DBT

another look

- Tests
- Macros
- Sources

Sources.yml

A **declarative mapping** of the raw tables in your warehouse that dbt models depend on

Lives in your repo (usually `models/sources.yml` or `models/staging/sources.yml`)

Lets you reference upstream tables with `{{ source('schema_name','table_name') }}` instead of hard-coding fully qualified names

when you reference it in a model, you use **Jinja** to call the `source()` function.

```yaml
version: 2
sources:
  - name: raw
    database: zeta-axiom-468312-f1     # your project
    schema: raw_data                    # your dataset
    tables:
      - name: weather_raw
      - name: excersise_1_customers
      - name: excersise_2_customers
      - name: excersise_1_sessions
      - name: excersise_1_orders
      - name: excersise_1_playlists
```

```
select *
from {{ source('raw', 'your_table_name') }}
```

You have **two kinds** of tests in dbt:

Generic (schema) tests – declared in YAML next to the model

- Put them in a YAML file, typically beside the model:
  models/staging/stg_your_table.yml
- These are things like not_null, unique, accepted_values

Those are **built-in generic tests** provided by dbt. In YAML, you attach them to a column (or model) and dbt compiles them into SQL that **returns any failing rows**. If the query returns ≥1 row, the test fails.

Most-used built-ins (dbt Core):

- not_null – column has no NULLs
- unique – column has no duplicates
- accepted_values – column is in an allowed set

Singular (data) tests – written as raw SQL that returns failing rows

- Put each test as a `.sql` file under `tests/`, e.g.:
  `tests/no_future_dates.sql`

TL;DR: **Keep your YAML schema tests near the model (common practice).** Use the `tests/` folder for one-off SQL checks that don't neatly fit a generic test.

Adding testing (generic)

we create our model: models/staging/stg_your_table.**sql**

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

select
  *
from {{ source('raw', 'YOUR_TABLE_NAME') }}
```

Creating super simple test

create `models/staging/stg_your_table.yml`

```yaml
version: 2
models:
  - name: stg_your_table
    columns:
      - name: id                          # replace with a real key column
        tests: [not_null, unique]
```

Run it

```
dbt run   -s stg_your_table
dbt test  -s stg_your_table
```

Handy tips

Store failing rows to inspect them:

```
# in dbt_project.yml or per test
tests:
  +store_failures: true
```

Severity (warn vs error)

```
tests:
  - not_null:
      severity: warn
```

Deduplication

it can be done in mainly two ways.

Through SQL in a model where it is conceptually does this:

```
select * except(_rn)
from (
  select
    *,
    row_number() over (
      partition by <partition_by>
      order by <order_by>
    ) as _rn
  from <relation>
)
where _rn = 1
```

By using macro

`dbt_utils.deduplicate` is a **macro** from the `dbt-utils` package that keeps **one row per key** using a `ROW_NUMBER()` window so you don’t have to hand-write the pattern each time.

Macros

In your project root create `packages.yml` with an exact version:

```yaml
packages:
  - package: dbt-labs/dbt_utils
    version: "1.3.0"
```

run:
dbt clean (Deletes the dbt_packages/ to get a clean state)

dbt deps (This command installs the dependencies specified in your packages.yml)

Quick sanity check after install: A dbt_packages/dbt_utils/ appears

Using the macro

```
select *
from {{ dbt_utils.deduplicate(
  relation = ref('stg_customers__cleaned'),
  partition_by = 'customer_id',
  order_by = 'loaded_at desc'
) }}
```

Parameters breakdown:

`relation = ref('stg_customers__cleaned')`

- Specifies the source table/model to deduplicate
- `ref()` creates a proper reference to the `stg_customers__cleaned` model
- This ensures dbt understands the dependency relationship

`partition_by = 'customer_id'`

- Defines what constitutes a "duplicate"
- Records with the same `customer_id` are considered duplicates
- The deduplication logic will group rows by this field

`order_by = 'loaded_at desc'`

- Determines which duplicate record to keep when multiples exist
- `desc` means it keeps the most recent record (highest `loaded_at` timestamp)
- If you had multiple records for the same customer, it would keep the one with the latest `loaded_at` value

Important notice

- When using macro with you will have use two staging tables

  one to clean/unnested (standard)

  than you run deduplicating macro in a new model and reference the cleaned staging model in it.