Tutorial DBT

Autor: **Thiago Vilarinho Lemes**
Data: **03/08/2025**
Descrição: **está apostila é apenas um guia básico para que o desenvolvedor possa tirar dúvidas.**
LinkedIn: **https://www.linkedin.com/in/thiago-v-lemes-b1232727/**

O dbt é usado principalmente para:

1. **Transformação de dados em ambientes analíticos (ELT):**
   - Após os dados serem carregados no data warehouse (como BigQuery, Snowflake, Redshift ou PostgreSQL), o dbt realiza a **limpeza, padronização, junções, agregações e cálculos**.
2. **Modelagem de dados com SQL modular:**
   - Usa arquivos .sql organizados em modelos (models/) para construir tabelas e visualizações reutilizáveis.
3. **Testes automatizados de dados:**
   - Permite definir testes como:
     - Verificar **unicidade**, **ausência de nulos**, **valores esperados**, etc.
     - Ex: unique, not_null, accepted_values
4. **Documentação viva e navegável:**
   - Gera documentação com links entre modelos, descrições de colunas e gráficos de dependência interativos.
   - Comando: dbt docs generate && dbt docs serve
5. **Uso de versionamento com Git e execução em CI/CD:**
   - Permite integração fácil em pipelines de deploy com controle de versão.
6. **Reutilização com ref() e source():**
   - O ref() permite conectar modelos entre si.
   - O source() conecta tabelas brutas do banco ao projeto.
   -

Benefícios de usar dbt:

- SQL limpo, modular e reutilizável
- Pipeline versionado com Git
- Testes automáticos para garantir integridade dos dados
- Documentação atualizável e com gráfico de dependências
- Suporte a múltiplos bancos analíticos (Postgres, BigQuery, etc.)
- Integração com ferramentas como Airflow, Dagster, Prefect, dbt Cloud

Comandos Básicos

dbt run	# Executa todos os modelos
dbt test	# Roda testes definidos nos schema.yml
dbt seed	# Carrega arquivos CSV da pasta seeds para o banco
dbt debug	# Verifica conexão e ambiente
dbt docs generate	# Gera a documentação em HTML
dbt docs serve	# Abre um servidor local para visualizar a documentação
dbt build	# Executa run + test + seed + snapshot (pipeline completo)

Passos para projeto:

1. Iniciar o projeto com o comando:
   - **dbt init nome_projeto**
2. Em seguida verificar o arquivo **dbt_project.yml** que está localizado na raiz do projeto, abaixo temos um modelo:

dbt_project.yml

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

# Perfil de conexão a ser utilizado
profile: 'dbt_etl_cliente'

# Caminhos dos modelos, sementes, testes, etc.
model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

# Configurações para o target 'dev'
target-path: "target"  # diretório para compilação
clean-targets:         # diretórios a serem limpos com 'dbt clean'
  - "target"
  - "dbt_packages"

# Configuração dos modelos
models:
  etl_pipeline_dbt_estudo:
    +schema: staging
    # Definição dos subdiretórios e materializações
    staging:
      materialized: view
    transform:
      materialized: table
    reporting:
      materialized: table
```

Este **name** será utilizado no arquivo **profiles.yml**

3. Logo após verificar o arquivo, deve ser criado um arquivo com o nome **profiles.yml** na raiz do projeto, este arquivo que fará conexão com o banco de dados.

OBS.: O name que está no arquivo **dbt_project.yml** será utilizado para criar o **profiles.yml,** abaixo temos um exemplo:

profiles.yml

```yaml
dbt_etl_cliente:
  target: dev
  outputs:
    dev:
      type: postgres
      host: localhost
      user: nome_usuario_postgres
      password: senha_postgres
      dbname: nome_tabela
      schema: public
      threads: 1
      port: 5432
```

Mesmo **name** utilizado no arquivo **dbt_project.yml**

4. Para testar a conexão com o banco de dados, utilize o comando abaixo:
   - **dbt debug**

   Se a conexão for bem sucedida irá retornar: **Connection test: [OK connection ok]**
5. Criando a estrutura pastas do models:

```
dbt_etl_cliente
├── models/
│   ├── reporting
│   │   └── vw_clientes.sql
│   ├── staging
│   │   └── stg_clientes.sql
│   │   └── schema.yml
│   ├── transform
│   │   └── dim_clientes.sql
│   │   └── schema.yml
├── dbt_project.yml
├── profiles.yml  (configurações para conectar no Postgres)
```

Para que servem as três camadas:

- **Staging**: padroniza e expõe os dados brutos de forma limpa.
- **Transform**: aplica lógica de negócio e agregações.
- **Reporting**: organiza modelos finais para visualização ou exportação.

models/reporting/vw_clientes.sql

```sql
with vw_clientes as (
    select * from {{ ref('dim_clientes') }}
)

select
    ano_cadastro,
    count(cliente_id) as total_clientes
from vw_clientes
group by ano_cadastro
order by ano_cadastro
```

models/staging/stg_clientes.sql

```sql
{{ config(
    materialized='view',
    schema='staging'
) }}

with source as (
    select
        id as cliente_id,
        nome,
        email,
        data_cadastro
    from {{ source('public', 'clientes') }}

),
renamed as (
    select
        cliente_id,
        nome,
        email,
        data_cadastro
    from source
)
select * from renamed
```

models/staging/schema.yml

```yaml
version: 2

sources:
  - name: public
    schema: public
    tables:
      - name: clientes

models:
  - name: stg_clientes
    description: "Staging da tabela cliente"
    columns:
      - name: cliente_id
        tests:
          - not_null
          - unique
      - name: email
        tests:
          - not_null
```

models/transform/dim_clientes.sql

```sql
with stg_clientes as (
    select * from {{ ref('stg_clientes') }}
)

select
    cliente_id,
    nome,
    email,
    data_cadastro,
    extract(year from data_cadastro) as ano_cadastro
from stg_clientes
```

6. Rode o comando abaixo:
   - **dbt run**
7. Em seguida rode o test para verificar os dados:
   - **dbt test**

Como Usar Seeds:
Crie arquivos **CSV** dentro da pasta **`data/`** ou **`seeds/`** e rode o comando **`dbt seed`** para importar para o banco.

Como usar ref() e source():

- Usando ref() para referenciar models:
  select * from {{ ref('stg_products') }}
- Usando source() para tabelas externas:
  select * from {{ source('northwind', 'products') }}

Criação dos arquivos .sql

Ao inserir no cabeçalho do arquivo **{{ config(materialized='view', schema='silver') }}** será gerado somente um **schema** com a **view** corresponde ao código.

Caso seja colocado **{{ config(materialized=table, schema='silver') }}** será gerado a tabela correspondente ao código.

Gerando a documentação:

Para gerar a documentação execute o comando abaixo:

- **dbt docs generate && dbt docs serve**

Instalando Pacotes

Para instalar um pacote crie na raiz do projeto um arquivo chamado **packages.yml**, nele será inserido as bibliotecas para serem instaladas.

Segue um exemplo para instalar um pacote:

packages.yml

```yaml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.3.0
```

Para instalar uma biblioteca utilize o comando baixo:

- **$ dbt deps**

Dicas e Boas Práticas

- Separe bem as camadas (staging, transform, reporting);
- Use snake_case(letras minúsculas) para tudo;
- Documente tudo no schema.yml;
- Use dbt build para rodar o pipeline completo.