencing them in `schema.yml`, making the test reusable across multiple models or directly callable from the tests folder. Python-based custom tests can also be created, but dbt Core does not support them natively. To run Python tests within dbt, you must use a third-party package like Elementary, which enables Python test execution by embedding Python code inside SQL macros. Note that macros themselves must be written in SQL using Jinja and Python cannot be used to define macros.

1. SQL Macro Custom Test
2. Python Test

This is the SQL Macro Test



```
macros > order_value_positive.sql
1  {% test order_value_positive(model, column_name) %}
2      SELECT *
3      FROM {{ model }}
4      WHERE {{ column_name }} < 0
5  {% endtest %}
6
```

```

- name: join_tables
  description: "Joined data from
  columns:
    - name: order_id
      tests:
        - not_null
        - unique

    - name: customer_id
      tests:
        - not_null

    - name: product_id
      tests:
        - not_null

    - name: order_value
      tests:
        - not_null
        - order_value_positive

```

Here We are writing a macro in SQL to check if the order value is positive. As macro is created we have to add that in the schema.yml file as well to perform this test.

To Write Custom tests in Python, we have to first install elementary dbt test packages. We have to create a package.yml file in the parent directory and include elementary in there.

```

! packages.yml
1 packages:
2   - package: elementary-data/elementary
3     version: 0.18.3
4

```

And then run dbt deps to install this.

This enables us to write tests in python.

Test: Compare two models and find mismatches in key columns.

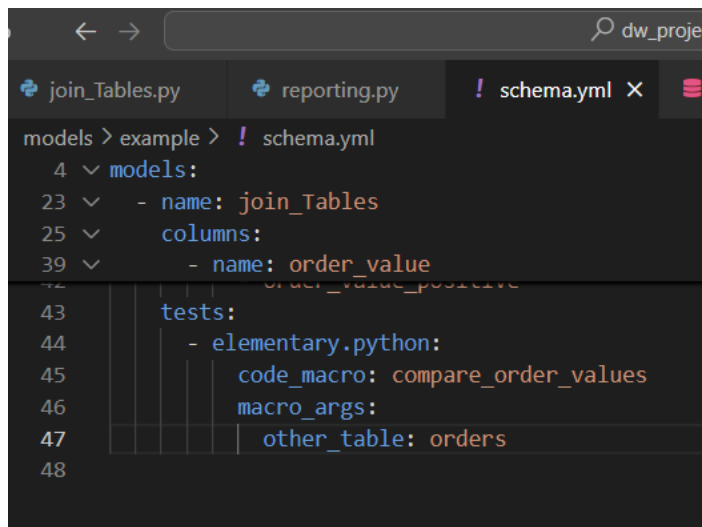
```

EXPLORER
DW_PROJECT
  > analyses
  > dbt_packages
  > logs
  > macros
    .gitkeep
    compare_order_values.sql 9+
    order_value_positive.sql 6
  > models
    example
      my_first_dbt_model.sql
      my_second_dbt_model.sql
      schema.yml

macros > compare_order_values.sql
1 {% macro compare_order_values(args) %}
2   import pandas as pd
3
4   def test(model_df, ref, session):
5     raw_df = ref('{{ args.other_table }}').toPandas()
6
7     merged = pd.merge(model_df.toPandas(), raw_df, on="order_id", suffixes=('_actual', '_expected'))
8
9     mismatches = merged[merged["order value actual"] != merged["order value expected"]]
10
11     return mismatches
12   {% endmacro %}
13

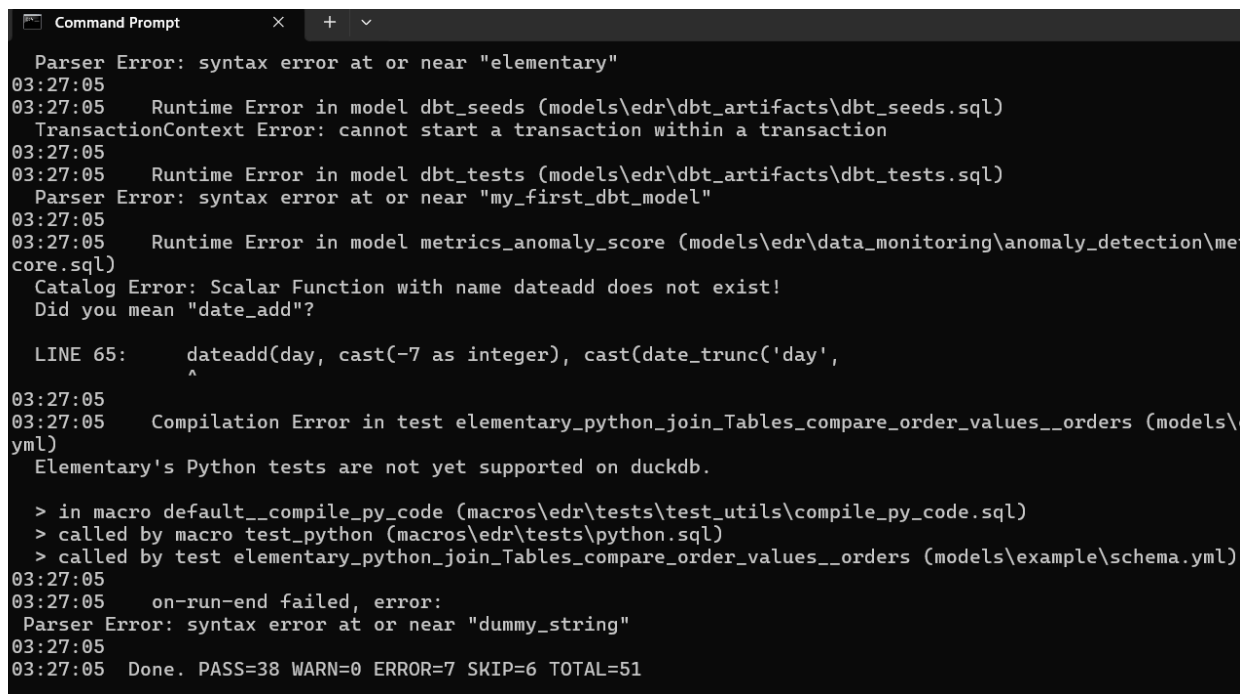
```

Add this in the schema.yml
Under elementary as shown below



```
models > example > ! schema.yml
4  ∨ models:
23 ∨   - name: join_Tables
25 ∨     columns:
39 ∨       - name: order_value
43       tests:
44         - elementary.python:
45             code_macro: compare_order_values
46             macro_args:
47               other_table: orders
48
```

Now try running dbt build, dbt run and dbt test on the cmd.
I ran dbt build.



```
Command Prompt
Parser Error: syntax error at or near "elementary"
03:27:05
03:27:05 Runtime Error in model dbt_seeds (models\edr\dbt_artifacts\dbt_seeds.sql)
TransactionContext Error: cannot start a transaction within a transaction
03:27:05
03:27:05 Runtime Error in model dbt_tests (models\edr\dbt_artifacts\dbt_tests.sql)
Parser Error: syntax error at or near "my_first_dbt_model"
03:27:05
03:27:05 Runtime Error in model metrics_anomaly_score (models\edr\data_monitoring\anomaly_detection\me
core.sql)
Catalog Error: Scalar Function with name dateadd does not exist!
Did you mean "date_add"?

LINE 65:      dateadd(day, cast(-7 as integer), cast(date_trunc('day',
^
03:27:05
03:27:05 Compilation Error in test elementary_python_join_Tables_compare_order_values__orders (models\
.yml)
Elementary's Python tests are not yet supported on duckdb.

> in macro default__compile_py_code (macros\edr\tests\test_utils\compile_py_code.sql)
> called by macro test_python (macros\edr\tests\python.sql)
> called by test elementary_python_join_Tables_compare_order_values__orders (models\example\schema.yml)
03:27:05
03:27:05 on-run-end failed, error:
Parser Error: syntax error at or near "dummy_string"
03:27:05
03:27:05 Done. PASS=38 WARN=0 ERROR=7 SKIP=6 TOTAL=51
```

Here I am using duckdb. Elementary tests are not supported on duckdb yet. Maybe we can try using it on the other database platforms like snowflake or databricks.

2. To Integrate Jupyter Notebook with DBT

I tried a library called nbdbt but it's a failure as it has not been updated for 3 years and dbt has been continuously evolving. It no longer exists in newer versions of dbt-core, hence does not support python models.

So, instead of this I am trying to directly integrate dbt with commands in the notebook.

As we know, we can run cmd commands using ! in the beginning of that command.

```
code = '''
import pandas as pd

def model(dbt, session):
    dbt.config(materialized="table", packages=["pandas"])

    sales_df = dbt.ref("sales_data").to_df()

    result = (
        sales_df
        .groupby("customer", as_index=False)["amount"]
        .sum()
        .rename(columns={"amount": "total_amount"})
    )

    return result

'''

# Save this as a dbt Python model
with open("models/total_sales.py", "w") as f:
    f.write(code)
```

In this way we can create a python model file and then run using ! in the beginning as shown below.

```
> ~
[10] !dbt run --select total_sales

...
04:13:21 Running with dbt=1.9.6
04:13:22 Registered adapter: duckdb=1.9.3
04:13:24 Found 3 models, 1 seed, 4 data tests, 428 macros
04:13:24
04:13:24 Concurrency: 1 threads (target='dev')
04:13:24
04:13:28 1 of 1 START python table model main.total_sales ..... [RUN]
04:13:32 1 of 1 OK created python table model main.total_sales ..... [OK in 3.61s]
04:13:32
04:13:32 Finished running 1 table model in 0 hours 0 minutes and 7.39 seconds (7.39s).
04:13:32
04:13:32 Completed successfully
04:13:32
04:13:32 Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```


Similarly we can create tests and run those as well in the notebook.

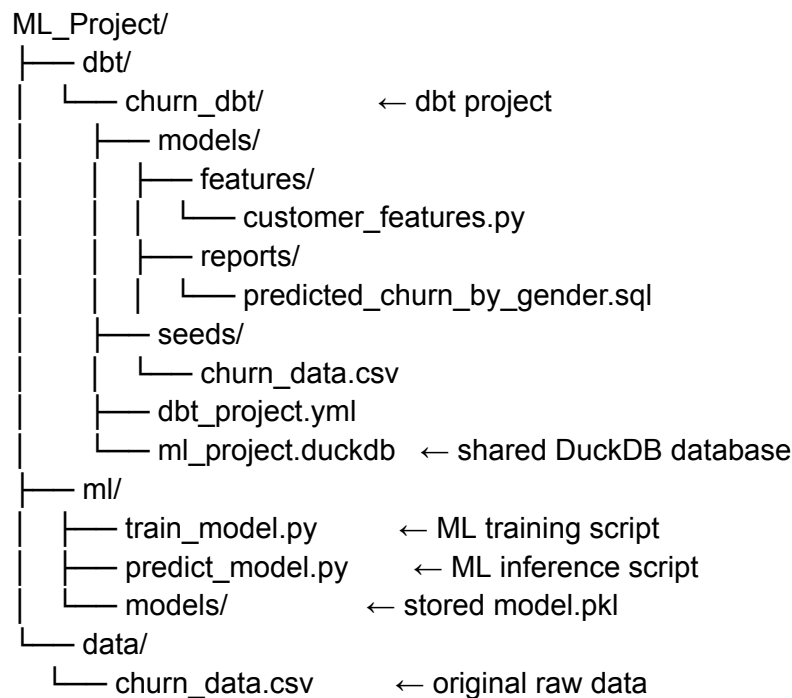
Here we can extend and connect the database. Then run ML models as required.

3. DBT for ML Project

Step 1: Initialize Your Project and folder, project setup

Created a project folder to separate all the dbt and ML logic, while connecting both via DuckDB. Ran dbt seed to ingest the data into the duckdb database.

This enables us a shared, local analytical database for both dbt and ML workflows while maintaining a clean separation of these workflows.



Step 2: Feature Engineering using dbt Python Model

Created a python model inside features folder of the dbt project. Wrote a few transformation logic using pandas inside the dbt python model like column renaming.

Ran dbt run --select --customer_features.py

A new clean feature set table is stored in the database now for the ML usage.

Step 3: Training the Machine Learning Model

Created a python to train in the ml folder. Connected to the database and then used this new cleaned and featured table. Training done using logistic regression.

This allowed consistent feature reuse across training and inference. This made model training easily repeatable with minimal dependencies just by running a python script.

Step 4: Storing Predictions to duckdb

Created a predict_model.py in the ml folder. Loaded the saved model and generated predictions and wrote them back to the database as the new table.

We closed the ML loop by bringing the predictions into the shared database.

Step 5: dbt Models on ML Predictions

Created a SQL model inside the new folder in the models/ reports/ . In this model, we can query the new predictions table and make transformations as we wish.

This enabled us to integrate the ML outputs into the dbt's layer through a connected database.

4. DBT Mesh

dbt Mesh enables large organizations to scale their data transformation workflows by allowing multiple dbt projects to work together in a governed, modular way. It supports cross-project model referencing, domain-specific ownership, and strict access control. This makes it ideal for enterprises with complex team structures and growing governance needs.

5. Connecting with PowerBI or any visualisation tool

You can connect the database to the visualisation tool and continue. As we update that database, we can see the same in the visualization as well.

6. DBT fal

dbt-fal used to let you run Python models locally with Pandas. It's super handy for quick prototyping without needing cloud setup and is compatible with any dbt adapter. However, dbt-fal is now archived and no longer maintained, and is not recommended for use in production environments.

7. Ideal configuration details

Ideal configuration matters only if we are using the DBT core version(Local Development). If I am using dbt-cloud, then all of this is taken care of by dbt labs.

Ideal configuration of DBT:

OS: as dbt is a python based tool, it works on all environments. Linux is preferred Especially if we are using a server based environment. For example, if we are orchestrating data from real time pipelines.

Docker vs VM:

VMs provide strong isolation but are heavy and slow. Use them only if you need full OS control or strict security. Docker is lightweight, fast, and ideal for consistent environments, especially useful for development, CI/CD, and cloud deployments.

Most Feature-Rich and Databases:

Snowflake, BigQuery, and Redshift are the most feature-rich and reliable adapters, with direct support from dbt Labs and deep integration with database features. Other data warehouses are supported but may lack advanced capabilities or even writing models or tests in Python.

Extra things to explore:

<https://freedium.cfd/https://medium.com/data-science/streamline-dbt-model-development-with-notebook-style-workspace-eb156fe6e81>

- Mage Tool

DBT with snowflake