Contents

Guida Completa a dbt (data build tool): Dalle Basi all'Analytics Engineering Avanzato

Autore: Esperto Tecnico e Divulgatore
Argomento: dbt (data build tool)
Livello: Principiante - Avanzato

---

1. Introduzione

Cos'è dbt?

dbt (data build tool) è un framework di sviluppo open-source che consente agli analisti dati e agli ingegneri di trasformare i dati nel proprio warehouse in modo efficace. A differenza degli strumenti ETL tradizionali (Extract, Transform, Load) che spesso gestiscono l'estrazione e il caricamento, dbt si concentra esclusivamente sulla **T** (Transform) nei flussi di lavoro **ELT** (Extract, Load, Transform).

La Filosofia di dbt

dbt nasce con un obiettivo preciso: portare le best practice dell'ingegneria del software nel mondo dei dati. Questo approccio è noto come **Analytics Engineering**. Le pratiche chiave introdotte da dbt includono:

- **Version Control:** Tutto il codice è gestito tramite Git.
- **Testing Automatizzato:** I test sulla qualità dei dati sono scritti ed eseguiti insieme al codice.
- **Documentazione Dinamica:** La documentazione è generata automaticamente dal codice.
- **Modularità:** Il codice SQL è riutilizzabile e modulare (DRY - Don't Repeat Yourself).

Perché è importante?

Prima di dbt, la logica di trasformazione dei dati era spesso sepolta in stored procedures complesse, script Python non versionati o viste SQL ingestibili. dbt democratizza l'ingegneria dei dati permettendo a chiunque conosca l'SQL di costruire pipeline di dati di livello produttivo, garantendo al contempo governance e affidabilità.

2. Concetti Chiave

Per padroneggiare dbt, è fondamentale comprendere i componenti che costituiscono un progetto.

2.1. Il Progetto dbt (`dbt_project.yml`)

Ogni istanza di dbt è un "progetto". Il file `dbt_project.yml` è il cuore della configurazione: definisce il nome del progetto, la versione, i percorsi delle cartelle e le configurazioni globali per i modelli.

2.2. Models (Modelli)

Un **model** in dbt è semplicemente un file `.sql` che contiene una singola istruzione `SELECT`. * Non c'è bisogno di scrivere DDL (Create Table/View). dbt gestisce la creazione degli oggetti nel database. * Un modello rappresenta una tabella logica o una vista nel data warehouse.

2.3. Materializations (Materializzazioni)

La materializzazione definisce *come* dbt costruisce il modello nel database. Le principali sono:

Tipo	Descrizione	Pro	Contro
View	Crea una vista (`CREATE VIEW`). Default.	Veloce da creare, dati sempre freschi.	Lenta da interrogare se la logica è complessa.
Table	Crea una tabella fisica (`CREATE TABLE AS`).	Veloce da interrogare.	Lenta da costruire, richiede rigenerazione completa.
Incremental	Aggiorna la tabella esistente aggiungendo/modificando solo le righe nuove.	Molto efficiente per grandi dataset.	Configurazione complessa, rischio di drift dei dati.
Ephemeral	Non crea oggetti nel DB. È una CTE (Common Table Expression) riutilizzabile.	Mantiene pulito il warehouse.	Può rendere il debug difficile se usata troppo.

2.4. Il grafo delle dipendenze (DAG) e la funzione `ref()`

La caratteristica più potente di dbt è la gestione automatica delle dipendenze. Invece di scrivere `FROM nome_schema.nome_tabella`, si usa la funzione Jinja `{{ ref('nome_modello') }}`. * dbt compila il codice sostituendo `ref` con il nome corretto della tabella. * Costruisce un **DAG** (Directed Acyclic Graph) per determinare l'ordine di esecuzione corretto (es. esegue prima le tabelle di staging, poi quelle finali).

2.5. Jinja e Macros

dbt usa **Jinja2**, un linguaggio di templating per Python, all'interno dell'SQL. Questo permette di: * Usare strutture di controllo (`if`, `for`). * Usare variabili d'ambiente. * Creare **Macro**: funzioni riutilizzabili (es. una macro per convertire valute o gestire formati data) che possono essere chiamate come funzioni in qualsiasi modello.

2.6. Seeds e Sources

- **Seeds:** File CSV (piccoli, come tabelle di decodifica o elenchi di paesi) che dbt carica nel database e tratta come tabelle.
- **Sources:** Tabelle "grezze" caricate da strumenti esterni (come Fivetran o Stitch). Definirle in dbt permette di tracciare la lineage (provenienza) dei dati fin dall'origine.

3. Setup e Configurazione Iniziale

Prima di scrivere codice, vediamo come si configura un ambiente di lavoro.

Requisiti

- Python installato.
- Un Data Warehouse (PostgreSQL, Snowflake, BigQuery, Redshift, Databricks, DuckDB, etc.).
- Git (opzionale ma raccomandato).

Installazione

```
pip install dbt-core dbt-postgres
# Nota: installare l'adattatore specifico per il proprio DB (es. dbt-snowflake)
```

Inizializzazione

```
dbt init nome_progetto
```

Questo comando crea la struttura delle cartelle: * `/models`: Dove risiede il codice SQL. * `/seeds`: Dove mettere i CSV statici. * `/tests`: Test custom. * `/analyses`: Query analitiche che non creano tabelle. * `/snapshots`: Per gestire la storicizzazione (SCD Type 2).

Configurazione Connessione (`profiles.yml`)

dbt cerca un file chiamato `profiles.yml` (solitamente nella home dell'utente `~/.dbt/`) per connettersi al DB.

```yaml
# Esempio per PostgreSQL
my_dbt_profile:
  target: dev
  outputs:
    dev:
      type: postgres
      host: localhost
      user: my_user
      password: my_password
      port: 5432
      dbname: analytics
      schema: dbt_dev
      threads: 4
```

4. Esempi Pratici

Immaginiamo di lavorare per un e-commerce. Abbiamo dati grezzi di ordini e clienti e vogliamo creare una tabella analitica pulita.

Esempio 1: Definizione di una Source

Creiamo un file `models/staging/sources.yml` per dichiarare da dove arrivano i dati grezzi.

```yaml
version: 2

sources:
  - name: raw_ecommerce
    database: raw_db
    schema: public
    tables:
      - name: raw_orders
      - name: raw_customers
```

Esempio 2: Modello di Staging (Base)

Creiamo un modello per pulire i clienti: `models/staging/stg_customers.sql`.

```sql
WITH source AS (
    -- Uso la funzione source per riferirmi ai dati grezzi
    SELECT * FROM {{ source('raw_ecommerce', 'raw_customers') }}
),

renamed AS (
    SELECT
        id AS customer_id,
        first_name,
        last_name,
        email,
        -- Logica di pulizia: gestiamo i null
        COALESCE(status, 'active') AS status
    FROM source
)

SELECT * FROM renamed
```

Esempio 3: Modello "Mart" (Finale)

Creiamo una tabella che unisce clienti e ordini: `models/marts/customers_orders.sql`. Qui usiamo `ref()` per riferirci al modello di staging creato sopra.

```sql
{{ config(materialized='table') }} -- Questo modello sarà una tabella fisica

WITH customers AS (
    SELECT * FROM {{ ref('stg_customers') }}
),

orders AS (
    SELECT * FROM {{ ref('stg_orders') }} -- Ipotizziamo esista anche questo
),

customer_orders AS (
```

```
    SELECT
        customer_id,
        MIN(order_date) AS first_order_date,
        MAX(order_date) AS most_recent_order_date,
        COUNT(order_id) AS number_of_orders,
        SUM(amount) AS lifetime_value
    FROM orders
    GROUP BY 1
),

final AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        co.first_order_date,
        co.lifetime_value
    FROM customers c
    LEFT JOIN customer_orders co USING (customer_id)
)

SELECT * FROM final
```

5. Testing e Documentazione

Una delle parti più rivoluzionarie di dbt è l'integrazione nativa di test e documentazione.

5.1. Test Generici (Schema Tests)

Si definiscono nel file YAML associato ai modelli. dbt offre 4 test nativi: `unique`, `not_null`, `accepted_values`, `relationships` (integrità referenziale).

File `models/marts/schema.yml`:

```yaml
version: 2

models:
  - name: customers_orders
    description: "Tabella riepilogativa dei clienti e del loro valore."
    columns:
      - name: customer_id
        description: "Chiave primaria del cliente."
        tests:
          - unique
          - not_null

      - name: lifetime_value
        tests:
          - not_null

      - name: status
        tests:
          - accepted_values:
              values: ['active', 'inactive', 'churned']
```

Per eseguire i test:

```
dbt test
```

Se un test fallisce, dbt segnala l'errore e la pipeline può essere interrotta.

5.2. Documentazione

Avendo aggiunto le descrizioni nel file YAML sopra, possiamo generare un sito web statico che documenta tutto il nostro data warehouse, inclusa la lineage visiva.

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

Questo comando apre un sito web locale dove è possibile esplorare il dizionario dati e vedere graficamente da dove provengono i dati di ogni colonna.

6. Funzionalità Avanzate

6.1. Modelli Incrementali

Quando le tabelle diventano enormi (miliardi di righe), rigenerarle da zero ogni giorno è impossibile. I modelli incrementali aggiornano solo le novità.

Esempio `models/marts/fct_events.sql`:

```
{{
    config(
        materialized='incremental',
        unique_key='event_id'
    )
}}

SELECT
    event_id,
    event_type,
    event_timestamp,
    user_id
FROM {{ source('raw_ecommerce', 'events') }}

-- Logica speciale per dbt:
{% if is_incremental() %}
  -- Questa clausola viene eseguita solo nelle run successive alla prima
  WHERE event_timestamp > (SELECT MAX(event_timestamp) FROM {{ this }})
{% endif %}
```

6.2. Snapshots (SCD Type 2)

Come tracciare i cambiamenti storici se il database sorgente sovrascrive i dati (es. un cliente cambia indirizzo e non abbiamo lo storico)? dbt usa gli **snapshots**.

File `snapshots/customers_snapshot.sql`:

```
{% snapshot customers_snapshot %}

{{
    config(
      target_database='analytics',
      target_schema='snapshots',
      unique_key='id',
      strategy='timestamp',
      updated_at='updated_at',
    )
}}

SELECT * FROM {{ source('raw_ecommerce', 'raw_customers') }}

{% endsnapshot %}
```

Quando si esegue `dbt snapshot`, dbt crea colonne `dbt_valid_from` e `dbt_valid_to` per gestire la storicizzazione.

6.3. Pacchetti (Packages)

Come in Python (pip) o Node (npm), dbt ha un gestore di pacchetti. Il più famoso è `dbt-utils`, che contiene macro avanzate per date, pivot, e SQL generators.

File `packages.yml`:

```yaml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1
```

Esegui `dbt deps` per installarli.

7. Casi d'Uso e Best Practices

Quando usare dbt?

1. **Migrazione al Cloud:** Passaggio da DB on-premise a Snowflake/BigQuery.
2. **Pulizia del codice:** Quando si hanno centinaia di script SQL ingestibili senza dipendenze chiare.
3. **Data Quality:** Quando i dati arrivano rotti alla dashboard e si vuole intercettare l'errore prima.
4. **Self-Service BI:** Creare modelli "Marts" puliti e documentati che gli analisti di business possono usare in Tableau/PowerBI senza dover fare join complesse.

Struttura consigliata dei layer

Un progetto dbt maturo segue solitamente questa struttura a tre livelli:

1. **Staging:**
   - Materializzazione: View.
   - Scopo: Rinominare colonne, cast dei tipi di dato, pulizia leggera. Relazione 1:1 con le sorgenti.
2. **Intermediate:**
   - Materializzazione: View o Ephemeral.
   - Scopo: Logica di business complessa, join intermedie, raggruppamenti specifici.
3. **Marts (o Core):**
   - Materializzazione: Table o Incremental.
   - Scopo: Tabelle pronte per la BI (Business Intelligence). Schemi a stella (Facts & Dimensions) o tabelle larghe (OBT - One Big Table).

8. Conclusione

dbt è diventato lo standard *de facto* per la trasformazione dei dati nei moderni stack tecnologici. Ha spostato il potere dalle mani di pochi ingegneri ETL specializzati a una platea più ampia di analisti che conoscono SQL.

Imparare dbt non significa solo imparare un nuovo tool, ma adottare una metodologia di lavoro più robusta, collaborativa e scalabile.

Prossimi Passi

1. Installa dbt e connettilo a un database di test (DuckDB è ottimo per iniziare in locale senza costi).
2. Segui il corso gratuito "dbt Fundamentals" sul sito ufficiale.
3. Prova a convertire una vecchia query SQL complessa in una serie di modelli dbt modulari.

Fine del documento.