Repository & Slides

codecentric/dbt-workshop

Ein Datenbank-Client für BigQuery ist empfehlenswert:

- DBeaver
- DBVisualizer
- TablePlus
- IntelliJ/PyCharm
- Metabase
- …

Config

- Projekt:
- Key:

codecentric

dbt für Anfänger und Fortgeschrittene

Agenda

9:00 1. Block

- Kurze Einführung dbt
- Einrichten + Datenbankverbindung
- DBT Sources & Modelle

10:45 Pause

11:00 2. Block

- Dokumentation & Lineage
- Plugins
- Testen

12:30 Mittagspause

13:30 3. Block

- Makros
- Deployment, Automatisierung
- Staging, Production, CI/CD

14:30 Pause

14:45 Abschluss

- Unit Tests
- Incremental Models
- Linting, Formatter

15:30 Ende

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

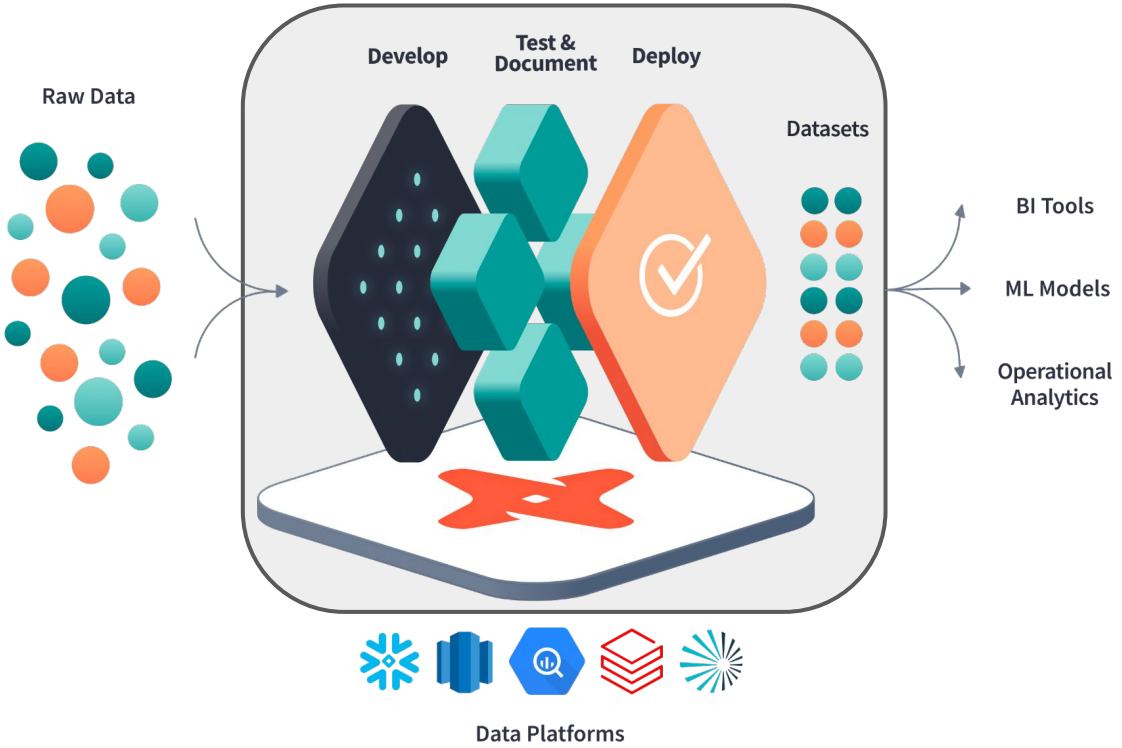
```
query = """
SELECT
 *
FROM employee e1
WHERE salary >
    (SELECT AVG(salary)
     FROM employee e2
     WHERE e1.departmet_id = e2.department_id)
"""
result = pd.read_sql(query)
result[["first_name", "last_name", "salary"]]
```

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 get started

Repository & Slides

codecentric/dbt-workshop

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

profiles.yml

- in current working directory
- default to the `~/.dbt/`

- example in Repo-Root: `profiles_template.yml`

- Config
  - BigQuery Projekt:
  - Key File:

Exercise: Connect to BigQuery

- Create a profiles.yml in your repository root (there is a template as well)
- Configure the BigQuery target
  - Projekt:
  - Key File:
  - Download the Key File and provide the local path
- Check with `dbt debug`
- Install dependencies with `dbt deps`

dbt_project.yml

- the main config for the project
- defines the profile
- general model, seed, snapshot config
- override default directory structure

dbt_project.yml

```yaml
name: 'dbt_workshop'
version: '1.0.0'

profile: 'dbt_workshop'

models:
  dbt_workshop:
    +materialized: table
    staging:
      schema: staging

seeds:
  +schema: seeds
```

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

...
```

Exercise: Generate Source yml

- Use this documentation: https://github.com/dbt-labs/dbt-codegen#usage
- `dbt run-operation generate_source --args '{"schema_name": "raw","generate_columns": true}'`

- Generate sources for `raw_payments` and add it to `models/sources/sources.yml`

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 %}
```

Exercise: Execute Snapshot

- Review the the snapshot for the orders in `snapshots/orders_snapshot.sql`
- Create the snapshot with `dbt snapshot`
- Watch the generated snapshot table in the `snapshots` schema
- Change some state in the `raw_order` table
- Run `dbt snapshot` again
- Watch the changes in the generated snapshot table

Seeds

- (Nearly) static data, e.g country codes
- Seeds are versioned controlled
- Not a mechanism to regularly import data!

country_codes.csv

```
country_code,country_name
US,United States
CA,Canada
GB,United Kingdom
DE,Germany
AT,Austria
CH,Switzerland
```

Exercise: Explore & add seeds

- Create the snapshot with `dbt seed`
- Watch the generated seed table in the `seeds` schema

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') }}

),

...
```

Stages/Zones in the DWH

Exercise: Create Models

- Create the staging model for `payments`
- Use Common Table expression
- rename the id column to `payment_id`

- Finish the `customers` analysis model
- add the payment information
  - use the data in the `payment` staging table
  - join the data with the `order` information to get the `customer_id`
  - the `customer_payments` should contain the total payment amount per customer
  - add this value to the final output as `customer_lifetime_value`

Do not forget to define the yml!

Use `dbt run --select staging.stg_payments` / `dbt run --select +analysis.customers`

Exposure

- Usages of models in external applications, processes..

Available properties

Required:

- **name**: a unique exposure name written in snake case
- **type**: one of `dashboard`, `notebook`, `analysis`, `ml`, `application` (used to organize in docs site)
- **owner**: `name` or `email` required; additional properties allowed

Expected:

- **depends_on**: list of refable nodes, including `ref`, `source`, and `metric` (While possible, it is highly unlikely you will ever need an `exposure` to depend on a `source` directly)

Optional:

- **label**: may contain spaces, capital letters, or special characters.
- **url**: enables the link to **View this exposure** in the upper right corner of the generated documentation site
- **maturity**: one of `high`, `medium`, `low`

General properties (optional)

exposures.yml

```yaml
version: 2

exposures:

  - name: weekly_jaffle_metrics
    label: Jaffles by the Week
    type: dashboard
    maturity: high
    url: https://bi.tool/dashboards/1
    description: >
      Did someone say "exponential growth"?

    depends_on:
      - ref('fct_orders')
      - ref('dim_customers')
      - source('gsheets', 'goals')
      - metric('count_orders')

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

Exercise: Create an Exposure

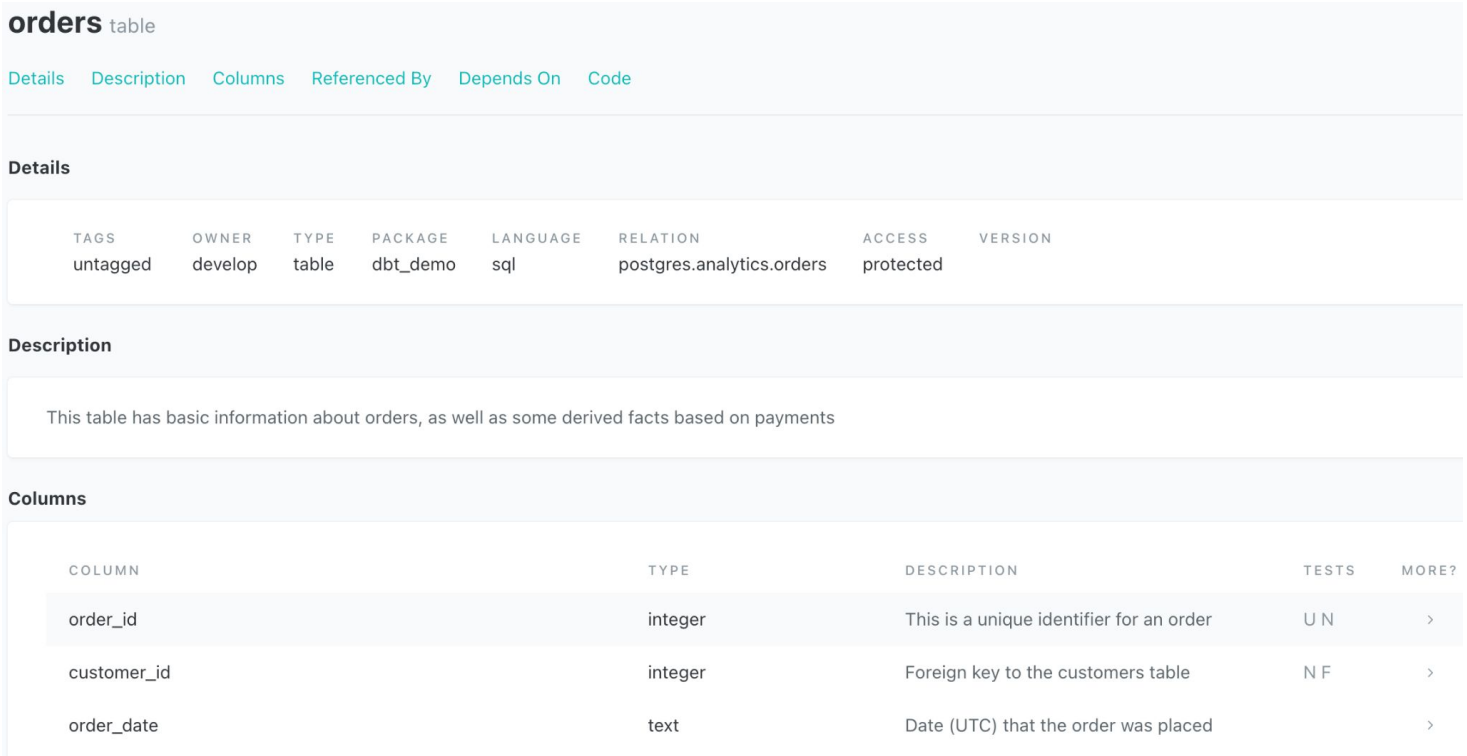
- “Business” is using a dashboard on the customers table, to see the valuable customers. The upper management relies heavily on this for decision making.
- The “Data Science Team” uses all the staging data for training a demand forecast model. The project is still experimental.

Lineage

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

Documentation

Build from all the information available



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

Exercise: Generate & Visit Documentation

- `dbt docs generate`
- `dbt docs serve`

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

Exercise: Add dbt provided Generic Tests

- Add generic tests to Source `payments`
  - `id`: unique, not null
  - `order_id`: reference zu source raw_orders, Feld id
- Staging stg_payments
  - `id`: unique, not null
  - `payment_method`: erwartet credit_card, coupon, bank_transfer oder gift_card
- Analysis `customers`
  - `id`: unique, not null

Relevante Dokumentation:

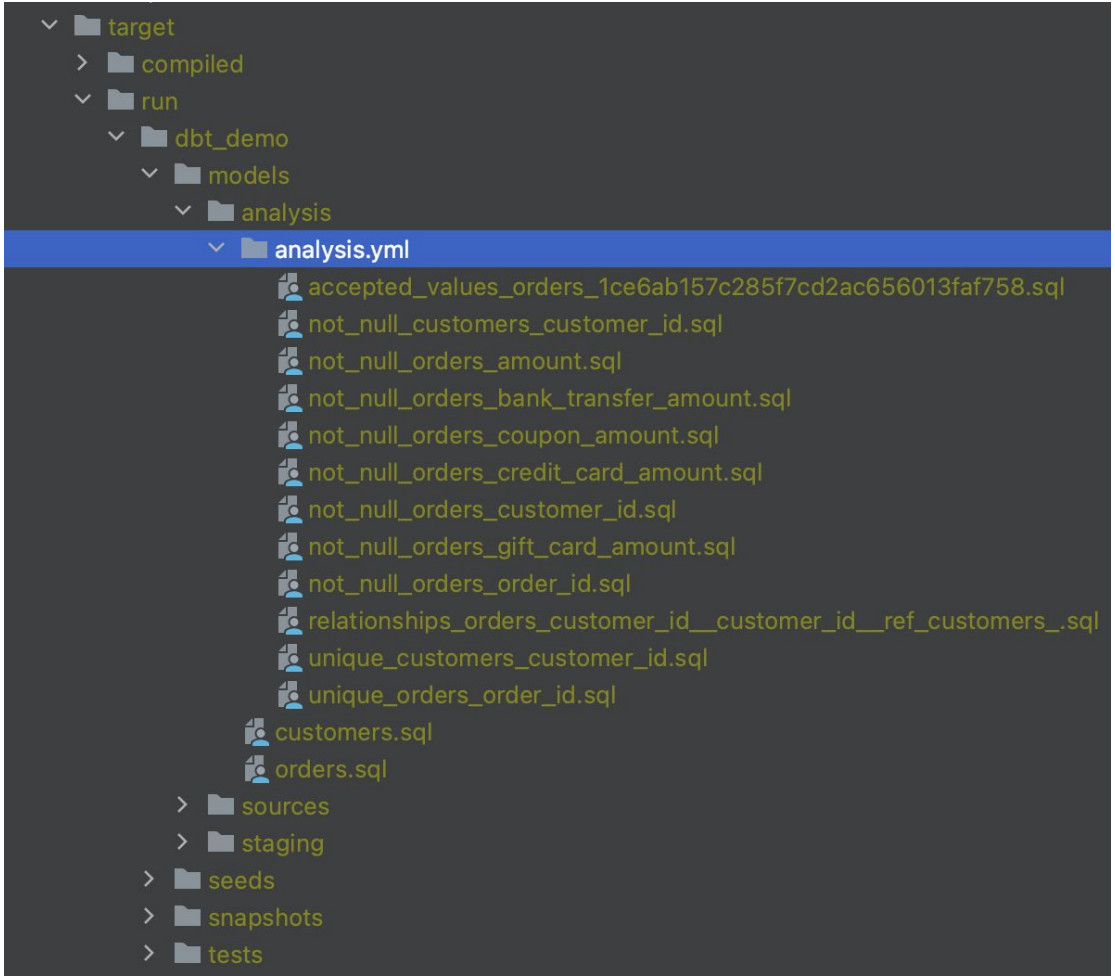
https://docs.getdbt.com/docs/build/tests#generic-tests

Exercise: Run tests & view failures

- Run tests with `dbt test`
- Remove some of the accepted payment methods
- Run tests with `dbt test --store-failures`
- Have a look at the failures

The target folder

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



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

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

Install with `dbt deps`

Exercise: Install Packages

- Review the `packages.yml`
- Add `elementary-data/elementary` in version 0.10.0 oder einer Bugfix version (0.10.1)
- Add `calogica/dbt_expectations` in version 0.8.0 oder einer Bugfix version (0.10.1)
- Run `dbt deps`

- Configure elementary models in `dbt_project.yml`

```
models:
  elementary:
    +schema: elementary
```

Exercise: Write dbt-expectations tests

- Add one or two dbt-expactation tests
- Ideas:
  - Shape (number of rows, columns)
  - Aggregations (Expect Max/Min Values)

- Available tests: https://github.com/calogica/dbt-expectations#available-tests

Exercise: Use Elementary for test-visualization

- make sure to have the `elementary-data` Python Package installed (version 0.10.x)
- run models and tests (`dbt run` + `dbt test` or `dbt build`)
- create the report with `edr report`

- Bonus:
  - Explore the anomaly tests of elementary data: https://docs.elementary-data.com/guides/add-elementary-tests
  - Add an anomaly detection test, e.g. a volume test on the staging order model
  - Run the test and visualize it again

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

Exercise: Write a macro

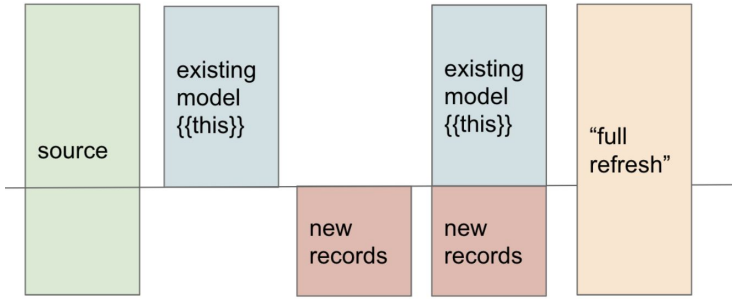
- Write and use a Macro:
  - in `stg_payments` the `amount` is stored in cents
  - convert it do dollar
  - write a macro `cents_to_dollars` so it can be reused
  - (Hint: BigQuery does not support casting to paramterized types like `NUMERIC(15,2)`, therefore we only cast the result to `NUMERIC`)

Exercise: Jinja2 + Model

- Finish the `orders` analysis model
- add the payment information
  - use the data in the `payment` staging table
  - for each order in the payment table
    - have a column for each payment method and the amount paid with the method
    - and a `total_amout` column
- The final table should contain the separated payment information and the order information

At the top of the `orders.sql` the model is deactivated. Active it when you work on this!

Exercise: Incremental models



Incremental models are very useful.

Using an incremental model limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.

This is a great source for explanation

https://docs.getdbt.com/guides/best-practices/materializations/4-incremental-models

Task:

- enable the models `raw_orders_copy`, `orders_per_year`
- do `dbt run --select raw_orders_copy`
- do `dbt run --select orders_per_year`
- in `raw_orders_copy`, set the config `append_data=true`
- do `dbt run --select orders_per_year` and see how it increments

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
```

Demo/Exercise:
Use data-diff on a branch with changes

only with Postgres!

Make sure to adjust your default target in the profiles.yml

- Make sure pip packages `data-diff` and `data-diff[postgres]` are installed
- Make sure the production schema (`postgres.analysis`) and the data exist.

  Locally you will work on your dev schema.
- Add the following vars to the dbt_project.yml

```
vars:
  data_diff:
    prod_database: postgres
    prod_schema: analysis
```

- Do a local change (e.g. remove a column from a `SELECT` or change a `WHERE` clause)
- Do `dbt run`!
- Run `data-diff --dbt` on the CLI

Exercise: dbt-unit-testing

Sometimes, business logic is so crucial that it should be tested on a toy dataset to check if everything works as expected.

dbt-unit-testing comes in handy here!

https://github.com/EqualExperts/dbt-unit-testing

- Write the expected result for the mock data in the unit test
- Find out what quite invasive part is missing for it to work

dbt + Formatting: sqlfluff

.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

Exercise: Do a commit & use pre-commit

- Make sure the `pre-commit`, `sqlfluff` and `sqlfluff-templater-dbt` Python Packages are installed
- Install hooks with `pre-commit install`
- Do a commit and watch the checks, adjust the code until the commit succeeds

- Checks can also be executed by `pre-commit run --all-files`

I could go on ..

- 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

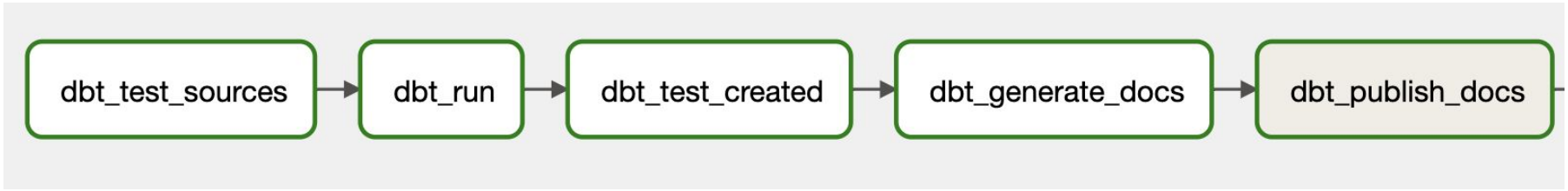
https://github.com/Hiflylabs/awesome-dbt

Orchestration

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

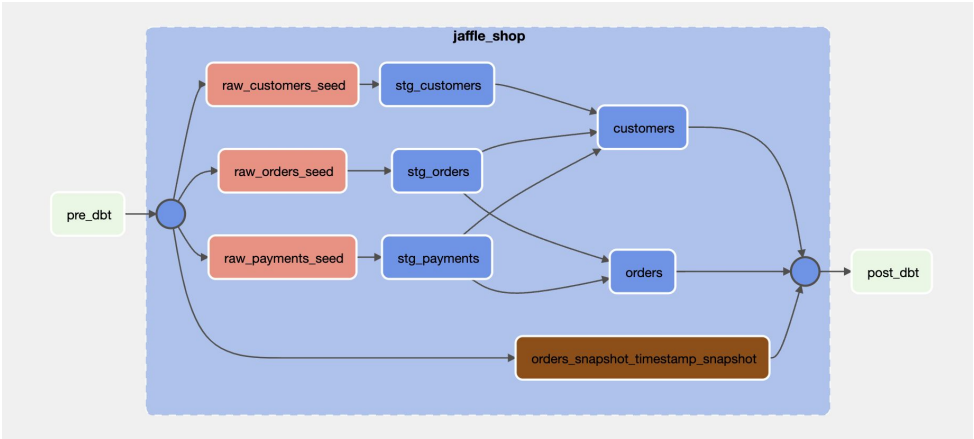
data import DAG:

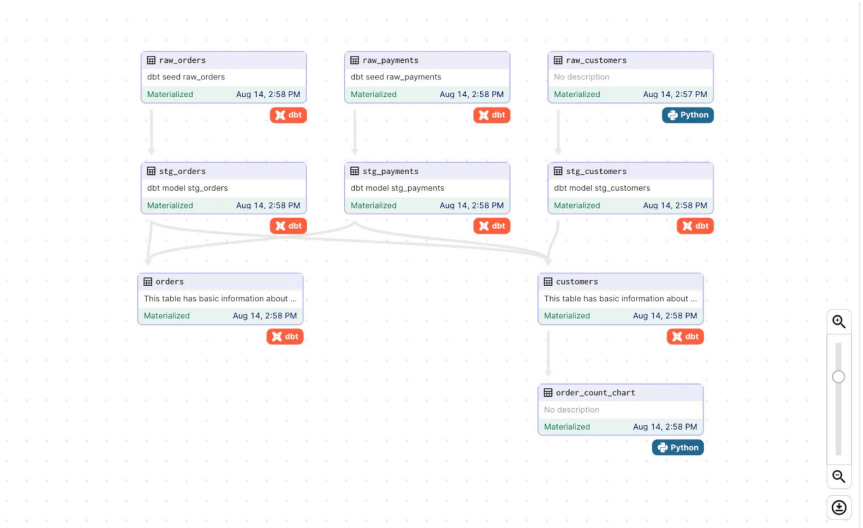
dbt DAG:



Orchestration

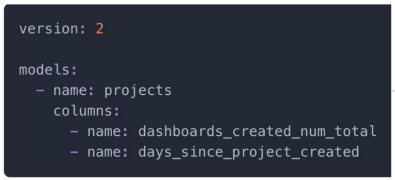
- At least dagster & airflow support “included dbt” - dbt models & tests als nodes in a job/DAG
  - Airflow: https://astronomer.github.io/astronomer-cosmos/index.html
  - dagster: nativ

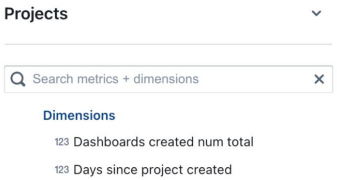


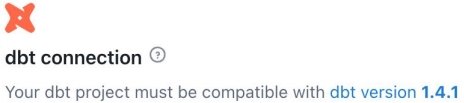


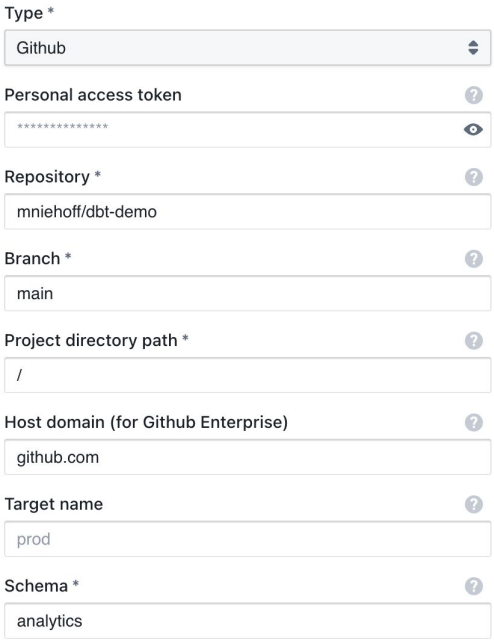
Vizualization + Query

- 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

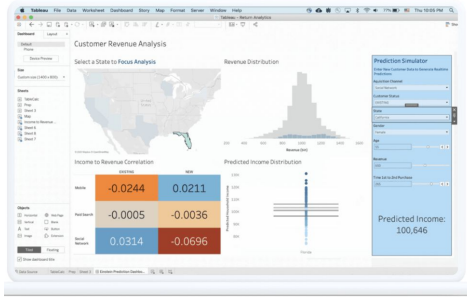
- huge ecosystem with great integrations
- works with every major storage
- ready to adapt with new trends and technologies

What is it not?

Import

Visualization

Data Catalog



Low-/No-Code


THANK
YOU

We support you with your data projects

Feel free to contact us!

codecentric AG
Hochstraße 11
42697 Solingen

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

codecentric