



Tempest

ACADEMY

Conference  
2023

# Pipelines de dados eficientes: transformando dados brutos em informações facilmente consumíveis

by José henrique Davino





ACADEMY

Conference

# Quem sou?

- Um nordestino VIP - “Vindo do interior de Pernambuco”
- Na área de tecnologia desde 2003
- Mestrado em Ciência da Computação pela UFPE
- Desde 2017 atuando em projetos de Dados
- Na Tempest desde 2021
  - Senior Staff Data Engineer / Research Advisor

# Objetivo

Apresentar uma visão geral do que pode ser usado para lidar grandes volumes [muitas vezes complexos] de forma escalável e gerenciada.



**ACADEMY**

Conference

**01** Introdução

**02** Benefícios e desafios quando se trabalha com dados

**03** A natureza dos dados de cyber

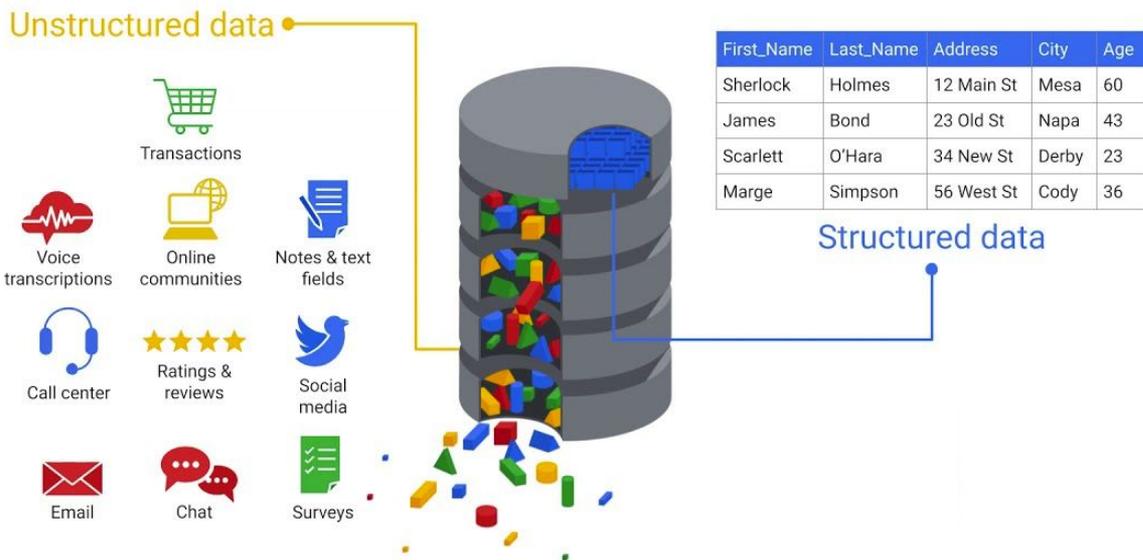
**04** O que o mercado oferece como solução para lidar com Big Data e o que escolhemos

**05** Considerações finais

# O cenário atual

Um **volume enorme** de dados é **coletado** pelas empresas **todos os dias**, em **múltiplos canais** e em **diversos formatos**.

Em geral, as **empresas utilizam** diversos **softwares** para **obtê-los** e **mantê-los**, o que faz com que **se espalhem em silos**.



Empresas que buscam **organizar** e **integrar** seus **dados** para **auxiliar em decisões**, ampliam as chances de assertividade em suas estratégias de negócios.

# Os benefícios são diversos

- Construir uma **fonte única da verdade**;
- **Comportamentos, tendências, padrões** e outras informações sobre nosso próprio negócio **podem ser descobertos**;
- Ser **guiados pelos dados** além do **conhecimento do especialista**;
- **Ampliar a assertividade** na criação de **novas estratégias** de negócios/serviços.

# Desafios mais comuns

- As aplicações **fazem uso de tecnologias distintas**, inclusive de armazenamento;
- **Nem sempre dispõem de mecanismos** [APIs, por exemplo] **de acesso aos dados** para qualquer usuário;
- **A criação** de tais mecanismos **demandaria horas de desenvolvimento**;
- Dependendo da fonte, **pode não possuir um *esquema* definido**.



# Dados de Cyber



[ACADEMY]

Conference

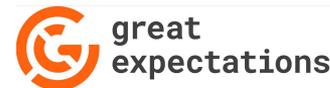


# Dados de Cyber

**Tratar e disponibilizar dados** dessa natureza é, muitas vezes, uma **atividade complexa** que **demandam tempo** e **recurso computacional**, principalmente quando o volume de dados é expressivo!

Quando **realizado de forma inadequada** o **ônus financeiro vem junto!** :(



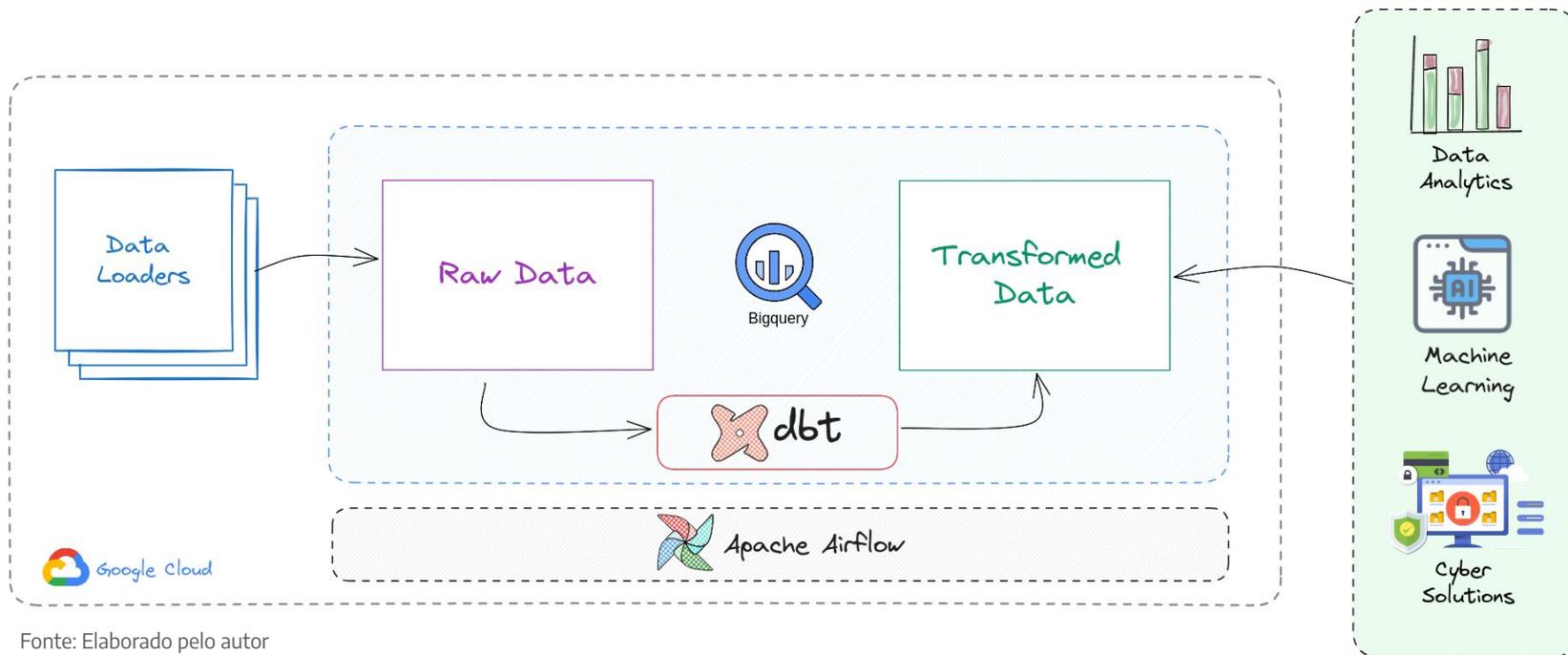


# Desafios e motivações que guiaram a escolha do conjunto de soluções

- Tornar os **dados disponíveis e úteis** para qualquer pessoa dentro de uma organização;
- **Processar** grandes volumes de dados **de forma recorrente**;
- **Curva de aprendizado** que fosse **a menor possível**;
- Que a **sintaxe de consulta** fosse **SQL**;
- Possibilidade **reuso de código** e **trabalho colaborativo**;
- **Algo utilizado** no mercado **por equipes de engenharia de dados**;
- **Menor custo financeiro possível**.



# Arquitetura de Dados - Visão simplificada



Fonte: Elaborado pelo autor

# BigQuery



O BigQuery é um **serviço de armazenamento e análise de dados em nuvem**, totalmente **gerenciado pela Google Cloud** que **permite** às empresas **armazenar e analisar vastos volumes de dados** de maneira rápida e escalável.



Google BigQuery

# O que é o DBT

É uma ferramenta de **transformação SQL** que viabiliza o **desenvolvimento de fluxos de trabalho analítico**, de forma **rápida** e **colaborativa**, seguindo **boas práticas de engenharia de software**, como: modularidade, reuso de código, versionamento de código, testes, entre outros.



# DBT - Principais características

- SQL + Jinja
- Sources, Models, Macros
- Testes
- Documentação
- Snapshots, Seeds, Packages \*

# Como as coisas eram sem o DBT?

## Simple e velho SQL

```
select
  order_id,
  sum(case when payment_method = 'bank_transfer' then amount end) as bank_transfer_amount,
  sum(case when payment_method = 'credit_card' then amount end) as credit_card_amount,
  sum(case when payment_method = 'gift_card' then amount end) as gift_card_amount,
  sum(amount) as total_amount
from app_data.payments
group by 1
```

# Como são agora!

SQL + linguagem de modelagem Jinja.

```
{% set payment_methods = ["bank_transfer", "credit_card", "gift_card"] %}

select
  order_id,
  {% for payment_method in payment_methods %}
  sum(case when payment_method = '{{payment_method}}' then amount end) as {{payment_method}}_amount,
  {% endfor %}
  sum(amount) as total_amount
from app_data.payments
group by 1
```

# Instalando e criando um projeto

```
$ pip install \  
dbt-core \  
dbt-bigquery\  
dbt-postgres \  
dbt-redshift \  
dbt-snowflake \  
dbt-trino  
....
```



```
$ dbt init demo
```



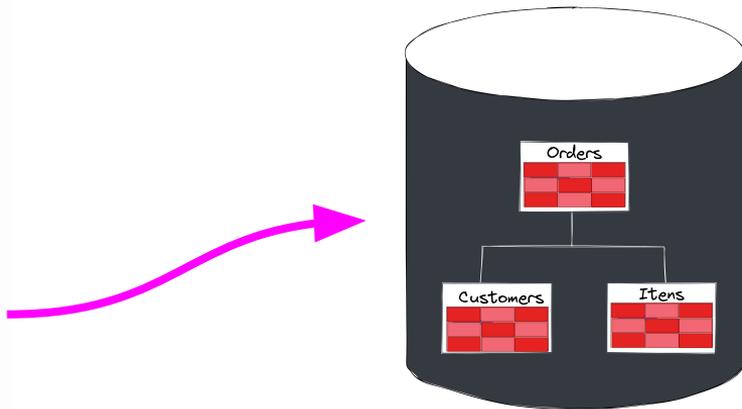
```
.  
├── analyses  
├── dbt_project.yml  
├── macros  
├── models  
│   └── example  
│       ├── my_first_dbt_model.sql  
│       ├── my_second_dbt_model.sql  
│       └── schema.yml  
├── README.md  
├── seeds  
├── snapshots  
├── target  
│   ├── manifest.json  
│   └── partial_parse.msgpack  
└── tests
```

# Sources

```
version: 2

sources:
  - name: sales
    tables:
      - name: orders
      - name: customers

  - name: billing
    tables:
      - name: payments
```



- Definições escritas em arquivo **.yml**;
- Usado para mapear entidades existentes (tabelas, views) em um base de dados.

# Sources

Sources podem ser referenciados a partir de um **model** usando a sintaxe jinja

```
model_a.sql  
  
select  
  ...  
  
from {{ source('sales', 'orders') }}
```



```
model_a.sql  
  
select  
  ...  
  
from sales.orders
```

# Models

- São **queries** construídas em um **arquivo .sql**
  - Seus **resultados são persistidos** no banco de dados **em forma de tabelas ou views**;
  - Cada **Model** deve conter apenas uma declaração SELECT;
  - O **nome do arquivo será o nome da entidade criada no banco de dados** (tabela/view).

```
model_a.sql

with customer_orders as (
  SELECT
    customer_id,
    MAX(order_datetime) as most_recent_order,
    COUNT(order_id) as number_of_orders
  FROM orders
)

SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  co.most_recent_order,
  COALESCE(co.number_of_orders,0) as number_of_orders

FROM customers c

LEFT JOIN customer_orders co
ON co.customer_id = c.customer_id
```

# Models

Podem ser referenciados a partir de outros **models** usando a sintaxe jinja + a função **ref()**

```
model_a.sql  
  
select  
  ...  
  
from {{ ref('model_a') }}
```



```
model_a.sql  
  
select  
  ...  
  
from sales.orders
```

# Macros

- Trechos de **códigos reutilizáveis** definidos em um **arquivo.sql** dentro do diretório **macros**;
- São análogas a “funções” em outras linguagens de programação;
- Ajudam a manter o código mais limpo e mais fácil de entender.

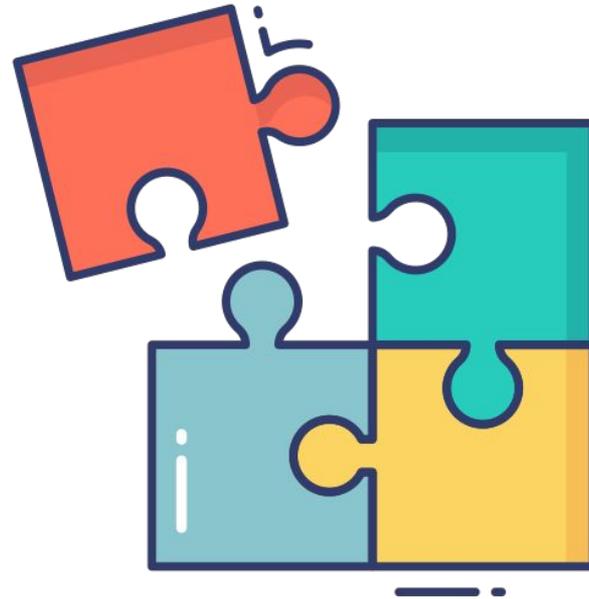
```
{% macro generate_where(field_name, initial_date, final_date) %}  
    {{ field_name }} >= DATE(initial_date)  
    AND {{ field_name }} <= DATE(final_date)  
  
    {% if target.name == 'dev' %}  
        LIMIT 10  
    {%- endif -%}  
  
{% endmacro %}
```

```
model_a.sql  
  
select  
    ...  
from {{ ref('model_a') }}  
  
{{ generate_where("created_at", "2023-07-01", "2023-07-10") }}
```

Fonte: <https://docs.getdbt.com/docs/build/jinja-macros>

# Modularidade

A abordagem modular do dbt permite dividir um pipeline complexo em partes menores.



# Modularidade

Pode ser difícil manter e testar um **Model** contendo várias transformações de dados.

```
complex_model.sql

WITH stations AS (
  SELECT station_id, council_district,
         CASE
           WHEN property_type IN ('parkland', 'sidewalk') THEN 'free_parking'
           ELSE property_type
         END AS property_type,
  FROM `bigquery-public-data.austin_bikeshare.bikeshare_stations`
  WHERE
    property_type IN ('parkland', 'sidewalk', 'paid_parking')
),
trips AS (
  SELECT start_station_id
  FROM `bigquery-public-data.austin_bikeshare.bikeshare_trips`
  WHERE
    start_station_id IS NOT NULL
)
SELECT
  stations.property_type,
  COUNT(*) AS trips,
FROM trips
JOIN stations ON trips.start_station_id = stations.station_id
GROUP BY
  stations.property_type
```

# Modularidade

```
complex_model.sql

WITH stations AS (
  select * from {{ref('model_stations')}}
),
trips AS (
  select * from {{ref('model_trips')}}
)
SELECT
  stations.property_type,
  COUNT(*) AS trips,
FROM trips
JOIN stations ON trips.start_station_id = stations.station_id
GROUP BY
  stations.property_type
```

```
model_stations.sql
...
```

```
model_trips.sql
...
```

# Modularidade

Viabiliza o reuso



[ACADEMY]

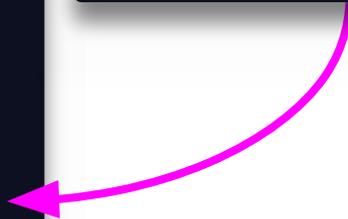
Conference

stations\_per\_district.sql

```
SELECT
  property_type,
  council_district,
  count(*) total_station
FROM
  {{ref('model_stations')}}
GROUP BY
  property_type,
  council_district
```

model\_stations.sql

```
SELECT station_id, council_district,
  CASE
    WHEN property_type IN (...) THEN 'free_parking'
    ELSE property_type
  END AS property_type,
FROM `bigquery-public-data.austin_bikeshare.bikeshare_stations`
WHERE
  property_type IN (...)
```



# Modularidade

Viabiliza o trabalho em equipe simultaneamente

models/model\_stations.sql

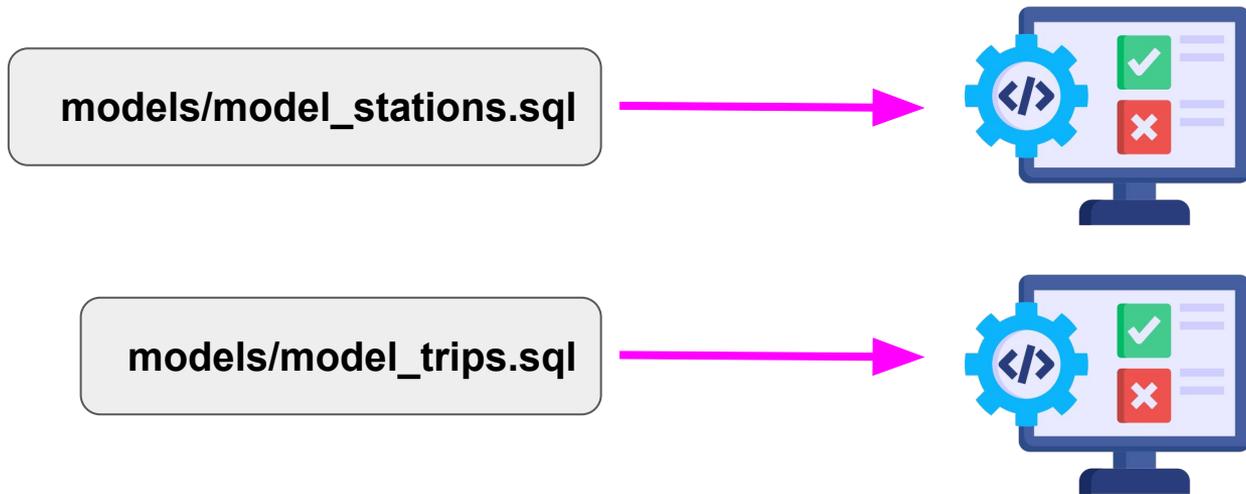


models/model\_trips.sql



# Modularidade

Possibilita testar e validar um **Model** de forma independente

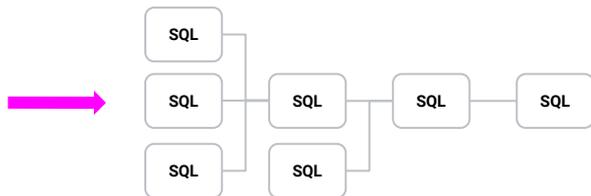
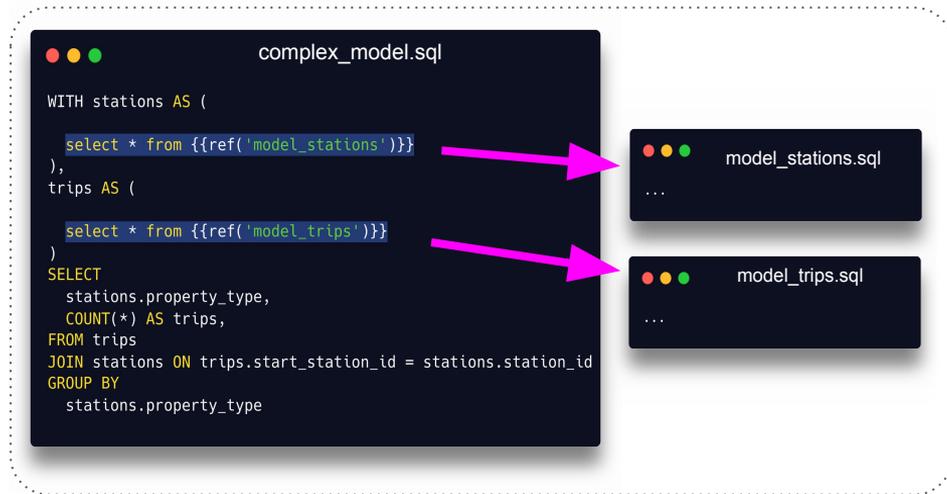


# Dependência

Onde a mágica acontece

```
complex_model.sql

WITH stations AS (
  select * from {{ref('model_stations')}}
),
trips AS (
  select * from {{ref('model_trips')}}
)
SELECT
  stations.property_type,
  COUNT(*) AS trips,
FROM trips
JOIN stations ON trips.start_station_id = stations.station_id
GROUP BY
  stations.property_type
```



## Compilação em SQL

Ocorre em tempo de execução.  
Resolvendo dependências e checando erros.



Google  
BigQuery



## Executa no mecanismo de processamento distribuído

Todas as transformações ocorrem diretamente no **BigQuery**, rodando comandos SQL.

# Testes

Permite checar a corretude de um Model

```
version: 2

models:
  - name: stg_stations
    columns:
      - name: station_id
        tests:
          - not_null # Should not contain null values.
          - unique   # Should be unique
      - name: property_type
        tests:
          # Should be either 'free_parking' or 'paid_parking'
          - accepted_values:
              values: ['free_parking', 'paid_parking']
```

Ajudam a rastrear alterações nos dados ao longo do tempo e garantem que a lógica de transformação permaneça correta à medida que os dados subjacentes mudam!

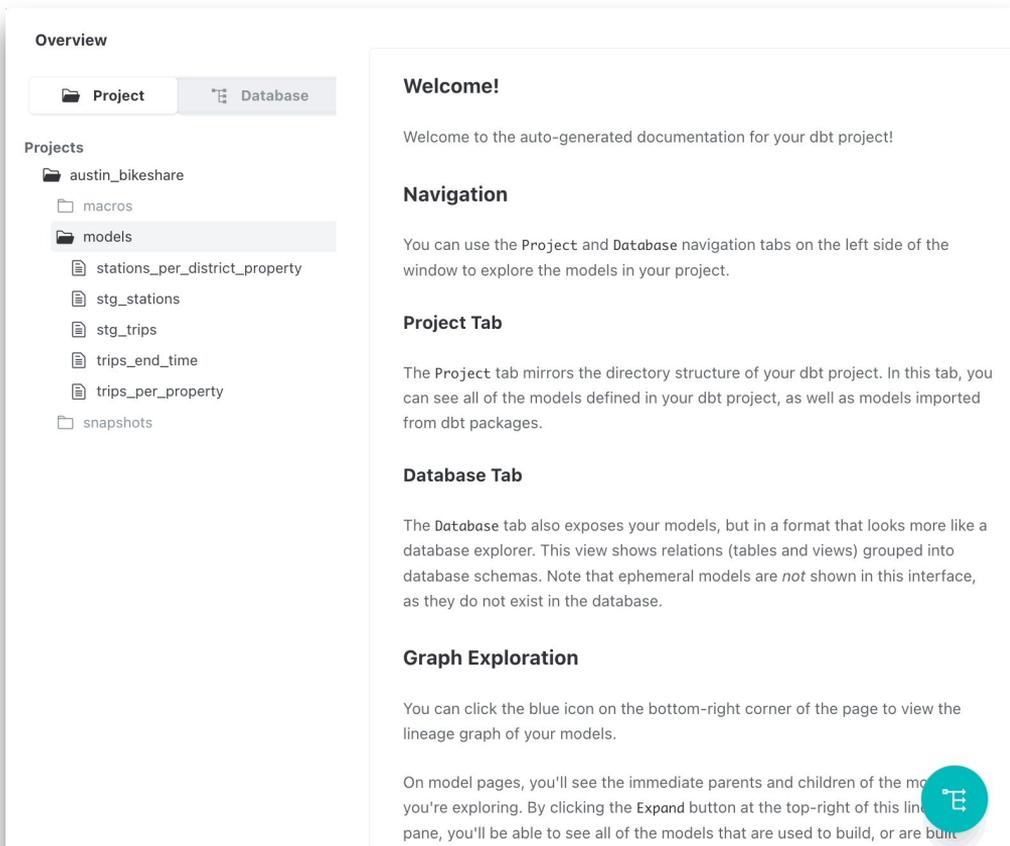
# DBT - Documentação

É possível documentar os modelos e compartilhar a documentação com a equipe.

```
version: 2

models:
  - name: stg_stations
    description: A model station data
    columns:
      - name: station_id
        description: A unique identifier for each station
      tests:
        - not_null # Should not contain null values.
        - unique # Should be unique
    - name: property_type
      tests:
        # Should be either 'free_parking' or 'paid_parking'
        - accepted_values:
            values: ['free_parking', 'paid_parking']
```

# DBT - Documentação



**Overview**

Project Database

**Projects**

- austin\_bikeshare
  - macros
  - models
    - stations\_per\_district\_property
    - stg\_stations
    - stg\_trips
    - trips\_end\_time
    - trips\_per\_property
  - snapshots

**Welcome!**

Welcome to the auto-generated documentation for your dbt project!

**Navigation**

You can use the Project and Database navigation tabs on the left side of the window to explore the models in your project.

**Project Tab**

The Project tab mirrors the directory structure of your dbt project. In this tab, you can see all of the models defined in your dbt project, as well as models imported from dbt packages.

**Database Tab**

The Database tab also exposes your models, but in a format that looks more like a database explorer. This view shows relations (tables and views) grouped into database schemas. Note that ephemeral models are *not* shown in this interface, as they do not exist in the database.

**Graph Exploration**

You can click the blue icon on the bottom-right corner of the page to view the lineage graph of your models.

On model pages, you'll see the immediate parents and children of the model you're exploring. By clicking the **Expand** button at the top-right of this lineage pane, you'll be able to see all of the models that are used to build, or are built

A documentação pode ser acessada via interface web:

1. ***dbt docs generate*** para gerar a documentação do projeto;
2. ***dbt docs serve*** para iniciar um host local com a documentação;
3. ***http://localhost:8080*** via navegador para acessar o conteúdo.

# DBT - Como executá-lo na prática?

## Command line interface (CLI)

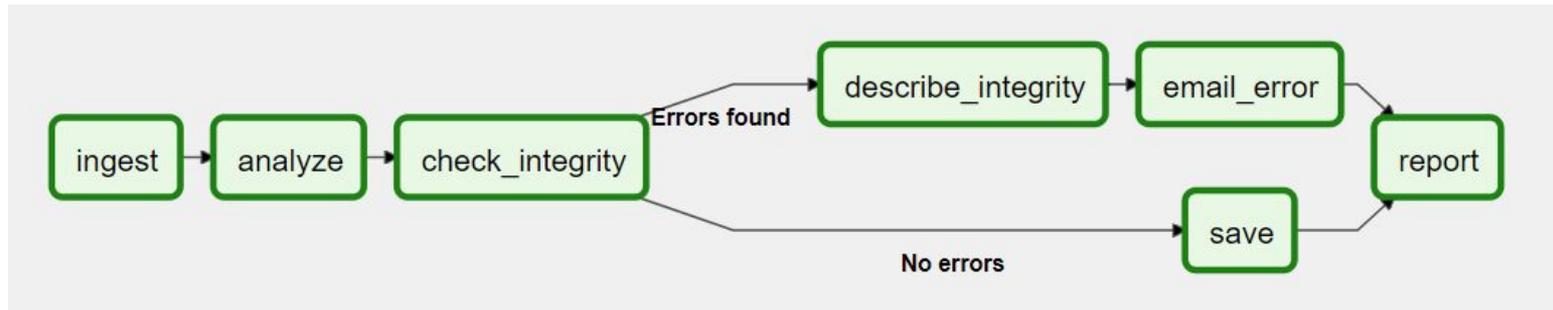
- **dbt compile** - Compila (mas não executa) os modelos em um projeto;
- **dbt run** - Executa os modelos definidos em um projeto;
- **dbt build** - Constrói e testa os resources [models, testes, etc];
- **dbt test** - Executa os testes.

## Ferramentas de orquestração (airflow).

- **DbtRunOperator**
- **DbtTestOperator**

# O que é o Apache Airflow?

É uma plataforma de código aberto para **desenvolver, agendar e monitorar fluxos de trabalho**, baseados em lote (batch). Usa o conceito de **Gráficos acíclicos direcionados** (Directed Acyclic Graphs, DAG) para gerenciar o fluxo da execução de um conjunto de tarefas/ações.

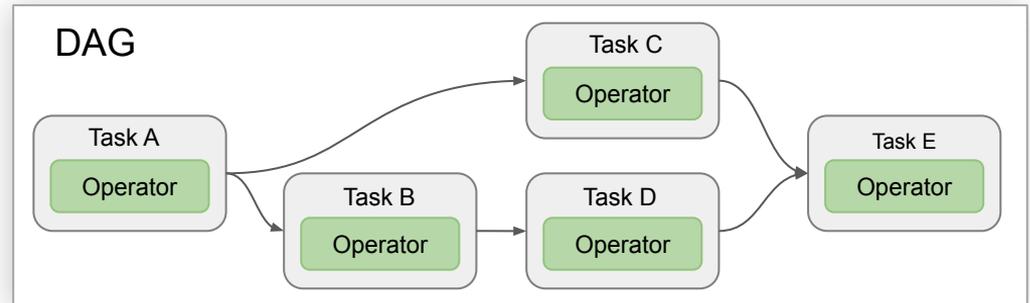


# Por que Apache Airflow?

- **Fácil de operar** - Básico de Python é o único requisito;
- **Lista de Operadores** - Suporta uma infinidade de serviços (cloud, infraestrutura, banco e outros);
- **Extensível** - Permite criar operadores customizados para uma necessidade específica;
- Segue a ideia de “**dividir para conquistar**”

# Airflow - Características Principais

- **DAG** - Espécie de “grafo” que pode possuir caminhos paralelos vindos de um ponto de origem.
- **Task** - Determina como e quando executar um **Operator** dentro do contexto de uma DAG.
- **Operator** - É um tipo de “template” que nos permite executar uma determinada atividade.
  - **PythonOperator**: executa código Python
  - **DbtRunOperator**: executa o comando “dbt run”
  - **PostgresOperator**: realiza consultas SQL no PostgreSQL



# Exemplo de DAG com DBT

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow_dbt.operators.dbt_operator import DbtRunOperator
from datetime import datetime

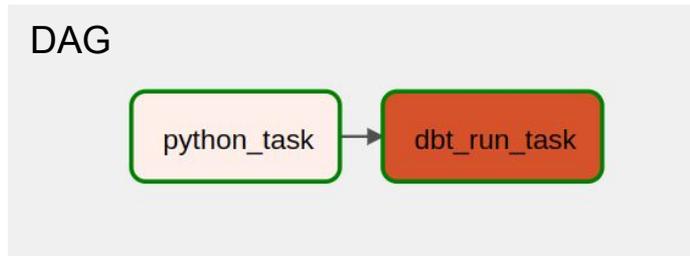
def print_message():
    print("Hello Tempest")

with DAG(
    dag_id="dag_exemplo",
    start_date=datetime(2023, 2, 1),
    schedule_interval = "0 * * * *", #@daily
) as dag:

    task1 = PythonOperator(
        task_id='python_task',
        python_callable=print_message,
    )

    task2 = DbtRunOperator(
        task_id='dbt_run_task',
        select='my_model_name'
    )

    task1 >> task2 #execution order
```

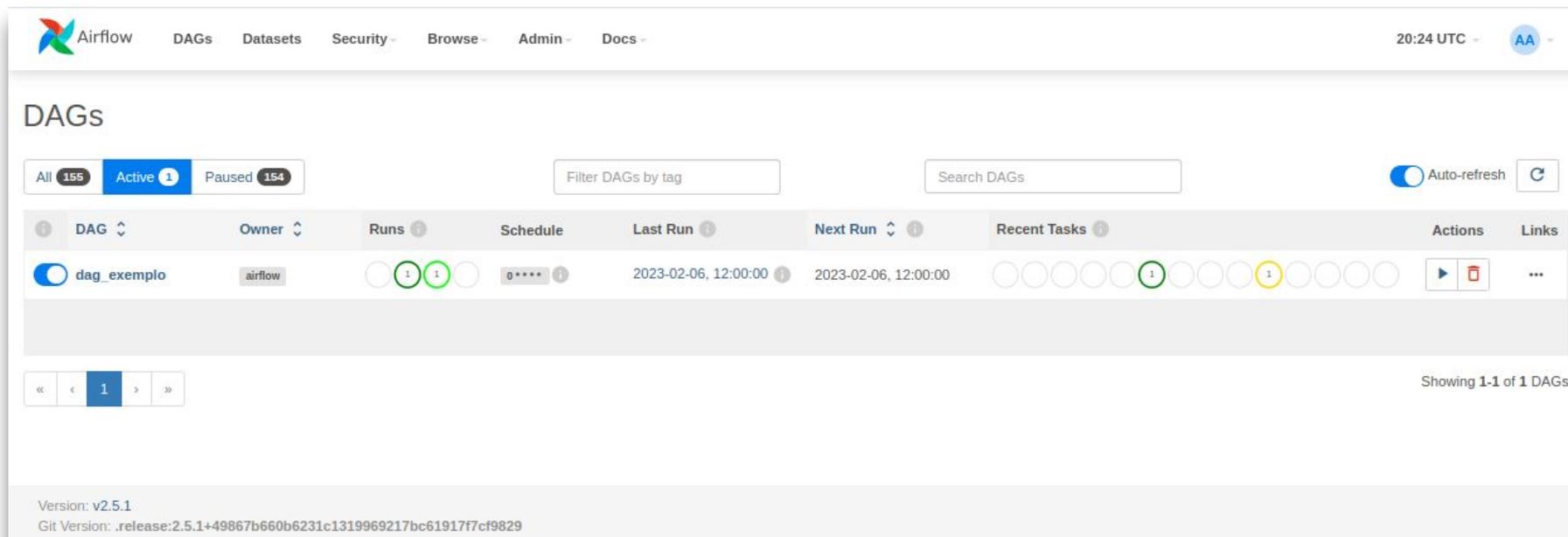


# Orquestrando a execução

- Definir intervalos de execuções;
- Número de tentativas em casos de falhas;
- *Timeout* para finalizar execuções;
- Notificações em casos de falhas;
- Inclusão de sensores para checagens.

```
(...)  
  
with DAG(  
    dag_id="dag_exemplo",  
    start_date=datetime(2023, 2, 1),  
    schedule_interval = "0 * * * *", #@daily  
    default_args = {'retries': 2, 'retry_delay': timedelta(minutes=1)},  
    max_active_runs = 1  
) as dag:  
  
    (...)  
  
    task2 = DbtRunOperator(  
        task_id='dbt_run_task',  
        select='my_model_name',  
        execution_timeout=timedelta(minutes=15),  
    )
```

# UI do airflow



The screenshot displays the Apache Airflow web interface. At the top, there is a navigation bar with the Airflow logo and menu items: DAGs, Datasets, Security, Browse, Admin, and Docs. The current time is 20:24 UTC, and the user is logged in as 'AA'. The main heading is 'DAGs'. Below this, there are filters for 'All' (155), 'Active' (1), and 'Paused' (154). There is also a 'Filter DAGs by tag' input field and a 'Search DAGs' search bar. An 'Auto-refresh' toggle is set to 'on'. The main content is a table of DAGs with the following columns: DAG, Owner, Runs, Schedule, Last Run, Next Run, Recent Tasks, Actions, and Links. The first row shows a DAG named 'dag\_exemplo' owned by 'airflow'. It has 2 successful runs (green circles with '1'), a schedule of '0\*\*\*\*', a last run on 2023-02-06 at 12:00:00, and a next run on 2023-02-06 at 12:00:00. The 'Recent Tasks' column shows 10 task instances, with the first one successful (green circle with '1') and the second one failed (yellow circle with '1'). The 'Actions' column contains a play button and a trash icon. The 'Links' column has a three-dot menu. At the bottom of the table, there is a pagination control showing '1' of 1 DAGs. The footer of the interface displays the version 'v2.5.1' and the Git release hash 'release:2.5.1+49867b660b6231c1319969217bc61917f7cf9829'.

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
<input checked="" type="checkbox"/> dag_exemplo	airflow	<input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>	0****	2023-02-06, 12:00:00	2023-02-06, 12:00:00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/>	<input type="button" value="▶"/> <input type="button" value="🗑️"/>	⋮

# UI do airflow

The screenshot displays the Apache Airflow web interface. At the top, the navigation bar includes 'Airflow', 'DAGs', 'Datasets', 'Browse', 'Admin', and 'Docs'. The current time is 00:35 UTC. The main header shows the DAG name 'datawarehouse' with a 'success' status, a 'Schedule: 0 \*\*\*\*' indicator, and a 'Next Run: 2023-07-18, 00:00:00'.

Below the header, there are several tabs for visualization: 'Grid', 'Graph' (selected), 'Calendar', 'Task Duration', 'Task Tries', 'Landing Times', 'Gantt', 'Details', 'Code', and 'Audit Log'. A search bar 'Find Task...' is also present.

The main content area shows a task graph for the DAG. The graph includes the following tasks:

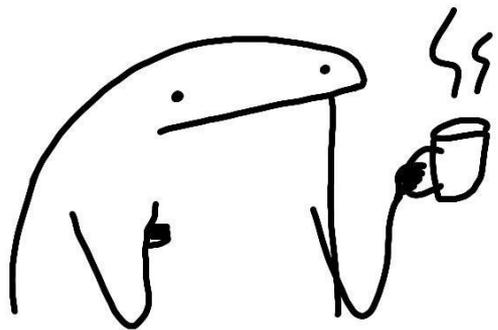
- `BranchPythonOperator` (light blue)
- `DbtRunOperator` (orange)
- `GCSObjectsWithPrefixExistenceSupersededSensor` (light green)
- `GCSObjectsWithPrefixNotUpdatedAfterSensor` (light green)
- `wait_next_hour` (green)
- `wait_while_source_updates` (green)
- `month_changed` (yellow)
- `dbt_month_changed` (red)
- `dbt_run` (orange)

The task graph shows dependencies: `wait_next_hour` and `wait_while_source_updates` both depend on `month_changed`. `month_changed` depends on `dbt_month_changed`, which in turn depends on `dbt_run`. There are also direct dependencies from `wait_next_hour` and `wait_while_source_updates` to `dbt_run`.

At the bottom of the interface, there is a legend for task statuses: deferred, failed, queued, removed, restarting, running, scheduled, shutdown, skipped, success, up\_for\_reschedule, up\_for\_retry, upstream\_failed, and no\_status. An 'Auto-refresh' toggle is also visible.

# Considerações Finais

- **BigQuery + Dbt + airflow** mostrou-se uma combinação perfeita para **Pipelines de dados**.
  - **Boas práticas de engenharia de software** aplicadas no mundo de Analytics.
  - Dados limpos e normalizados **entregues de forma gerenciada** com a ajuda do **Airflow**.
  - **Terabytes de dados processados** nessa infraestrutura.
    - Algumas fontes chegando a processar ~ **1.3 milhões de registros** por hora.
- Grande variedade de **produtos e serviços** consumindo esses dados.
- Diversas soluções de **Machine learning**.
- Diversos **Dashboards**.



# É isso, gente!

Espero que tenham curtido!



[jose.davino@tempest.com.br](mailto:jose.davino@tempest.com.br)



[/in/jose-henrique-davino](https://www.linkedin.com/in/jose-henrique-davino)



Tempest

ACADEMY

Conference

2023

