Especializando dados através do DBT - Data Build Tool

Como já vimos, validar dados é uma tarefa essencial em qualquer projeto de análise de dados. A validação de dados garante que os dados que estamos analisando sejam precisos, confiáveis e relevantes para o nosso negócio. No entanto, a validação manual de dados pode ser uma tarefa demorada e propensa a erros. Felizmente, podemos usar uma ferramenta chamada DBT (Data Build Tool) para automatizar a validação de dados em nossos projetos de análise de dados.

O DBT é uma ferramenta de construção de dados que nos permite definir, testar e documentar nossas transformações de dados. Com esta ferramenta, podemos definir modelos de dados, transformar e preparar dados, e também testar nossas transformações para garantir que nossos dados estejam corretos. Vamos explorar como o DBT pode ajudar a validar dados de forma automatizada e eficiente:

## Primeiros passos com DBT

### Passo 1: Configuração do ambiente

Antes de começarmos, precisamos configurar nosso ambiente de trabalho para utilizar o DBT. Primeiro, precisamos instalar o DBT em nosso computador. Para tanto, podemos usar o gerenciador de pacotes Python pip. Abra o terminal do seu computador e execute o seguinte comando:

pip install dbt-core

Depois de instalar o DBT, precisamos configurar um perfil de conexão com o banco de dados que vamos usar em nosso projeto. O perfil de conexão é um arquivo de configuração que contém as informações de autenticação e conexão com o banco de dados. Podemos configurar um perfil de conexão adicionando um arquivo chamado profiles.yml no diretório .dbt na raiz do nosso projeto, conforme exemplo abaixo:

my\_database:

target: dev

outputs:

dev:

type: postgres

host: localhost

port: 5432

user: myuser

password: mypassword

dbname: mydatabase

Neste exemplo, definimos um perfil de conexão chamado my\_database que se conecta a um banco de dados PostgreSQL em localhost:5432. Também fornecemos as credenciais de autenticação necessárias para se conectar ao banco de dados.

### Passo 2: Definindo modelos de dado

Com o ambiente configurado, agora podemos começar a definir nossos modelos de dados. Os modelos de dados são consultas SQL que definem uma tabela ou visão derivada de outras tabelas ou visões. Podemos definir modelos de dados usando o DBT em um arquivo chamado models.sql no diretório models na raiz do nosso projeto. Segue exemplo:

-- models/customers.sql

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

SELECT

customer\_id,

first\_name,

last\_name,

email

FROM

my\_database.public.customers

WHERE

email IS NOT NULL

Neste exemplo, definimos um modelo de dados chamado customers que seleciona quatro colunas da tabela my\_database.public.customers. Também adicionamos uma cláusula WHERE para filtrar apenas as linhas em que o campo email não é nulo. Por fim, adicionamos a diretiva config(materialized='view') para indicar que este modelo é uma visão em vez de uma tabela física.

### Passo 3: Testando modelos de dados

Depois de definir nossos modelos de dados, precisamos testá-los para garantir que estão produzindo resultados corretos. Podemos usar o DBT para definir testes para nossos modelos de dados em arquivos separados de teste. Os arquivos de teste devem estar localizados no diretório tests na raiz do nosso projeto e devem ter o sufixo \_test.sql. Segue exemplo do arquivo customers\_test.sql:

-- tests/customers\_test.sql

-- Testa se o modelo customers retorna pelo menos 1000 linhas

SELECT

COUNT(\*)

FROM

{{ ref('customers') }}

WHERE

email IS NOT NULL

HAVING

COUNT(\*) >= 1000

Neste exemplo, definimos um teste que conta o número de linhas retornadas pelo modelo customers e verifica se o resultado é maior ou igual a 1000. Usamos a função ref para referenciar o modelo customers definido anteriormente.

### Passo 4: Executando tarefas com DBT

Com nossos modelos e testes definidos, podemos executar 3 tarefas principais com o DBT:

1. dbt run: executa os modelos de dados e cria ou atualiza as tabelas e visões resultantes no banco de dados de destino.
2. dbt test: executa os testes definidos para os modelos de dados e verifica se os resultados são os esperados.
3. dbt docs generate: gera documentação para os modelos de dados em um formato legível.

Podemos executar essas tarefas com o comando dbt, seguido pelo nome da tarefa e o perfil de conexão a ser usado. Por exemplo, para executar os modelos de dados e testes usando o perfil my\_database, podemos executar os seguintes comandos no terminal:

dbt run --profiles-dir .dbt --profile my\_database

dbt test --profiles-dir .dbt --profile my\_database

O primeiro comando executa os modelos de dados e cria ou atualiza as tabelas e visões no banco de dados de destino. O segundo comando executa os testes definidos para os modelos de dados e verifica se os resultados são os esperados.

### Passo 5: Agendando tarefas com DBT

Além de executar tarefas manualmente, podemos agendar as tarefas do DBT para serem executadas automaticamente em intervalos regulares. Podemos fazer isso usando uma ferramenta de agendamento como o cron no Linux ou o Agendador de Tarefas no Windows.

Para agendar uma tarefa do DBT, precisamos definir um comando de agendamento que execute o comando dbt com o nome da tarefa e o perfil de conexão a ser usado. Por exemplo, para agendar a execução dos modelos de dados e testes do DBT a cada hora usando o perfil my\_database, podemos adicionar a seguinte linha ao arquivo de configuração do cron:

0 \* \* \* \* dbt run --profiles-dir /path/to/project/.dbt --profile my\_database && dbt test --profiles-dir /path/to/project/.dbt --profile my\_database

Esta linha agendará a execução dos modelos de dados e testes a cada hora (no início da hora) usando o perfil de conexão my\_database.

### Passo 6: Configuração arquivos e pastas iniciais para funcionamento do dbt-core

Para que o dbt-core funcione corretamente, é necessário criar uma estrutura de pastas e arquivos em seu projeto. Essa estrutura segue uma convenção pré-estabelecida pelo dbt.

A estrutura de pastas e arquivos recomendada pelo dbt é a seguinte:

project/

├── dbt\_project.yml

├── profiles.yml

├── models/

│ ├── schema\_1/

│ │ ├── table\_1.sql

│ │ └── table\_2.sql

│ ├── schema\_2/

│ │ ├── table\_3.sql

│ │ └── table\_4.sql

├── tests/

│ ├── schema\_1/

│ │ ├── test\_1.sql

│ │ └── test\_2.sql

│ ├── schema\_2/

│ │ ├── test\_3.sql

│ │ └── test\_4.sql

├── data/

│ ├── schema\_1/

│ │ └── data.csv

│ ├── schema\_2/

│ │ └── data.csv

├── macros/

│ ├── macro

Nessa estrutura, os arquivos e pastas têm a seguinte finalidade:

* dbt\_project.yml: arquivo de configuração principal do dbt;
* profiles.yml: arquivo de configuração das conexões com os bancos de dados;
* models/: pasta que contém os modelos de dados;
* tests/: pasta que contém os testes;
* data/: pasta que contém os dados de entrada para os modelos de dados;
* macros/: pasta que contém as macros, que são comandos em SQL reutilizáveis.

Para criar a estrutura de pastas e arquivos, basta criar as pastas e arquivos com os nomes e localizações indicados acima.

### Passo 7: Implementação de modelos básicos das tabelas via dbt-core

Agora que temos a estrutura de pastas e arquivos do dbt, podemos implementar os modelos de dados. Vamos criar um modelo de dados para a tabela customers que criamos anteriormente.

Crie um arquivo SQL na pasta models/ com o nome customers.sql. O conteúdo do arquivo deve ser o seguinte:

-- models/customers.sql

{{ config(

materialized='table',

schema='my\_schema',

alias='customers'

) }}

SELECT

id,

name,

email,

created\_at

FROM {{ source('my\_source', 'customers') }}

Neste exemplo, definimos um modelo de dados chamado customers que seleciona as colunas id, name, email e created\_at da tabela customers na fonte de dados my\_source. Usamos a função config para definir o materialized como table, o schema como my\_schema e o alias como customers.

### Passo 8: Modelos de materialização customizados

Os modelos de materialização do dbt são responsáveis por definir como os dados serão armazenados nos bancos de dados. O dbt fornece alguns modelos de materialização padrão, como tabelas e visões, mas também permite que sejam criados modelos de materialização customizados. Isso pode ser útil, por exemplo, para criar tabelas particionadas no BigQuery ou Redshift.

Para criar um modelo de materialização customizado, basta definir um arquivo SQL na pasta macros/ com o nome do modelo de materialização. Em seguida, é possível usar o modelo de materialização customizado nos modelos de dados usando a diretiva {{ config(materialized='nome\_do\_modelo\_de\_materializacao') }}.

Por exemplo, para criar um modelo de materialização customizado para criar uma tabela particionada no BigQuery, podemos definir o seguinte arquivo bigquery\_partitioned\_table.sql na pasta macros/:

{% macro bigquery\_partitioned\_table() %}

DISTSTYLE EVEN

DISTKEY (id)

SORTKEY (created\_at)

PARTITION BY RANGE(created\_at)

INTERVAL 1 DAY

(

START (DATE '2020-01-01')

END (DATE '2030-01-01')

EVERY (INTERVAL 1 DAY)

)

{% endmacro %}

Em seguida, podemos usar o modelo de materialização customizado nos modelos de dados:

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

create table my\_table

{{ dbt\_macro('bigquery\_partitioned\_table') }}

as (

select \* from my\_source\_table

)

### Passo 9: Documentação de modelos

A documentação de modelos é uma funcionalidade útil para documentar o propósito e o funcionamento dos modelos de dados. O dbt permite adicionar comentários em Markdown nos modelos de dados, que podem ser exibidos na interface do usuário do dbt ou em um site de documentação de dados gerado pelo dbt.

Para adicionar documentação a um modelo de dados, basta adicionar um comentário em Markdown logo abaixo da diretiva {{ config(...) }} no arquivo SQL do modelo.

Por exemplo:

{{ config(

materialized='view',

schema='my\_schema',

unique\_key='id'

) }}

-- Documentação do modelo

--

-- Este modelo calcula a média de um conjunto de dados.

select avg(value) as average

from my\_table

### Passo 10: Transformações e modificações de dados

O dbt permite realizar transformações e modificações de dados usando a diretiva {{ do(...) }}. Essa diretiva permite executar uma instrução SQL ou uma macro definida em um arquivo SQL na pasta macros/.

Por exemplo, para atualizar uma tabela com os dados mais recentes de uma fonte de dados, podemos usar a seguinte diretiva do no arquivo SQL do modelo:

{{ do('''

delete from my\_table where created\_at > (select max(created\_at) from my\_source\_table);

insert into my\_table select \* from my\_source\_table;

''') }}

Isso executa uma instrução SQL que apaga os registros da tabela my\_table que foram criados após a data mais recente na tabela my\_source\_table e insere os dados mais recentes da tabela my\_source\_table na tabela my\_table.

### Passo 11: Criação de pacotes

O dbt permite criar pacotes que contêm modelos de dados, macros, documentação e outros arquivos necessários para criar e manter pipelines de dados. Esses pacotes podem ser distribuídos e reutilizados em projetos diferentes.

Para criar um pacote, basta criar uma pasta com o nome do pacote na pasta packages/ e adicionar os arquivos necessários. O dbt automaticamente detecta e carrega os pacotes.

Por exemplo, para criar um pacote que contém um modelo de dados e uma macro, podemos criar a seguinte estrutura de arquivos:

packages/

my\_package/

macros/

my\_macro.sql

models/

my\_model.sql

README.md

Em seguida, podemos usar o modelo e a macro em outros modelos de dados:

{{ config(

schema='my\_schema',

materialized='view'

) }}

-- Documentação do modelo

--

-- Este modelo usa a macro my\_macro para calcular a média de um conjunto de dados.

select {{ dbt\_macro('my\_macro', 'my\_table', 'value') }} as average

from my\_table

### Outras funcionalidades do DBT

Além das funcionalidades que já foram apresentadas, o dbt possui outras características e funcionalidades que são úteis para o desenvolvimento de pipelines de dados:

1. Variáveis: o dbt permite definir variáveis que podem ser usadas nos modelos de dados e macros. As variáveis são definidas no arquivo dbt\_project.yml e podem ser usadas nos arquivos SQL usando a sintaxe {{ var('nome\_da\_variavel') }}.
2. Atualizações incrementais: o dbt permite a criação de atualizações incrementais para os modelos de dados. Com as atualizações incrementais, o dbt compara os dados de origem com os dados de destino e aplica apenas às mudanças necessárias para atualizar os dados de destino. Isso pode reduzir significativamente o tempo de processamento em pipelines de dados que lidam com grandes volumes de dados.
3. Ambientes: o dbt permite definir ambientes de desenvolvimento, teste e produção, o que facilita a configuração de perfis de conexão diferentes para cada ambiente. Os arquivos de configuração podem ser definidos separadamente para cada ambiente, permitindo a criação de pipelines de dados que funcionem em diferentes ambientes sem necessidade de alterar os arquivos de configuração manualmente.

Essas são apenas algumas das funcionalidades e características do dbt. O dbt é uma ferramenta poderosa e flexível para o desenvolvimento de pipelines de dados e pode ser personalizado para atender a várias necessidades de processamento de dados.

## Avançando em validações automatizadas via dbt

Nesta seção, discutiremos algumas técnicas de validação de dados via dbt, que são úteis para garantir que os dados em um banco de dados estejam em um formato apropriado e possam ser usados para análises. Em particular, discutiremos técnicas de renomeação, casting, sumarizações e categorizações. Vamos utilizar tanto Python quanto SQL para demonstrar como essas técnicas podem ser aplicadas em prática, lembrando que são utilizadas em conjunto com um banco de dados funcional e configurado em seu ambiente.

### Renomeação

A técnica de renomeação é útil quando os nomes das colunas em um conjunto de dados precisam ser alterados para que sejam mais fáceis de entender ou estejam em um formato específico. Por exemplo, pode ser necessário renomear colunas para seguir uma convenção de nomenclatura de sua equipe ou para garantir que todas as colunas tenham o mesmo tipo de nome.

Vamos supor que temos um conjunto de dados chamado orders, com as seguintes colunas:

order\_id

customer\_id

order\_date

total\_amount

Se quisermos renomear as colunas para seguir uma convenção de nomenclatura específica, podemos usar a seguinte sintaxe em SQL:

SELECT

order\_id AS order\_identifier,

customer\_id AS customer\_identifier,

order\_date AS order\_time,

total\_amount AS total\_value

FROM orders;

Isso renomeará as colunas order\_id, customer\_id, order\_date e total\_amount para order\_identifier, customer\_identifier, order\_time e total\_value, respectivamente.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

SELECT

order\_id AS order\_identifier,

customer\_id AS customer\_identifier,

order\_date AS order\_time,

total\_amount AS total\_value

FROM {{ ref('orders') }}

Note que usamos a função ref() para se referir à tabela orders em nosso banco de dados.

### Casting

A técnica de casting é útil quando precisamos converter o tipo de dados de uma coluna para outro tipo. Isso pode ser necessário quando o tipo de dados original não é adequado para o tipo de análise que queremos fazer. Por exemplo, se uma coluna de uma tabela estiver armazenando datas como strings, pode ser necessário convertê-las para o tipo de dados date para realizar cálculos de tempo.

Vamos supor que a coluna order\_date na tabela orders esteja armazenada como uma string. Podemos usar a seguinte sintaxe em SQL para converter essa coluna para o tipo de dados date:

SELECT

order\_id,

customer\_id,

CAST(order\_date AS DATE) AS order\_date,

total\_amount

FROM orders;

Isso converterá a coluna order\_date para o tipo de dados date.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

SELECT

order\_id,

customer\_id,

CAST(order\_date AS DATE) AS order\_date,

total\_amount

FROM {{ ref('orders') }}

Observe que estamos usando a função CAST() para converter a coluna order\_date para o tipo de dados date.

### Sumarizações

A técnica de sumarização é útil quando precisamos agregar dados de uma ou mais colunas. Por exemplo, podemos querer calcular o valor médio de uma coluna, o número total de linhas em uma tabela ou o número de ocorrências de valores únicos em uma coluna.

Vamos supor que queremos calcular o valor médio do campo total\_amount na tabela orders e, para tanto, podemos usar a seguinte sintaxe em SQL:

SELECT

AVG(total\_amount) AS average\_total\_amount

FROM orders;

Isso nos dará o valor médio do campo total\_amount.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

SELECT

AVG(total\_amount) AS average\_total\_amount

FROM {{ ref('orders') }}

### Categorizações

A técnica de categorização é útil quando precisamos agrupar valores em uma ou mais colunas em categorias específicas. Por exemplo, podemos querer categorizar valores numéricos em faixas específicas ou agrupar valores de texto em categorias com base em um padrão específico.

Vamos supor que queremos categorizar o valor do campo total\_amount na tabela orders em três categorias: "baixo" (valores menores do que 100), "médio" (valores entre 100 e 500) e "alto" (valores maiores do que 500). Podemos usar a seguinte sintaxe em SQL:

SELECT

CASE

WHEN total\_amount < 100 THEN 'baixo'

WHEN total\_amount BETWEEN 100 AND 500 THEN 'médio'

ELSE 'alto'

END AS total\_amount\_category,

COUNT(\*) AS count

FROM orders

GROUP BY total\_amount\_category;

Isso nos dará o número de pedidos em cada categoria.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

SELECT

CASE

WHEN total\_amount < 100 THEN 'baixo'

WHEN total\_amount BETWEEN 100 AND 500 THEN 'médio'

ELSE 'alto'

END AS total\_amount\_category,

COUNT(\*) AS count

FROM {{ ref('orders') }}

GROUP BY total\_amount\_category;

## Usando jinja no DBT

Agora, iremos explorar a utilização de loops e variáveis através do Jinja no dbt. O Jinja é uma linguagem de template que permite escrever código SQL dinâmico, o que pode ser especialmente útil quando estamos trabalhando com modelos e precisamos gerar SQL com base em variáveis ou loops.

Antes de começarmos, é importante lembrar que o Jinja é uma linguagem de template e, portanto, não pode ser usada diretamente em um ambiente Python ou SQL. No entanto, o dbt usa Jinja como uma linguagem de template para gerar SQL, então é importante estar familiarizado com os conceitos de Jinja para aproveitar ao máximo o dbt.

### Variáveis no Jinja

Uma variável é um espaço reservado para armazenar um valor que pode ser usado em uma expressão ou declaração. No Jinja, as variáveis são denotadas por duas chaves ({{ }}). Podemos atribuir um valor a uma variável usando o operador de atribuição (=).

Vamos supor que queremos gerar um SQL dinâmico para selecionar registros de uma tabela com base em uma variável. Podemos usar a seguinte sintaxe em Jinja:

{% set min\_order\_date = '2022-01-01' %}

SELECT \*

FROM orders

WHERE order\_date >= '{{ min\_order\_date }}';

Observe que estamos atribuindo o valor '2022-01-01' à variável min\_order\_date. Em seguida, estamos usando essa variável em uma expressão SQL para selecionar registros com data de pedido igual ou posterior à data armazenada na variável.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

{% set min\_order\_date = '2022-01-01' %}

SELECT \*

FROM {{ ref('orders') }}

WHERE order\_date >= '{{ min\_order\_date }}';

Observe que estamos usando o Jinja para definir a variável min\_order\_date e depois usando a mesma variável em uma expressão SQL para selecionar registros com data de pedido igual ou posterior à data armazenada na variável.

### Loops no Jinja

Um loop é uma estrutura de controle que permite executar uma ação repetidamente com base em uma condição. No Jinja, os loops são denotados pela palavra-chave for. Podemos usar loops para gerar SQL dinâmico com base em uma lista de valores ou objetos.

Vamos supor que queremos gerar um SQL dinâmico para selecionar registros de uma tabela para cada mês do ano. Podemos usar a seguinte sintaxe em Jinja:

{% for month in ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12'] %}

SELECT \*

FROM orders

WHERE order\_date >= '2022-{{ month }}-01'

AND order\_date < '2022-{{ month|int + 1 }}-01';

{% endfor %}

Observe que estamos usando um loop for para iterar sobre cada mês do ano. Em seguida, estamos usando o valor de cada mês em uma expressão SQL para selecionar registros com data de pedido dentro do mês.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

{% for month in ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12'] %}

SELECT \*

FROM {{ ref('orders') }}

WHERE order\_date >= '2022-{{ month }}-01'

AND order\_date < '2022-{{ month|int + 1 }}-01';

{% endfor %}

Observe que estamos usando um loop for para iterar sobre cada mês do ano. Em seguida, estamos usando o valor de cada mês em uma expressão SQL para selecionar registros com data de pedido dentro do mês.

### Combinação de loops e variáveis

Podemos combinar loops e variáveis para gerar SQL dinâmico ainda mais poderoso. Vamos supor que queremos gerar um SQL dinâmico para selecionar registros de uma tabela para cada mês do ano, mas desta vez, também queremos agrupar os resultados por mês. Podemos usar a seguinte sintaxe em Jinja:

{% for month in ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12'] %}

{% set start\_date = '2022-' + month + '-01' %}

{% set end\_date = '2022-' + month|int + 1 ~ '-01' %}

SELECT '{{ month }}' as month, \*

FROM orders

WHERE order\_date >= '{{ start\_date }}'

AND order\_date < '{{ end\_date }}';

{% endfor %}

Observe que estamos usando um loop for para iterar sobre cada mês do ano. Em seguida, estamos definindo as variáveis start\_date e end\_date com base no valor de cada mês. Finalmente, estamos usando essas variáveis em uma expressão SQL para selecionar registros com data de pedido dentro do mês e agrupando os resultados por mês.

Em dbt, podemos aplicar a mesma técnica da seguinte forma em um arquivo chamado orders.sql:

-- orders.sql

{% for month in ['01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '11', '12'] %}

{% set start\_date = '2022-' + month + '-01' %}

{% set end\_date = '2022-' + month|int + 1 ~ '-01' %}

SELECT '{{ month }}' as month, \*

FROM {{ ref('orders') }}

WHERE order\_date >= '{{ start\_date }}'

AND order\_date < '{{ end\_date }}';

{% endfor %}

Observe que estamos usando o Jinja para definir as variáveis start\_date e end\_date com base no valor de cada mês e, em seguida, usando essas variáveis em uma expressão SQL para selecionar registros com data de pedido dentro do mês e agrupando os resultados por mês.

O uso de loops e variáveis através do Jinja no dbt permite que escrevamos SQL dinâmico de uma forma muito mais poderosa e flexível. Os exemplos apresentados nesta aula podem ser facilmente adaptados para atender às necessidades de uma ampla variedade de cenários de análise de dados. À medida que você se familiarizar com o uso de Jinja no dbt, você encontrará muitas outras maneiras de aproveitar ao máximo esta ferramenta poderosa.

# Referências

* Documentação dbt: <https://docs.getdbt.com/>
* Usando Jinja no dbt: <https://docs.getdbt.com/docs/get-started/learning-more/using-jinja>
* Overview dbt: <https://www.analytics8.com/blog/dbt-overview-what-is-dbt-and-what-can-it-do-for-my-data-pipeline/>
* Livro virtual Automate the Boring Stuff with Python: <https://automatetheboringstuff.com/>
* Playlist Python no canal da Ada no Youtube: <https://www.youtube.com/watch?v=cDqMbI02hRs&list=PLcmVV8telDGzH6wFY_9h_KZ3Sn7EfjT1D>