O que iremos aprender
Na aula de hoje teremos uma apresentação, e o setup do dbt-core. Além disso, entenderemos o uso do dbt para preparação de dados e construção das tabelas e visões especializadas.

Durante a aula realizaremos a configuração de arquivos e pastas iniciais para funcionamento do dbt-core, e a implementação de modelos básicos das tabelas via dbt-core.

O que é o DBT?
O Data Build Tool (dbt) é uma ferramenta poderosa de engenharia de dados que permite aos usuários definir, testar e documentar transformações de dados de forma eficaz e eficiente. Com dbt, os engenheiros de dados podem:

- Definir modelos de dados: Estruturar e organizar modelos que refletem o fluxo lógico e as necessidades analíticas dos negócios.
- Transformar dados: Utilizar SQL para realizar transformações complexas diretamente na fonte de dados.
- Testar transformações: Implementar testes automatizados para verificar a integridade e a precisão dos dados, garantindo que os modelos sejam confiáveis e estejam livres de erros.

Como funciona o dbt
O dbt transforma o processo de engenharia de dados usando SQL, uma linguagem familiar aos analistas de dados, para:

- Extrair e transformar dados: Apesar de não extrair dados para armazenamentos intermediários como datalakes, o dbt opera diretamente nas fontes de dados (bancos de dados, APIs etc.), otimizando o processo de transformação.
- Automação de testes: O dbt compila e executa modelos de dados juntamente com scripts SQL, permitindo testes contínuos e automatizados que garantem a qualidade e a confiabilidade dos dados.
- Execução de consultas: As queries são processadas diretamente na fonte de dados, explorando sua capacidade de processamento e minimizando o transporte de dados.

Benefícios do dbt
O dbt oferece uma série de vantagens que revolucionam a engenharia de dados:

- Padronização e Automatização de Testes: Resolve problemas comuns de inconsistências de SQL através de testes automatizados, proporcionando maior confiabilidade e padronização.
- Reutilização de Código: Facilita a reutilização de modelos e transformações de dados, o que economiza tempo e esforço no desenvolvimento e manutenção de pipelines de dados.
- Documentação e Linhagem de Dados: Gera automaticamente documentação detalhada dos processos, incluindo a linhagem de dados, o que facilita o entendimento e a auditoria das transformações realizadas.

Renomeação e Casting de Dados

- Renomeação de Colunas
    - Exemplo de SQL: SELECT old_column_name AS new_column_name FROM table;
    - Uso no dbt: Adaptação em models/my_model.sql
- Casting de Dados
    - Exemplo de SQL: SELECT CAST(column_name AS new_data_type) FROM table;
    - Importância do casting para garantir integridade dos dados.

Sumarização e Categorização

- Sumarização
    - Uso de funções de agregação como SUM(), AVG(), para calcular totais e médias.
    - Exemplo prático: SELECT product_id, SUM(sales) FROM sales_data GROUP BY product_id;
- Categorização
    - Agrupar dados com base em critérios específicos.
    - Exemplo prático: SELECT CASE WHEN age < 20 THEN 'Youth' WHEN age < 60 THEN 'Adult' ELSE 'Senior' END AS age_group FROM people;

Introdução ao Jinja

- O que é Jinja?
    - Jinja é uma linguagem de template utilizada dentro do dbt para gerar SQL dinâmico.
    - Permite a inserção de lógica condicional, loops e variáveis em consultas SQL.
- Por que Jinja é útil em dbt?
    - Facilita a reutilização de código e a manutenção de projetos de dbt.
    - Permite adaptações dinâmicas das consultas SQL baseadas em parâmetros ou condições específicas.

Loops e Variáveis com Jinja

- Utilizando Variáveis em Jinja
    - Exemplo de Definição de Variável: {% set threshold = 100 %}
    - Uso da Variável em SQL: SELECT * FROM orders WHERE amount > {{ threshold }}
- Utilizando Loops em Jinja
    - Exemplo de Loop para gerar múltiplas queries
    -

```
{% for table in ['customers', 'orders', 'products'] %}
SELECT COUNT(*) FROM {{ table }}
{% endfor %}
```

- Este loop gera uma contagem de linhas para cada tabela listada, criando múltiplas consultas SQL a partir de um template único.

Estruturação de Modelos para Preparação de Dados

- Como Estruturar Modelos em dbt
  - Modelos são definidos em arquivos SQL e configurados com metadados opcionais em YAML.
  - Dependências entre modelos são gerenciadas através de referências explícitas usando a função ref().
- Exemplo de Dependência de Modelo
  - Modelo de pedidos agregados:

```
SELECT date, SUM(amount) AS total_sales
FROM {{ ref('raw_orders') }}
GROUP BY date
```

  - ref('raw_orders') garante que o modelo de pedidos agregados seja construído após a atualização do modelo raw_orders.

Construção de Visões Especializadas

- Definição de Visões Especializadas
  - Visões especializadas são modelos dbt que são configurados para serem visualizados diretamente em ferramentas de BI ou consultas ad-hoc.
  - Exemplo de uma visão especializada para análise de clientes

```
SELECT customer_id, COUNT(order_id) AS total_orders
FROM {{ ref('orders') }}
WHERE order_date >= '2021-01-01'
GROUP BY customer_id
```

O que são Modelos Materializados?

- Definição de Modelos Materializados
  - Modelos materializados são resultados de consultas SQL que são armazenados fisicamente no banco de dados.
  - Reduzem o tempo de carregamento em consultas frequentes e pesadas

Criação de Modelos Materializados para Performance

- Exemplo de Modelo Materializado
  - Modelo para análise de vendas diárias:

```
{{ config(materialized='table') }}
SELECT date, product_id, SUM(quantity) AS total_quantity, SUM(sales) AS total_sales
FROM {{ ref('sales_data') }}
GROUP BY date, product_id
```

  - Este modelo é materializado como uma tabela para acesso rápido e eficiente, ideal para uso em dashboards e relatórios diários.**Hands on**

Instalando o dbt
Antes de tudo, vamos instalar o dbt no python executando o seguinte comando:
pip install dbt-core
No exemplo de hoje instalaremos o dbt-postgres também, porque o exemplo utilizará esse banco de dados:
pip install dbt-postgres

Criando um projeto no dbt
Para criarmos um projeto dbt, conseguimos realizar via terminal, característica que é vista em bibliotecas modernas como great expectation, executando o seguinte comando:
dbt init
Passando então o nome do projeto: exemplo_dbt

```
PS C:\Users\costa\workspace\ada-tech\engenharia-de-dados-cursos\analytics-engineering> cd
 .\aula_3\
PS C:\Users\costa\workspace\ada-tech\engenharia-de-dados-cursos\analytics-engineering\aul
a_3> dbt init
01:51:21  Running with dbt=1.7.11
Enter a name for your project (letters, digits, underscore): exemplo_dbt
01:51:57
Your new dbt project "exemplo_dbt" was created!

For more information on how to configure the profiles.yml file,
please consult the dbt documentation here:

  https://docs.getdbt.com/docs/configure-your-profile

One more thing:

Need help? Don't hesitate to reach out to us via GitHub issues or on Slack:

  https://community.getdbt.com/

Happy modeling!

01:51:57  Setting up your profile.
Which database would you like to use?
[1] duckdb
[2] postgres

(Don't see the one you want? https://docs.getdbt.com/docs/available-adapters)

Enter a number:
```

Configurando o postgres
E então, podemos passar o 2 postgres, passando as seguinte informações

```
01:51:57  Setting up your profile.
Which database would you like to use?
[1] duckdb
[2] postgres

(Don't see the one you want? https://docs.getdbt.com/docs/available-adapters)

Enter a number: 2
host (hostname for the instance): localhost
port [5432]:
user (dev username): postgres
pass (dev password):
dbname (default database that dbt will build objects in): ada
schema (default schema that dbt will build objects in): public
threads (1 or more) [1]: 1
01:53:10  Profile exemplo_dbt written to C:\Users\costa\.dbt\profiles.yml using target's
profile_template.yml and your supplied values. Run 'dbt debug' to validate the connection
```

Estrutura do projeto dbt

Por padrão o dbt criará a seguinte estrutura de projeto

```
dbt_project/
├── dbt_project.yml       # Arquivo de configuração principal do projeto
├── README.md             # Documentação do projeto
├── .gitignore            # Arquivo para ignorar arquivos no controle de versão
├── analyses/             # Pasta para consultas ad-hoc e análises exploratórias
│
├── models/               # Pasta principal para os modelos dbt
│   ├── staging/          # Modelos para a fase de staging dos dados
│   ├── transform/        # Modelos para transformações principais dos dados
│   └── mart/             # Modelos para criação de data marts ou agregações
│
├── data/                 # Pasta para armazenar dados de configuração
│
├── tests/                # Pasta para arquivos de teste dbt YAML
│   ├── schema.yml        # Testes de schema para garantir integridade dos dados
│   └── data.yml          # Testes de dados para verificar precisão das transformações
│
├── macros/               # Pasta para armazenar macros dbt reutilizáveis
│
└── docs/                 # Pasta para documentação adicional
```

Yaml Profile

O nosso exemplo de conexão do dbt com o postgres só foi possível porque temos um arquivo configurado na nossa pasta de usuário

C:\Users\<usuario>\.dbt\

Vamos criar o arquivo nessa pasta com o nome profiles.yml para o exemplo funcione e o configure da seguinte forma

```yaml
exemplo_dbt:
  outputs:
    dev:
      dbname: ada
      host: localhost
      pass: postgres
      port: 5432
      schema: public
      threads: 1
      type: postgres
      user: postgres
  target: dev
```

Caso tudo tenha funcionado, será possível ver que o dbt criara sozinho essa configuração, mas podem ocorrer erros caso o arquivo não exista.

Modelagem do DBT

No nosso exemplo, foram criados dois modelos dentro do arquivo yml schema, responsável por informar os modelos das tabelas, com o seguinte código

```yaml
models:
  - name: my_first_dbt_model
    description: "A starter dbt model"
    columns:
      - name: id
        description: "The primary key for this table"
        tests:
          - unique
          - not_null
```

Nesse caso vamos criar uma tabela no postgres que configuramos no setup, com uma única coluna, e validarmos se ela é única e não nula

```yaml
  - name: my_second_dbt_model
    description: "A starter dbt model"
    columns:
      - name: id
        description: "The primary key for this table"
        tests:
          - unique
          - not_null
```

O mesmo vale a o segundo modelo, mas nesse caso ele é uma view

No arquivo yaml dbt project, podemos visualizar as descrições de como esse projeto dbt vai funcionar

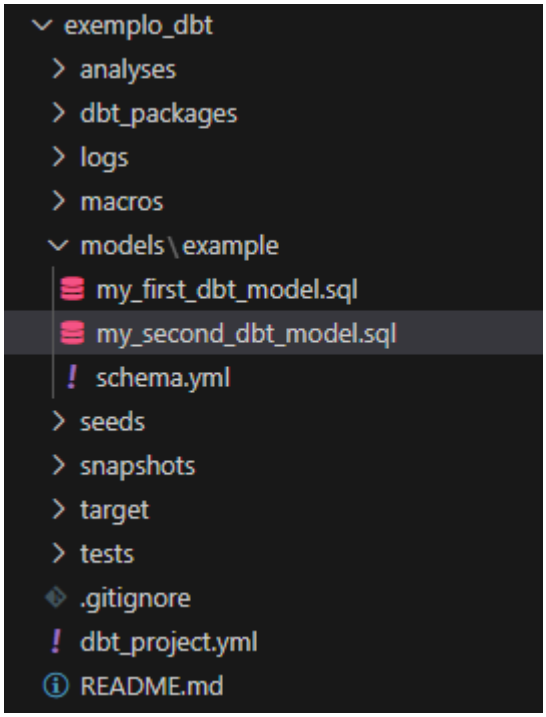
```yaml
# Name your project! Project names should contain only lowercase characters
# and underscores. A good package name should reflect your organization's
# name or the intended use of these models
name: 'exemplo_dbt'
version: '1.0.0'
config-version: 2

# This setting configures which "profile" dbt uses for this project.
profile: 'exemplo_dbt'

# These configurations specify where dbt should look for different types of file
# The `model-paths` config, for example, states that models in this project can
# found in the "models/" directory. You probably won't need to change these!
model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_packages"


# Configuring models
# Full documentation: https://docs.getdbt.com/docs/configuring-models

# In this example config, we tell dbt to build all models in the example/
# directory as views. These settings can be overridden in the individual model
# files using the `{{ config(...) }}` macro.
models:
  exemplo_dbt:
    # Config indicated by + and applies to all files under models/example/
    example:
      +materialized: view
```

As queries em SQL

As queries em sql que irão ser executados para as tabelas são, para a primeira tabela, podemos visualizar que o código força a config a criar esse CTE como uma tabela materializada mesmo que não tenhamos passado essa informação no yaml de dbt project, a tabela se chama my_first_dbt_model

```sql
/*
    Welcome to your first dbt model!
    Did you know that you can also configure models directly within SQL files?
    This will override configurations stated in dbt_project.yml

    Try changing "table" to "view" below
*/

{{ config(materialized='table') }}

with source_data as (

    select 1 as id
    union all
    select null as id

)

select *
from source_data

/*
    Uncomment the line below to remove records with null `id` values
*/

-- where id is not null
```

Para a view que iremos criar, chamada my_second_dbt_model, nós faremos um filtro na tabela materializada, e por padrão, ele a criará como uma view, de acordo com o que passamos no yaml dbt project, para a pasta de modelos “example”



Executando testes

Para executar os testes com o dbt, podemos executar o seguinte comando:

dbt test

```
01:53:54  Running with dbt=1.7.11
01:53:54  Registered adapter: postgres=1.7.11
01:53:55  Unable to do partial parsing because saved manifest not found. Starting full
rse.
01:53:56  Found 2 models, 4 tests, 0 sources, 0 exposures, 0 metrics, 401 macros, 0 gr
s, 0 semantic models
01:53:56
01:53:56  Concurrency: 1 threads (target='dev')
01:53:56
01:53:56  1 of 4 START test not_null_my_first_dbt_model_id .....................
. [RUN]
01:53:56  1 of 4 PASS not_null_my_first_dbt_model_id .....................
. [PASS in 0.05s]
01:53:56  2 of 4 START test not_null_my_second_dbt_model_id .....................
. [RUN]
01:53:56  2 of 4 PASS not_null_my_second_dbt_model_id .....................
. [PASS in 0.08s]
01:53:56  3 of 4 START test unique_my_first_dbt_model_id .....................
. [RUN]
01:53:56  3 of 4 PASS unique_my_first_dbt_model_id .....................
. [PASS in 0.04s]
01:53:56  4 of 4 START test unique_my_second_dbt_model_id .....................
. [RUN]
01:53:56  4 of 4 PASS unique_my_second_dbt_model_id .....................
. [PASS in 0.04s]
01:53:56
01:53:56  Finished running 4 tests in 0 hours 0 minutes and 0.35 seconds (0.35s).
```

Se tudo tiver ok, vamos ver que os testes passaram

Executando o dbt

Para executar o dbt basta rodar o comando

dbt run

```
01:58:05  Running with dbt=1.7.11
01:58:06  Registered adapter: postgres=1.7.11
01:58:06  Found 2 models, 4 tests, 0 sources, 0 exposures, 0 metrics, 401 macros, 0 grou
s, 0 semantic models
01:58:06
01:58:06  Concurrency: 1 threads (target='dev')
01:58:06
01:58:06  1 of 2 START sql table model public.my_first_dbt_model .....................
. [RUN]
01:58:06  1 of 2 OK created sql table model public.my_first_dbt_model ................
. [SELECT 2 in 0.11s]
01:58:06  2 of 2 START sql view model public.my_second_dbt_model .....................
. [RUN]
01:58:06  2 of 2 OK created sql view model public.my_second_dbt_model ................
. [CREATE VIEW in 0.07s]
01:58:06
01:58:06  Finished running 1 table model, 1 view model in 0 hours 0 minutes and 0.34 sec
nds (0.34s).
01:58:06
01:58:06  Completed successfully
01:58:06
01:58:06  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

Como podemos ver, tudo funcionou, vamos abrir um jupyter notebook e validar se as tabelas foram criadas corretamente.

Validando o DBT

Para conectar de forma assertiva, podemos executar a seguinte sequência de comandos que conectará no postgres para validar se as tabelas estão criadas ou não.

Importar a lib sql alchemy, pandas e datetime

```
from sqlalchemy import create_engine, text as sql_text
import pandas as pd
```

Com essas libs importadas no jupyter, conseguiremos criar uma conexão, via create_engine com o banco postgresql local

```
engine = create_engine('postgresql://postgres:ada@localhost/ada')
```

Agora que temos a conexão estabelecida, podemos executar uma query via pandas, passando a engine criada

```
query = """
SELECT *
FROM public.my_first_dbt_model
"""
df = pd.read_sql(sql=sql_text(query), con=engine.connect())
df
query = """
SELECT *
FROM public.my_second_dbt_model
"""
df = pd.read_sql(sql=sql_text(query), con=engine.connect())
df
```

Bom, então podemos verificar que todo o setup funcionou corretamente, viram só pessoal, é muito simples criar tabelas utilizando o dbt

Recapitulação
Nesta aula, aprendemos sobre o dbt, uma ferramenta de construção de dados que permite definir, testar e documentar transformações de dados de forma automatizada. Configuramos um projeto dbt, executamos testes e validamos os resultados no banco de dados. Com o dbt, podemos aumentar a confiança na qualidade dos dados e reduzir a necessidade de validação manual.

Nos vemos na próxima aula!