codecentric

Data Pipelines, Documentation and Lineange with SQL & dbt

Google
BigQuery

snowflake

Step 2: Create database connection

[9]

```
import os

connection = mysql.connector.connect(
    host=os.environ
    user=os.environ
    passwd=os.envir
    database=os.env
```

```
def connect(*args: Any,
            **kwargs: Any) -> None
```

Create or get a MySQL connection object

In its simpliest form, Connect() will open a connection to a MySQL server and return a MySQLConnection object.

When any connection pooling arguments are given, for example pool_name or pool_size, a pool is created or a previously one is used to return a PooledMySQLConnection.

Returns MySQLConnection or PooledMySQLConnection.

Step 3: Run Pandas

[14]

```
df = pd.read_sql_query("select * from datasources.gpu_model_data", con = connection)
df
```

Table Visualize

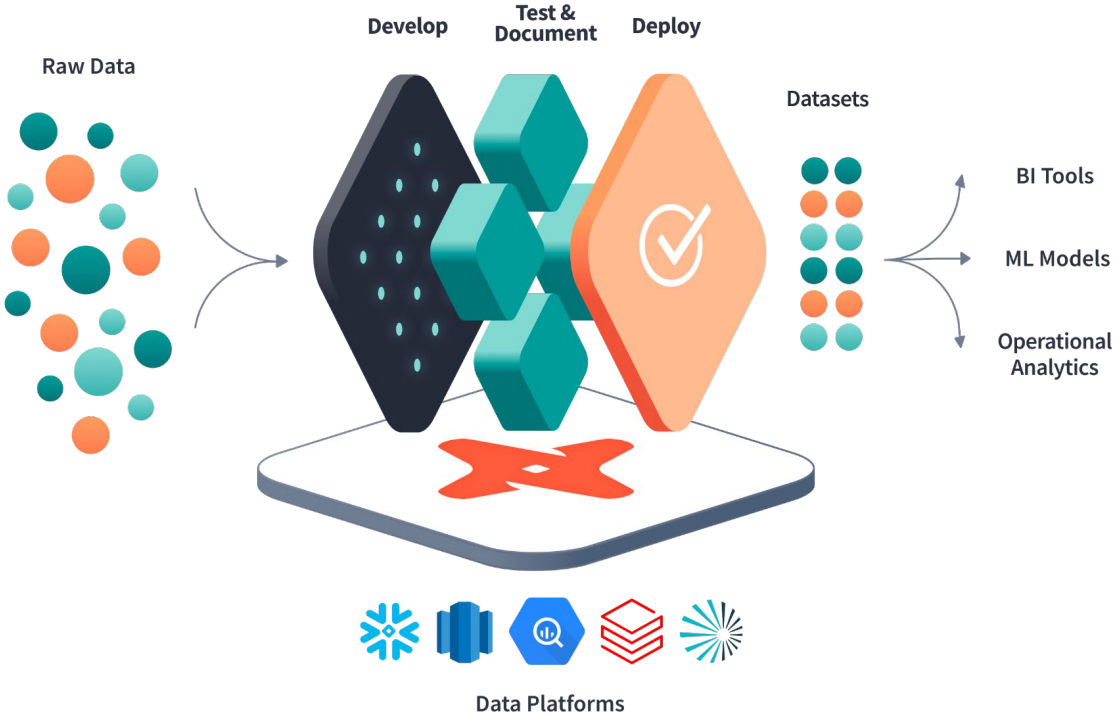

untitled1.py
query = """
SELECT
*
FROM employee e1
WHERE salary >
(SELECT AVG(salary)
FROM employee e2
WHERE e1.departmet_id = e2.department_id)
"""
result = pd.read_sql(query)
result[["first_name", "last_name", "salary"]]

Why is it bad?

- high degree of flexibility in the solution
- tendency to do “solved” work
- no standards
- complex to integrate
- complex to extend
- complex to maintain

dbt

“SQL Data Transformations with Software Engineering Best Practices”



- The T in ELT - running on the data in the warehouse
- A dataset centric approach for dashboards, reports and analysis
- Code & Configuration

- A cli tool (OpenSource)
- A SaaS Offering (by dbt Labs, Series D with 4,2 Billion validation)

Let’s have a look - Live

Sources

- the data loaded into the warehouse, without any further transformation
- references *physical assets* (e.g. tables)
- allows to do
  - test the data
  - test the freshness of the data
- can be generated with dbt-codegen package

`dbt run-operation generate_source`

sources.yml

```yaml
sources:
  - name: raw
    tables:
      - name: raw_customers
        columns:
          - name: id
          - name: first_name
          - name: last_name

      - name: raw_orders
        columns:
          - name: id
          - name: user_id
          - name: order_data
          - name: status

...
```

Snapshots

Build a history of regularly provided data representing the current state

orders_snapshot.sql

```
{% snapshot orders_snapshot %}

{{
    config(
        target_schema='snapshots',
        unique_key='id',
        strategy='check',
        check_cols=["status"]
    )
}}

select * from {{ source('raw', 'raw_orders') }}

{% endsnapshot %}
```

Seeds

- (Nearly) static data, e.g country codes
- Seeds are versioned controlled
- No mechanism to regularly import data!

Models

- defined as sql file
- with sql code
- with optional yml config

⇒ same for seeds/models/snapshots

stg_customers.sql

```sql
with source as (
    select * from {{ source('raw','raw_customers') }}
),

renamed as (
    select
        id as customer_id,
        first_name,
        last_name
    from source
)

select * from renamed
```

schema.yml

```yaml
version: 2

models:
  - name: stg_customers
    description: The raw customer data with some colum renaming.
    columns:
      - name: customer_id
        tests:
          - unique
          - not_null
```

References

Reference ***models, snapshots*** and ***seeds***

schema.yml

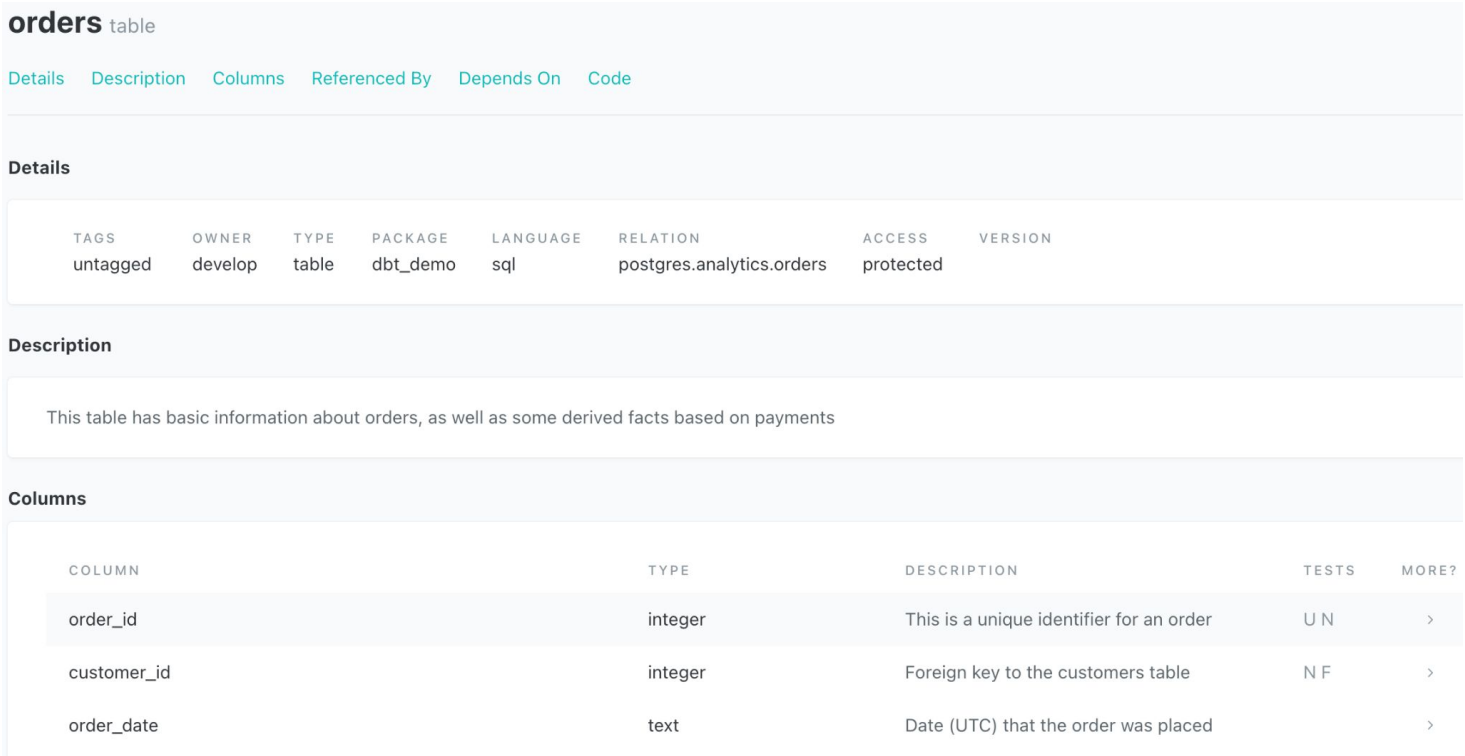
```
with orders as (

    select * from {{ ref('stg_orders') }}

),

payments as (

    select * from {{ ref('stg_payments') }}

),

...
```

Lineage

Built when running dbt run, based on the DAG of references

Documentation

Build from all the information available



orders table

Details Description Columns Referenced By Depends On Code

Details

TAGS	OWNER	TYPE	PACKAGE	LANGUAGE	RELATION	ACCESS	VERSION
untagged	develop	table	dbt_demo	sql	postgres.analytics.orders	protected	

Description

This table has basic information about orders, as well as some derived facts based on payments

Columns

COLUMN	TYPE	DESCRIPTION	TESTS	MORE?
order_id	integer	This is a unique identifier for an order	U N	›
customer_id	integer	Foreign key to the customers table	N F	›
order_date	text	Date (UTC) that the order was placed		›

Documentation

Can also be provided as separate markdown files

status text

Details

Description

Orders can be one of the following statuses:

STATUS	DESCRIPTION
placed	The order has been placed but has not yet left the warehouse
shipped	The order has ben shipped to the customer and is currently in transit
completed	The order has been received by the customer
return_pending	The customer has indicated that they would like to return the order, but it has not yet been received at the warehouse
returned	The order has been returned by the customer and received at the warehouse

Generic Tests

Accepted Values: placed, shipped, completed, return_pending, returned

Tests

Generic

staging.yaml

```yaml
models:
  - name: stg_customers
    description: The raw customer data with some colum renaming.
    columns:
      - name: customer_id
        tests:
          - unique
          - not_null
```

Single

tests/assert_customers_id_not_null.sql

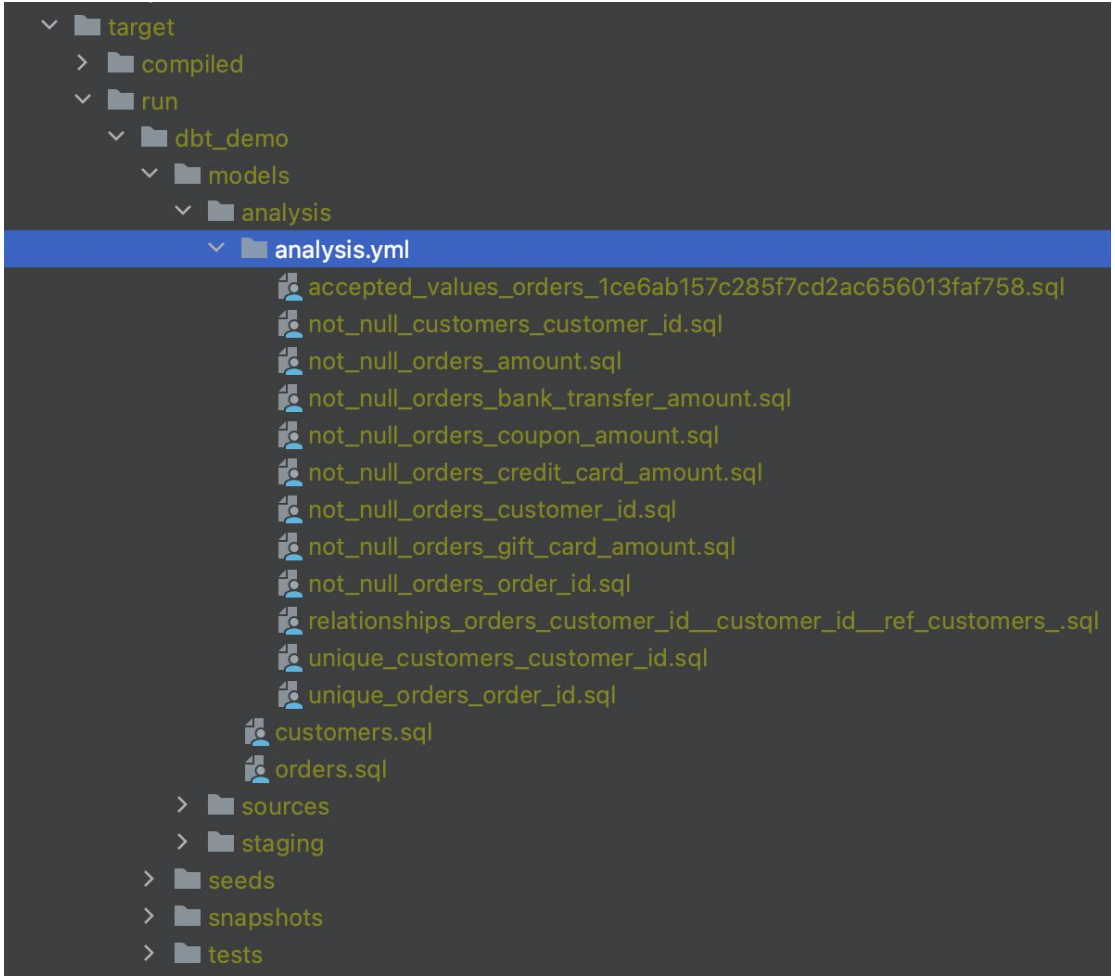
```sql
select *
from {{ ref("stg_customers") }}
where customer_id is null

```

Test-Driven model development (Test first, then yml + sql)

Develop against data contracts, verify them

The target folder

- contains the compiled and ran sql files
- the compiled graph
- the run results
- the docs (the catalog)



Jinja & Macros

orders.sql

```sql
select
    order_id,

    {% for payment_method in payment_methods -%}
    sum(case when payment_method = '{{ payment_method }}' then amount else 0 end) as {{ payment_method }}_amount,
    {% endfor -%}

    sum(amount) as total_amount

from payments

group by order_id
```

cents_to_dollars.sql

```sql
{% macro cents_to_dollars(column_name, precision=2) %}
    ({{ column_name }} / 100)::numeric(16, {{ precision }})
{% endmacro %}
```

having different behaviors depending on the target

Packages

- dbt-expectations, dbt-date, dbt-privacy, dbt-constraints, elementary, common data

Table shape

- expect_column_to_exist
- expect_row_values_to_have_recent_data
- expect_grouped_row_values_to_have_recent_data
- expect_table_aggregation_to_equal_other_table
- expect_table_column_count_to_be_between
- expect_table_column_count_to_equal_other_table
- expect_table_column_count_to_equal
- expect_table_columns_to_not_contain_set
- expect_table_columns_to_contain_set
- expect_table_columns_to_match_ordered_list
- expect_table_columns_to_match_set
- expect_table_row_count_to_be_between
- expect_table_row_count_to_equal_other_table
- expect_table_row_count_to_equal_other_table_times_factor
- expect_table_row_count_to_equal

Distributional functions

- expect_column_values_to_be_within_n_moving_stdevs
- expect_column_values_to_be_within_n_stdevs
- expect_row_values_to_have_data_for_every_n_datepart

packages.yml

```
packages:
  - package: calogica/dbt_expectations
    version: [">=0.8.0", "<0.9.0"]
```

Install with `dbt deps`

dbt + Formatting

.sqlfluff

```
[sqlfluff]
templater = dbt
dialect = postgres
max_line_length = 120

[sqlfluff:templater:jinja]
apply_dbt_builtins = true

```

sqlfluff

.pre-commit-config.yaml

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
      - id: mixed-line-ending
      - id: check-merge-conflict
      - id: check-added-large-files
        args: ["--maxkb=200"]
        exclude: ^seeds/
  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black
  - repo: https://github.com/charliermarsh/ruff-pre-commit
    rev: "v0.0.262"
    hooks:
      - id: ruff
        args: [--fix, --exit-non-zero-on-fix]
```

pre-commit hooks

Database Connection

dbt_project.yml

```yaml
name: 'dbt_demo'
version: '1.0.0'
config-version: 2

profile: 'dbt_demo'
```

profiles.yml

```yaml
dbt_demo: # profile name
  target: postgres_dwh # this is the default target
  outputs:
    postgres_dwh:
      type: postgres
      host: localhost
      user: develop
      password: develop
      port: 5432
      dbname: postgres
      schema: analytics
```

exact configuration depends on target system

Dev/Prod Seperation

profiles.yml

```yaml
dbt_demo: # profile name
  target: postgres_dwh_ci
  outputs:
    postgres_dwh_ci:
      type: postgres
      host: localhost
      user: develop
      password: develop
      port: 5432
      dbname: postgres
      schema: ci
```

profiles.yml

```yaml
dbt_demo: # profile name
  target: postgres_dev_mniehoff
  outputs:
    postgres_dev_mniehoff:
      type: postgres
      host: localhost
      user: develop
      password: develop
      port: 5432
      dbname: postgres
      schema: dev_mniehoff
```

beware: snapshots are not namespaced

CI/CD + Changes

- Usual Gitflow with Merge Requests
- Use own schema for CI
- Run + Test on CI
  - be explicit in model selection!

- Data Folds `data-diff` for creating diffs during development
  - If possible: integrate into MR

```
postgres.analytics.customers <> postgres.dev_mniehoff.customers
Column(s) removed: {'most_recent_order', 'first_order'}

  Rows Added    Rows Removed
------------  --------------
           0               0

Updated Rows: 59
Unchanged Rows: 41

Values Updated:
customer_lifetime_value: 59
number_of_orders: 59
last_name: 0
first_name: 0


postgres.analytics.orders <> postgres.dev_mniehoff.orders
No row differences
```

I could go on ..

- exposures
- metrics (WIP, the old dbt-metrics is deprecated)
- materialization
- incremental models
- ad-hoc analysis
- hooks
- authorization
- python models

exposures.yml

```yaml
exposures:

  - name: weekly_jaffle_report
    type: dashboard
    maturity: high
    url: https://bi.tool/dashboards/1
    description: >
      Did someone say "exponential growth"?

    depends_on:
      - ref('orders')
      - ref('customers')

    owner:
      name: Callum McData
      email: data@jaffleshop.com
```

Ecosystem + Integration

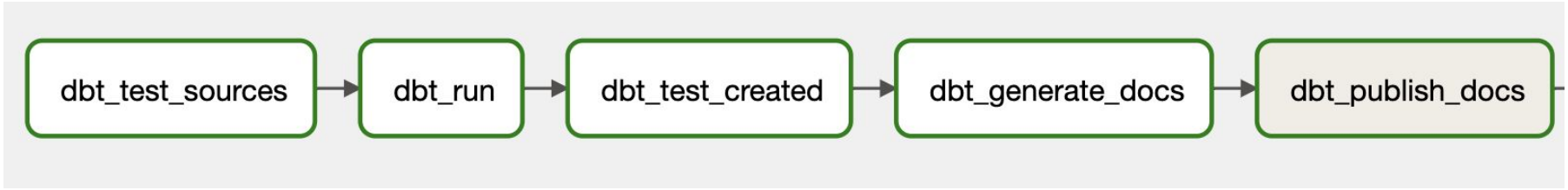
https://github.com/Hiflylabs/awesome-dbt

Orchestration

- Airflow, prefect, dagster, …
- Best experiences:
  - Create a docker container based on the project
  - Use this container to run commands

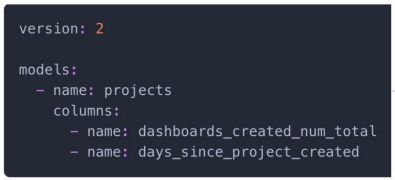
data import DAG:

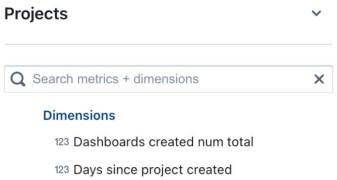
dbt DAG:



Vizualization + Query

- Everything that can connect to the storage
- Bonus: integration with dbt, like Lightdash





duckDB + dbt

- Community Plugin
- DuckDB: 🤯
  - sqllite but for OLAP
  - fast SQL layer for Files/Blobs/…
- ⇒ dbt on Blob Storage/Files

profiles.yml

```yaml
your_profile_name:
  target: dev
  outputs:
    dev:
      type: duckdb
      path: 'file_path/database_name.duckdb'
      extensions:
        - httpfs
        - parquet
      settings:
        s3_region: my-aws-region
        s3_access_key_id: "{{ env_var('S3_ACCESS_KEY_ID') }}"
        s3_secret_access_key: "{{ env_var('S3_SECRET_ACCESS_KEY') }}"
```

ELT

Why is it good?

Simplicity

- good starting point for documentation
- data lineage within the dwh can be easily understood and visualized
- supports dataset-centric visualization (opposed to query- or semantic-driven)

Engineering

- logic and config treated as code and stored in source control
- repeatable, automated test, even test driven would be possible
- easy automation and CI/CD

Ecosystem

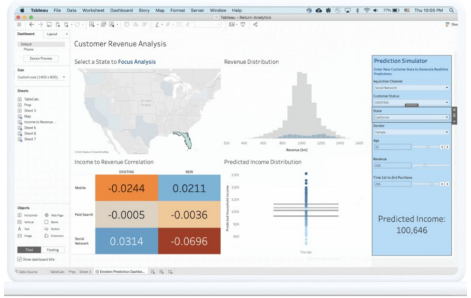
- huge ecosystem with great integrations
- works with every major storage
- ready to adapt with new trends and technologies

What is it not?

Import

Visualization

Data Catalog



Low-/No-Code

Repository & Slides

https://github.com/**mniehoff/dbt-demo**

Wir unterstützen Sie bei Ihren Data-Projekten.

Kontaktieren Sie uns gern!

codecentric AG
Hochstraße 11
42697 Solingen

Matthias Niehoff
Head of Data & AI
matthias.niehoff@codecentric.de
www.codecentric.de

codecentric