setting up dbt-core on windows :

1. Install python3.7+ version

2. Install git

3. create a venv in python using : "python3 -m venv dbt-env"

4. activate the venv using : "dbt-env\Scripts\activate"

5. to install dbt-core using pip :

pip install dbt-core dbt-postgres dbt-redshift dbt-snowflake dbt-bigquery

dbt CLI:

The target/compiled/ directory for compiled select statements

The target/run/ directory for compiled create statements

The logs/dbt.log file for verbose logging.

profiles.yml:

this file contains all configuration of sources and destnations (i.e.BQ/Snowflake/Redshift/postgres)

this file is located at C:/Users/<your\_username>/.dbt/profiels.yml

Seed are useful for loading country codes, employee emails, or employee account IDs

dbt Commands

|  |  |
| --- | --- |
| **Command** | **Description** |
| docs | Generate or serve the documentation website for your project. |
| source | Manage your project's sources |
| init | Initialize a new DBT project. |
| clean | Delete all folders in the clean-targets list (usually the dbt\_packages and target directories.) |
| debug | Show some helpful information about dbt for debugging. Not to be confused with the --debug option which increases verbosity. |
| deps | Pull the most recent version of the dependencies listed in packages.yml |
| List (ls) | List the resources in your project |
| build | Run all Seeds, Models, Snapshots, and tests in DAG order |
| snapshot | Execute snapshots defined in your project |
| run | Compile SQL and execute against the current target database. |
| compile | Generates executable SQL from source, model, test, and analysis files. Compiled SQL files are written to the target/ directory. |
| parse | Parses the project and provides information on performance |
| test | Runs tests on data in deployed models. Run this after `dbt run` |
| seed | Load data from csv files into your data warehouse. |
| run-operation | Run the named macro with any supplied arguments. |

DBT Job steps :

Clone Git Repository with deployment code

Create Profile from Connection BigQuery

Invoke dbt deps :Pull the most recent version of the dependencies listed in packages.yml

Invoke dbt source snapshot-freshness : Checks the freshness of source tables without breaking the job

Invoke dbt build : Run all Seeds, Models, Snapshots, and tests in DAG order

Most dbt commands (and corresponding RPC methods) produce artifacts:

**manifest**: produced by build, compile, run, test, docs generate, ls

**run results**: produced by build, run, test, seed, snapshot, docs generate

**catalog**: produced by docs generate

**sources**: produced by source freshness

Sequence of dbt command execution :

1. Dbt debug
2. Dbt seed
3. Dbt run
4. Dbt test
5. Dbt docs generate
6. Dbt docs serve
7. Dbt build

To view dbt docs from dbt core , we use command “ dbt docs serve”

“””””””

**When we change materialization of a model from table to view**

**Or view to table**

**We need to add --full-refresh after the build command**

**That is : dbt build –select <model\_name> --full-refresh**

**This will drop existing object(either table or view ) with same name and recreate it with given configuration**

“””””””

## DBT Materializations

**Tables**

* Built as tables in the database
* Data is stored on disk
* Slower to build
* Faster to query
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='table'

)}}

**Views**

* Built as views in the database
* Query is stored on disk
* Faster to build
* Slower to query
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='view'

)}}

**Ephemeral Models**

* Does not exist in the database
* Imported as CTE into downstream models
* Increases build time of downstream models
* Cannot query directly
* [Ephemeral Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/materializations#ephemeral)
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='ephemeral'

)}}

**Incremental Models**

* Built as table in the database
* On the first run, builds entire table
* On subsequent runs, only appends new records\*
* Faster to build because you are only adding new records
* Does not capture 100% of the data all the time
* [Incremental Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/materializations#incremental)
* [Discourse post on Incrementality](https://discourse.getdbt.com/t/on-the-limits-of-incrementality/303)
* Configuration is more advanced in this case. Consult the dbt documentation for building your first incremental model.

**Snapshots**

* Built as a table in the database, usually in a dedicated schema.
* On the first run, builds entire table and adds four columns: dbt\_scd\_id, dbt\_updated\_at, dbt\_valid\_from, and dbt\_valid\_to
* In future runs, dbt will scan the underlying data and append new records based on the configuration that is made.
* This allows you to capture historical data
* [Snapshots Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/snapshots)
* Configuration is more advanced in this case. Consult the dbt documentation for writing your first snapshot.

**DBT Model organization**

|  |  |
| --- | --- |
| **Category** | **Description** |
| Staging | Contains models which clean and standardize data |
| Marts | Contains models which combine or heavily transform data |

Dbt styles guidelines: <https://github.com/dbt-labs/corp/blob/main/dbt_style_guide.md>

**DBT Tests:**

What makes a good test?

|  |  |
| --- | --- |
| Automated | low effort/repeatable |
| Fast | if testing takes too long, no one will do it |
| Reliable | believe them when they say something does work |
| Informative | leave you clues about what to fix based on the error |
| Focused | every test should validate one assumption |

Test on One Database Object

What/ Why

* Assert something about the data that
* you think is true
* Contents of the data
* Constraints of the table
* The grain of the table
* Compare values in one model to a source of truth in another model
* Ensure data has neither been erroneously added or removed

Example Tests

* Unique
* not\_null
* accepted\_values
* relationships
* Other package's tests like:
  + dbt\_expectations.expect\_column\_proportion\_of\_unique\_values\_to\_be\_between
  + dbt\_utils.equality
  + dbt\_expectations.expect\_table\_row\_count\_to\_equal\_other\_table
* Freshness test : *dbt source freshness*

### When to test

There are 4 major points in time when you should consider testing your code

1. In **development**, it is critical to test your changes to modeling logic while you make changes. This can help individual developers find bugs before opening a pull request.
2. In **production**, it is important to continue testing your code to catch failures when they happen. This can empower the data team to catch data quality issues well before stakeholders are impacted.
3. When **proposing changes / opening a pull or merge request**, we can run automated tests against our proposed changes to catch any issues that may not have been caught in the development cycle mentioned above.
4. On a **middle / qa branch**, it can be helpful to test a batch of changes that have been made in an isolated testing environment before then merging the code to the main / production branch.

### Testing commands

There are several ways to run tests against your data in dbt. In development, it likely will make more sense to write a model and materialize it with `dbt run` and then test your models with `dbt test`.  However, once you are materializing and testing multiple models at once, we highly recommend running `dbt build` to materialize and test all nodes in DAG order.

### Storing test failures in the database

When you are testing your models in production or development, it can be helpful to capture the failing records for future inspection. The `store\_failures` configuration or command line flag allows you to store these failures in the database for further analysis.

### Test configurations

* Similar to models, there are several configurations that you can add to the tests in your dbt project.
  + **severity** allows you to toggle between **warn** and **error**when a test doesn't meet your assertions.
  + **warn\_if**and**error\_if** allow you to set thresholds for warning or errors for a specific test
  + **where**allows you to filter down to a subset of rows that you want to test
  + **limit** allows you to limit the number of returned failing records.
  + **store\_failures** allows you to enable storing of the failing records in your data platform
  + **schema** allows you to specify where you want to store the failing records if you enable store\_failures.
* Test configurations can be applied in various places including:
  + yaml configurations where generic tests are applied
  + config blogs in the top of singular tests
  + dbt\_project.yml to apply configurations to tests across your project in one place.

This is a long list of potential testing commands!

**## run all tests**

*dbt test*

**#run tests for one specific model**

*dbt test --select one specific model*

*dbt test --select customers*

**#run tests for all models in a subfolder (i.e. marts/core)**

*dbt test --select marts.core.\**

**#run tests for all models in package**

*dbt test --select some\_package.\**

--note: we do not currently have any packages installed that do have tests

**## run only tests defined singularly**

*dbt test --select test\_type:singular*

**##run only tests defined generically**

*dbt test --select test\_type:generic*

**##run singular tests limited to one specific model**

*dbt test --select one specific model, test\_type:singular*

*dbt test --select orders, test\_type: singular*

**##run generic tests limited to one specific model**

*dbt test --select one specific model, test\_type:generic*

*dbt test --select customers, test\_type:generic*

**## run only source tests**

*dbt test --select source:\**

[Writing custom generic tests](https://docs.getdbt.com/guides/legacy/writing-custom-generic-tests)

[Test configurations](https://docs.getdbt.com/reference/test-configs#test-specific-configurations)

dbt's node selection syntax makes it possible to run only specific resources in a given invocation of dbt. This selection syntax is used for the following subcommands:

| **command** | **argument(s)** |
| --- | --- |
| [run](https://docs.getdbt.com/reference/commands/run) | --select, --exclude, --selector, --defer |
| [test](https://docs.getdbt.com/reference/commands/test) | --select, --exclude, --selector, --defer |
| [seed](https://docs.getdbt.com/reference/commands/seed) | --select, --exclude, --selector |
| [snapshot](https://docs.getdbt.com/reference/commands/snapshot) | --select, --exclude --selector |
| [ls (list)](https://docs.getdbt.com/reference/commands/list) | --select, --exclude, --selector, --resource-type |
| [compile](https://docs.getdbt.com/reference/commands/compile) | --select, --exclude, --selector |
| [freshness](https://docs.getdbt.com/reference/commands/source) | --select, --exclude, --selector |
| [build](https://docs.getdbt.com/reference/commands/build) | --select, --exclude, --selector, --resource-type, --defer |

Fore more : <https://docs.getdbt.com/reference/node-selection/syntax>

## Some other packages to consider – Python

* [dbt-coverage](https://github.com/slidoapp/dbt-coverage)
  + [Compute](https://github.com/slidoapp/dbt-coverage#compute) coverage from [catalog.json](https://docs.getdbt.com/reference/artifacts/catalog-json) and [manifest.json](https://docs.getdbt.com/reference/artifacts/manifest-json) files found in a dbt project, e.g. jaffle\_shop.
* [pre-commit-dbt](https://github.com/offbi/pre-commit-dbt)
  + A comprehensive list of hooks to ensure the quality of your dbt projects.
  + check-model-has-tests: Check the model has a number of tests.
  + check-source-has-tests-by-name: Check the source has a number of tests by test name.
  + See [Enforcing rules at scale with pre-commit-dbt](https://docs.getdbt.com/blog/enforcing-rules-pre-commit-dbt)

## Some other packages to consider – dbt Packages

* [dbt\_dataquality](https://hub.getdbt.com/divergent-insights/dbt_dataquality/latest/)
  + Access and report on the outputs from dbt source freshness ([sources.json](http://sources.json/) and [manifest.json](https://docs.getdbt.com/reference/artifacts/manifest-json)) and dbt test ([run\_results.json](https://docs.getdbt.com/reference/artifacts/run-results-json) and [manifest.json](https://docs.getdbt.com/reference/artifacts/manifest-json))
  + Optionally tag tests and visualize quality by type
* [dbt-project-evaluator](https://github.com/dbt-labs/dbt-project-evaluator)
  + This package highlights areas of a dbt project that are misaligned with dbt Labs' best practices. Specifically, this package tests for:
  + This package is in its early stages!

**DBT Metrics**

A metric is a timeseries aggregation over a [table](https://docs.getdbt.com/terms/table) that supports zero or more dimensions. Some examples of metrics include:

* active users
* monthly recurring revenue (mrr)