Chapitre 4:

dbt: Data Build Tool

4DATA

Benjamin Jurczak

SUPINFO

Sommaire

1. Introduction à dbt
2. Architecture et fonctionnement de dbt
3. Orchestration et automatisation avec dbt
4. Bonnes pratiques et déploiement

1. Introduction à dbt

1. Introduction à dbt

dbt: Révolutionner la Transformation des Données

- dbt signifie **Data Build Tool**
- Créé par **Fishtown Analytics** (maintenant dbt Labs) en 2016
- Objectif initial : simplifier la transformation des données dans les entrepôts cloud
- Open-source (DBT Core) avec une version payante (DBT Cloud)
- Popularité croissante dans les architectures **ELT modernes**

1. Introduction à dbt

Pourquoi DBT ? Comparaison avec ETL traditionnels

- Approche classique **ETL (Extract - Transform - Load)**
  - Transformation avant le chargement des données dans le data warehouse
  - Coût élevé en termes d'infrastructure
  - Maintenance complexe

- Approche **ELT (Extract - Load - Transform) avec DBT**
  - Extraction et chargement des données brutes dans un entrepôt cloud
  - Transformation directement dans l'entrepôt via **SQL**
  - Exploitation de la puissance des bases de données modernes
  - Versioning, collaboration et documentation inclus

1. Introduction à dbt

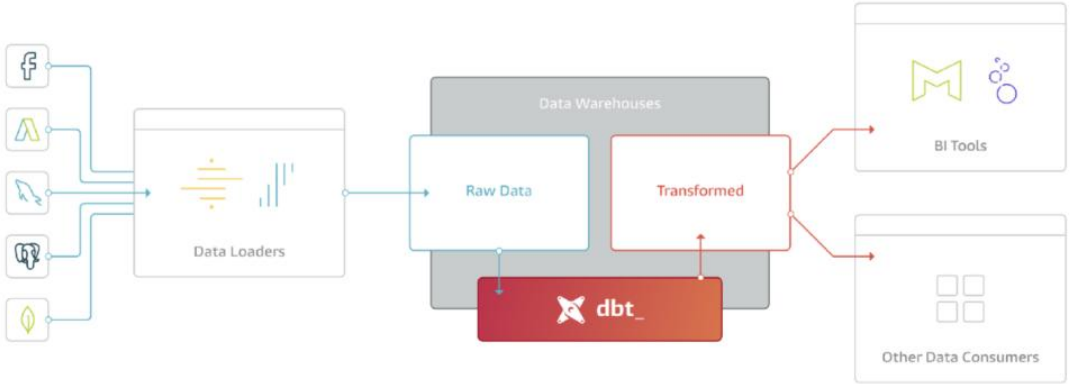
Différences entre DBT Core et DBT Cloud

Fonctionnalité	**dbt Core**	**dbt Cloud**
Type	Open Source	Saas payant
Exécution	Ligne de commande	Interface web
Intégration Git	Manuelle	Native
Planification	Non supportée	Planification auto.
Support technique	Communautaire	Professionnel

1. Introduction à dbt

Cas d’usage de DBT dans un pipeline ELT

- **Nettoyage et transformation des données** (filtrage, agrégation, enrichissement)
- **Standardisation et modélisation des données** (modèles en couches : raw, staging, marts)
- **Documentation et tests automatisés**
- **Gestion des dépendances** entre modèles de données
- **Optimisation des performances** via des materializations



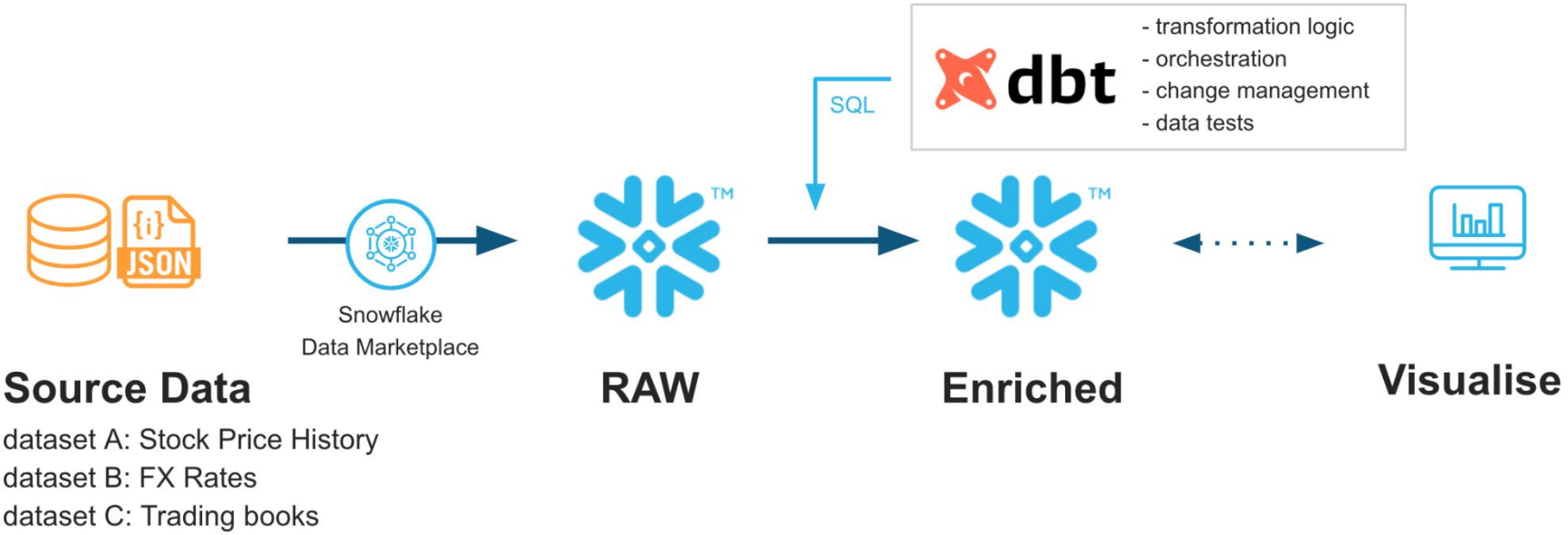
1. Introduction à dbt

Écosystème DBT : Jinja, SQL, Git, Airflow, Data Warehouses

- **SQL** : Langage principal pour la transformation
- **Jinja** : Moteur de templates pour automatiser SQL
- **Git** : Versioning et collaboration
- **Airflow** : Orchestration et planification
- **Data Warehouses** : Snowflake, BigQuery, Redshift, Databricks, etc.

1. Introduction à dbt

Principe de Fonctionnement



1. Introduction à dbt

2. Architecture et fonctionnement de dbt

2. Architecture et fonctionnement de dbt

Problème des Pipelines de Données Traditionnels

SQL utilisé de manière **non standardisée** : scripts SQL dispersés dans divers outils
Difficulté à **maintenir et auditer** les transformations
Problèmes de **qualité des données** et d'**intégrité**
Processus de transformation souvent **géré par des ETL propriétaires***

—> dbt comme Solution

Standardisation des transformations SQL
Automatisation des tests et documentation
Gestion des dépendances et exécution optimisée
Intégration native avec les entrepôts modernes (BigQuery, Snowflake, Redshift, Postgres)

💡 **dbt transforme SQL en un framework modulaire, documenté et testé pour la gestion des données analytiques.**

2. Architecture et fonctionnement de dbt

Principe de Fonctionnement (rappel)

dbt est un **moteur de transformation SQL** qui s'exécute directement sur un **entrepôt de données**. Il suit un modèle **ELT** (*Extract - Load - Transform*) où la transformation est effectuée **après** le chargement des données brutes dans l'entrepôt.

- **Schéma de Fonctionnement dbt**

RAW DATA (données brutes) ⟶ dbt ⟶ TRANSFORMATIONS SQL ⟶ ANALYTICS-READY DATA

💡 **Contrairement aux outils ETL, dbt n'extrait ni ne charge les données. Il effectue uniquement la transformation.**

2. Architecture et fonctionnement de dbt

Composants Clés d’un Projet dbt

Un projet dbt est **structuré en plusieurs dossiers** pour organiser et gérer efficacement les transformations SQL.

📂 **Structure Standard d’un Projet dbt**

```
my_dbt_project/
├── dbt_project.yml        # Configuration principale du projet
├── profiles.yml           # Connexions à l’entrepôt de données
├── models/                # Contient les transformations SQL
│   ├── staging/           # Transformation des données brutes
│   ├── marts/             # Modèles analytiques finaux
├── tests/                 # Vérifications de la qualité des données
├── macros/                # Fonctions SQL réutilisables
├── snapshots/             # Historisation des données
├── logs/                  # Fichiers de log des exécutions dbt
├── target/                # Résultats des exécutions (compilation des requêtes SQL)
```

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Comment DBT transforme les données ?

- DBT applique des **modèles SQL** pour transformer les données dans un entrepôt de données.
- Chaque modèle est un fichier SQL qui définit une transformation.
- DBT exécute ces modèles dans un ordre dépendant de leurs références (ref()).
- Utilisation de la puissance de calcul du Data Warehouse (Snowflake, BigQuery, Redshift, etc.).
- Séparation des couches : **Raw** → **Staging** → **Marts** pour structurer les transformations.

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Principe de transformation des données avec DBT

Exemple de modèle DBT : Fichier : models/staging/stg_orders.sql

```
WITH raw_orders AS (
    SELECT * FROM {{ source('ecommerce', 'orders') }}
)
SELECT
    order_id,
    customer_id,
    order_date,
    total_amount
FROM raw_orders
WHERE status = 'completed'
```

Ce modèle extrait uniquement les commandes **finalisées** depuis une table source orders.

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Exécution d'un modèle DBT

- dbt run est la commande principale qui exécute les transformations définies dans les modèles SQL.
- À chaque exécution :
  1. DBT **analyse** les modèles et génère des requêtes SQL optimisées.
  2. Il **exécute ces requêtes** directement dans le Data Warehouse.
  3. Il **stocke les résultats** en fonction de la matérialisation choisie (view, table, incremental).
- Exemple de commande :

  dbt run
- Exécution d'un modèle spécifique :

  dbt run --select stg_orders
- Exécution d'un ensemble de modèles en fonction de leur dépendance :

  dbt run --select tag:finance
- Résultat attendu : nouvelles tables/vues mises à jour dans le Data Warehouse.

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Compilation dans DBT

- dbt compile génère le SQL final de chaque modèle **sans l'exécuter**.
- Permet de **prévisualiser** les requêtes avant de les exécuter.
- Tous les fichiers SQL compilés sont stockés dans le dossier target/.

Exemple de commande :

dbt compile

Cela génère les fichiers SQL prêts à être exécutés, accessibles dans target/compiled/.

Affichage d'un fichier SQL compilé :

cat target/compiled/my_project/models/staging/stg_orders.sql

Cela permet de voir le SQL final généré par DBT après interprétation des fonctions comme source() et ref().

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Gestion des dépendances avec ref()

- ref() est une fonction qui permet de lier un modèle à un autre.
- Elle garantit que les modèles sont exécutés dans le bon ordre.

```
SELECT
    o.order_id,
    o.customer_id,
    o.total_amount,
    c.customer_name
FROM {{ ref('stg_orders') }} o
JOIN {{ ref('stg_customers') }} c
ON o.customer_id = c.customer_id
```

Exemple d'utilisation de ref() : Fichier : models/marts/mart_sales.sql

Cela garantit que stg_orders et stg_customers seront exécutés avant mart_sales.

Exécution avec dépendances :

dbt run --select mart_sales+

Cela exécutera mart_sales ainsi que tous les modèles dont il dépend.

2. Architecture et fonctionnement de dbt

Fonctionnement général et workflow de DBT

Débogage et logs

- Vérification de la configuration et des connexions avec dbt debug :
  dbt debug
  - Vérifie l'accès au Data Warehouse et la configuration des fichiers.
- Activation du mode **verbose** pour voir les requêtes SQL exécutées :
  dbt run --debug
- Logs détaillés enregistrés dans logs/dbt.log.
- En cas d'erreur :
  tail -f logs/dbt.log
  - Permet d'analyser les erreurs et de comprendre les échecs d'exécution.

2. Architecture et fonctionnement de dbt

Installation et configuration d'un projet DBT

Installation de DBT

- DBT est un outil basé sur Python, installable via pip.
- Commande pour installer DBT :
  pip install dbt-core
- DBT nécessite un adaptateur pour se connecter au Data Warehouse (PostgreSQL, BigQuery, Snowflake, Redshift, etc.).

  Exemple d'installation d'un adaptateur :
  pip install dbt-postgres # Pour PostgreSQL

  pip install dbt-bigquery # Pour BigQuery

  pip install dbt-snowflake # Pour Snowflake

- Vérification de l'installation : ***dbt --version***

2. Architecture et fonctionnement de dbt

Installation et configuration d'un projet DBT

Configuration du fichier de connexion

- profiles.yml contient les informations de connexion au Data Warehouse.
- Emplacement : ~/.dbt/profiles.yml (Linux/Mac) ou %USERPROFILE%\.dbt\profiles.yml (Windows).
- Exemple de configuration pour PostgreSQL :

```yaml
my_project:
  target: dev
  outputs:
    dev:
      type: postgres
      host: my-database.example.com
      user: my_user
      password: my_password
      port: 5432
      dbname: my_database
      schema: analytics
```

Pour tester la connexion : dbt debug

2. Architecture et fonctionnement de dbt

Installation et configuration d’un projet DBT

Initialisation d’un projet DBT

- Un projet DBT est initialisé avec la commande :
  dbt init my_project
- dbt_project.yml contient la configuration du projet.
- Les modèles SQL sont organisés dans models/.
- Les macros et tests personnalisés sont placés dans macros/ et tests/.

2. Architecture et fonctionnement de dbt

Installation et configuration d'un projet DBT

Structuration d'un projet DBT

- **Séparer les modèles en différentes couches** :
  - models/raw/ : Données brutes (sources)
  - models/staging/ : Nettoyage et transformation initiale
  - models/marts/ : Modèles finaux pour l'analyse
- **Utiliser des conventions de nommage** :
  - Préfixer les modèles de staging avec stg_
  - Préfixer les modèles finaux avec mart_
- **Documentation et tests** :
  - Ajouter des descriptions aux colonnes dans les fichiers YAML
  - Tester les modèles avec dbt test

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Modèles SQL (.sql) et leur exécution

- Un **modèle DBT** est un fichier .sql contenant une requête SQL qui transforme les données.
- Lors de l'exécution (dbt run), DBT génère et exécute les requêtes SQL sur le Data Warehouse.
- Un modèle peut être matérialisé en **table, vue ou incrémental**.

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Exemple de modèle simple (models/staging/stg_orders.sql) :

```
SELECT
    order_id,
    customer_id,
    order_date,
    total_amount
FROM {{ source('ecommerce', 'orders') }}
WHERE status = 'completed'
```

- Ce modèle récupère uniquement les commandes finalisées depuis la table source orders.
- Peut être exécuté avec :

  dbt run --select stg_orders

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Utilisation des fonctions ref() et source()

`ref()` et `source()`

- `source()` permet de référencer une table brute depuis le Data Warehouse.
- `ref()` permet d'établir une dépendance entre modèles DBT.

Déclaration d'une source (`models/sources.yml`) :

```
version: 2
sources:
  - name: ecommerce
    database: raw
    schema: public
    tables:
      - name: orders
```

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Utilisation des fonctions ref() et source()

Utilisation dans un modèle : ***SELECT * FROM {{ source('ecommerce', 'orders') }}***

Utilisation de `ref()` pour lier un modèle (`models/marts/mart_sales.sql`) :

```
SELECT
    o.order_id,
    o.customer_id,
    c.customer_name,
    o.total_amount
FROM {{ ref('stg_orders') }} o
JOIN {{ ref('stg_customers') }} c
ON o.customer_id = c.customer_id
```

- `ref()` assure que `stg_orders` et `stg_customers` sont exécutés avant `mart_sales`.
- Exécution de `mart_sales` avec toutes ses dépendances : ***dbt run --select mart_sales+***

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Gestion des dépendances et ordonnancement des modèles

Orchestration des modèles

- DBT génère un **DAG** (Directed Acyclic Graph) basé sur les dépendances définies par ref().
- Les modèles sont exécutés dans un ordre optimisé pour éviter les erreurs de dépendance.
- Visualisation des dépendances :

dbt docs generate

dbt docs serve

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Optimisation des requêtes SQL avec DBT

Bonnes pratiques d'optimisation

Utiliser **les materializations appropriées** :

- **Vue (view)** : rapide, mais recalculée à chaque exécution.
- **Table (table)** : persistée dans l'entrepôt, meilleure performance.
- **Incrémentale (incremental)** : optimise le temps de mise à jour en chargeant uniquement les nouvelles données.

2. Architecture et fonctionnement de dbt

Création et gestion des modèles dans DBT

Optimisation des requêtes SQL avec DBT

Exemple de modèle incrémental (models/incremental/incr_orders.sql) :

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

SELECT
    order_id,
    customer_id,
    order_date,
    total_amount
FROM {{ source('ecommerce', 'orders') }}
WHERE status = 'completed'
{% if is_incremental() %}
    AND order_date > (SELECT MAX(order_date) FROM {{ this }})
{% endif %}
```

- **Seules les nouvelles commandes sont ajoutées lors de l'exécution.**
- **Exécution optimisée :**
  dbt run --select incr_orders

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Vue (view) : Avantages et limitations

Matérialisation en vue (view)

- Une vue est une requête **stockée** dans le Data Warehouse qui s'exécute à chaque consultation.
- Avantages :
  - Exécution rapide pour de petites transformations.
  - Facile à maintenir car les données sont toujours à jour.
- Limitations :
  - Recalcul à chaque requête, pouvant impacter les performances sur de gros volumes de données.
  - Dépend de la performance du moteur SQL sous-jacent.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Exemple d'un modèle matérialisé en view :

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

SELECT
    order_id,
    customer_id,
    total_amount
FROM {{ source('ecommerce', 'orders') }}
WHERE status = 'completed'
```

Ce modèle génère une vue recalculée à chaque interrogation.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Table (table) : Quand l'utiliser ?

Matérialisation en table (table)

- Une table est **persistée** dans le Data Warehouse et **ne change pas** tant que dbt run n'est pas exécuté.
- Avantages :
  - Temps de requêtage rapide car les données sont stockées physiquement.
  - Idéal pour les tables analytiques utilisées fréquemment.
- Limitations :
  - Peut nécessiter un **rechargement complet** lors de l'exécution.
  - Augmente la consommation d'espace de stockage.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Exemple d'un modèle matérialisé en table :

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

SELECT
    order_id,
    customer_id,
    total_amount
FROM {{ ref('stg_orders') }}
```

Cette table est mise à jour uniquement lorsque dbt run est exécuté.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Ephemeral (ephemeral) : Transformations en mémoire

Matérialisation éphémère (ephemeral)

- Un modèle **éphémère** est une transformation **exécutée en mémoire** sans stocker les résultats.
- Avantages :
  - Évite la création de tables temporaires inutiles.
  - Optimisation pour des sous-requêtes souvent utilisées.
- Limitations :
  - Peut ralentir les requêtes imbriquées car chaque exécution génère une nouvelle requête.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Exemple de modèle éphémère

```
{{ config(
    materialized='ephemeral'
) }}

SELECT
    order_id,
    customer_id,
    total_amount
FROM {{ ref('stg_orders') }}
```

Ce modèle **ne crée pas de table physique** mais injecte directement son SQL dans les modèles qui l'utilisent.

2. Architecture et fonctionnement de dbt

Materializations : comment DBT stocke les données

Stratégies avancées d'incrémentation des données (`unique_key`, `merge`, `delete+insert`)

Pour éviter les doublons, DBT propose plusieurs stratégies avancées :

- **Utilisation de `unique_key` pour mettre à jour les données existantes :**
  - Permet de mettre à jour une ligne si `customer_id` existe déjà.

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

- **Stratégie `merge` (pour Snowflake, BigQuery, etc.) :**
  - Permet une mise à jour efficace sans supprimer et recréer toute la table.

```
{{ config(
    materialized='incremental',
    unique_key='order_id',
    incremental_strategy='merge'
) }}
```

- **Stratégie `delete+insert` (Redshift, Postgres) :**
  - Supprime les lignes existantes avant d'insérer les nouvelles.

```
{{ config(
    materialized='incremental',
    unique_key='order_id',
    incremental_strategy='delete+insert'
) }}
```

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Déclaration des sources (sources.yml)

Définition des sources dans DBT

- DBT permet de déclarer les sources de données brutes via un fichier YAML.
- Cela facilite la traçabilité des données et leur gouvernance.
- La déclaration d'une source inclut :
    - Le nom de la base de données et du schéma.
    - La liste des tables disponibles.

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Déclaration des sources (sources.yml)

Exemple de déclaration d'une source dans `sources.yml` :

```yaml
version: 2
sources:
  - name: ecommerce
    database: raw
    schema: public
    tables:
      - name: orders
      - name: customers
```

Une source peut être utilisée dans un modèle avec la fonction `source()` :

```
SELECT * FROM {{ source('ecommerce', 'orders') }}
```

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Snapshots pour gérer l'historisation des données (dbt snapshot)

Gestion des changements avec DBT Snapshots

- DBT permet de **suivre l'évolution des données** dans le temps grâce aux snapshots.
- Un snapshot crée une table qui stocke les différentes versions des lignes modifiées.
- Permet d'analyser les changements historiques et de détecter les mises à jour.

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Snapshots pour gérer l'historisation des données (`dbt snapshot`)

Exemple de snapshot (`snapshots/orders_snapshot.sql`) :

```
{% snapshot orders_snapshot %}

{{ config(
    target_schema='snapshots',
    unique_key='order_id',
    strategy='timestamp',
    updated_at='updated_at'
) }}

SELECT * FROM {{ source('ecommerce', 'orders') }}

{% endsnapshot %}
```

- Cette commande enregistre chaque mise à jour de la table `orders` en fonction du champ `updated_at`.
- Exécution du snapshot : ***dbt snapshot***

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Suivi des modifications des enregistrements (`strategy: timestamp` vs `strategy: check`)

Deux stratégies principales pour capturer les changements dans les snapshots :

- **`strategy: timestamp`** : Suit les changements en fonction d'un champ de type date/heure (`updated_at`).
- Utilisé lorsque les données ont une colonne indiquant la date de mise à jour.
- Exemple dans `snapshots.yml` :

```
strategy: timestamp
updated_at: updated_at
```

2. Architecture et fonctionnement de dbt

Gestion des sources et snapshots

Suivi des modifications des enregistrements (`strategy: timestamp` vs `strategy: check`)

Deux stratégies principales pour capturer les changements dans les snapshots :

- **`strategy: check`** : Compare toutes les colonnes pour détecter un changement.
- Utile si la table ne dispose pas d'un champ `updated_at`.
- Exemple dans `snapshots.yml` :

```
strategy: check
check_cols: ['customer_name', 'total_amount']
```

Le choix de la stratégie dépend du format des données sources et des besoins en historisation.

2. Architecture et fonctionnement de dbt

3. Orchestration et automatisation avec dbt

3. Orchestration et automatisation avec dbt

Orchestration avec Dagster

Rappels sur Dagster et son architecture

- Dagster est un orchestrateur moderne permettant d'exécuter et de surveiller des workflows de données.
- Dagster utilise une approche **basée sur les assets et le typage des données**.
- Il permet une meilleure modularité et un suivi précis des transformations.
- Intégration native avec DBT pour orchestrer des transformations de manière efficace.

3. Orchestration et automatisation avec dbt

Orchestration avec Dagster

Utilisation des jobs et assets DBT dans Dagster

- Dans Dagster, les tâches DBT sont définies comme des assets ou des ops.
- Exemple de définition d’un asset DBT :

```
from dagster_dbt import load_assets_from_dbt_project

DBT_PROJECT_DIR = "/path/to/dbt/project"
dbt_assets = load_assets_from_dbt_project(DBT_PROJECT_DIR)
```

Permet de suivre l’état des modèles DBT et leurs dépendances.

3. Orchestration et automatisation avec dbt

Orchestration avec Dagster

Gestion des dépendances entre tâches Dagster et DBT

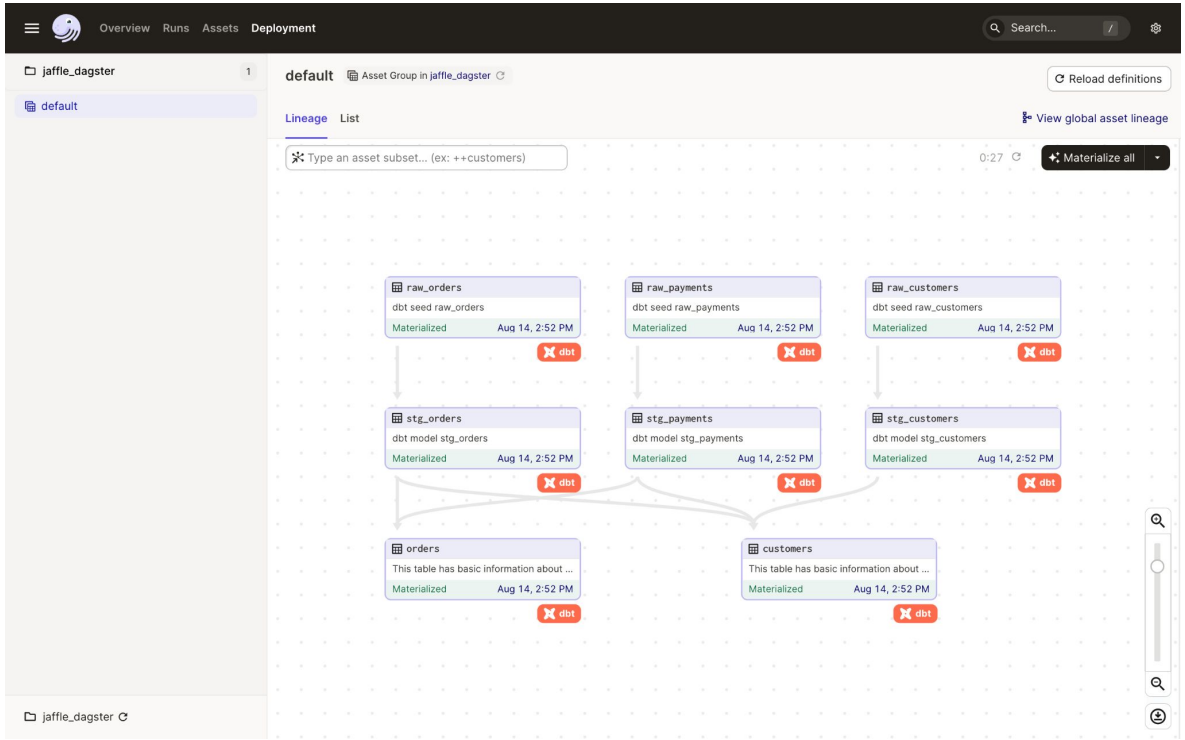
- Dagster permet de définir des **dépendances explicites** entre assets DBT.
- Exemple d'enchaînement d'assets :

```
@asset(deps=["stg_orders"])
def mart_sales(context):
    context.log.info("Transformation des ventes complétée")
```

Cela garantit que les transformations sont exécutées dans le bon ordre.

3. Orchestration et automatisation avec dbt

Orchestration avec Dagster



3. Orchestration et automatisation avec dbt

Orchestration avec Dagster

A découvrir lors du TP…

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Création et gestion des environnements DBT

- Les environnements DBT permettent d'exécuter des transformations dans dev, staging et prod.
- Configurer plusieurs environnements dans profiles.yml :

```
my_project:
  target: dev
  outputs:
    dev:
      type: postgres
      host: localhost
      schema: dev
    prod:
      type: postgres
      host: prod-db
      schema: analytics
```

Exécuter DBT dans un environnement spécifique :

dbt run --target prod

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Utilisation des variables d’environnement dans DBT

- Les variables peuvent être utilisées dans dbt_project.yml pour adapter les modèles :

```
vars:
  reporting_period: "2024-01-01"
```

Exécution avec une variable dynamique :

```
dbt run --vars '{"reporting_period": "2024-02-01"}'
```

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Stratégies de versioning et meilleures pratiques Git

- Suivre une **stratégie Git structurée** :
  - main : Version stable en production.
  - develop : Version en cours de développement.
  - feature-branches : Développement de nouvelles fonctionnalités.
- Utiliser des **Pull Requests** pour valider les modifications DBT avant intégration.
- Exemple de validation d'un modèle avant merge :

```
dbt test --select new_model
```

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Intégration de DBT avec CI/CD

Déploiement continu avec GitHub Actions/GitLab CI/CD

- Automatiser l'exécution de DBT après chaque merge sur main.
- Exemple de workflow GitHub Actions (.github/workflows/dbt.yml) :

```yaml
name: Deploy DBT
on: [push]

jobs:
  dbt_run:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
      - name: Setup DBT
        run: pip install dbt-core dbt-postgres
      - name: Run DBT
        run: dbt run
```

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Intégration de DBT avec CI/CD

Déploiement continu avec GitHub Actions/GitLab CI/CD

Automatisation des tests et de la documentation

- Ajouter une étape de tests dans CI/CD :

```
- name: Test DBT
  run: dbt test
```

- Générer et publier la documentation automatiquement :

```
- name: Generate Docs
  run: dbt docs generate
```

3. Orchestration et automatisation avec dbt

Gestion des environnements et du versioning

Intégration de DBT avec CI/CD

Meilleures pratiques pour un workflow CI/CD optimisé

- Exécuter dbt run uniquement sur les modèles modifiés :

  dbt run --select state:modified

- Ajouter un **contrôle qualité automatique** pour valider les changements avant mise en production.
- Surveiller les logs de CI/CD pour anticiper les erreurs d'exécution.

3. Orchestration et automatisation avec dbt

4. Bonnes pratiques et déploiement

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Objectif : Assurer la fiabilité et la transparence des modèles DBT.

Tests et validation des modèles

Pourquoi tester les modèles dans DBT ?

- Garantir l’intégrité et la cohérence des données transformées.
- Éviter la propagation d’erreurs en production.
- Automatiser la validation des transformations de données.
- DBT fournit des tests intégrés et permet d’ajouter des tests personnalisés.

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Tests intégrés dans DBT

DBT offre plusieurs types de tests **prêts à l'emploi** :

- not_null : Vérifie qu'une colonne ne contient pas de valeurs nulles.
- unique : Vérifie qu'une colonne ne contient pas de doublons.
- accepted_values : Vérifie qu'une colonne contient uniquement certaines valeurs définies.
- relationships : Vérifie les relations entre les tables (ex: clé étrangère).

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Exemple de tests intégrés (models/schema.yml)

```yaml
version: 2
models:
  - name: stg_orders
    columns:
      - name: order_id
        tests:
          - not_null
          - unique
      - name: order_status
        tests:
          - accepted_values:
              values: ['completed', 'pending', 'cancelled']
```

Exécution des tests : ***dbt test***

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Création de tests personnalisés avec des macros

- DBT permet d'écrire des tests personnalisés via des **macros Jinja**.
- Exemple de test personnalisé (`macros/test_positive.sql`) :

```
{% test test_positive(model, column_name) %}
SELECT *
FROM {{ model }}
WHERE {{ column_name }} < 0
{% endtest %}
```

Application du test dans `schema.yml` :

```
columns:
  - name: total_amount
    tests:
      - test_positive
```

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Débogage et gestion des erreurs (dbt test)

- DBT affiche les erreurs dans la console et dans logs/dbt.log.

- Exécuter les tests avec plus de détails :

  dbt test --select stg_orders --debug

- Affichage des résultats des tests :

  dbt test --store-failures

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Documentation et génération des rapports

Documentation des modèles DBT

- Chaque modèle peut être documenté dans schema.yml.
- Ajouter des descriptions pour chaque colonne :

```yaml
models:
  - name: stg_orders
    description: "Table contenant les commandes avec statut filtré."
    columns:
      - name: order_id
        description: "Identifiant unique de la commande."
      - name: customer_id
        description: "Référence du client ayant passé la commande."
```

4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Documentation et génération des rapports

Génération interactive de la documentation

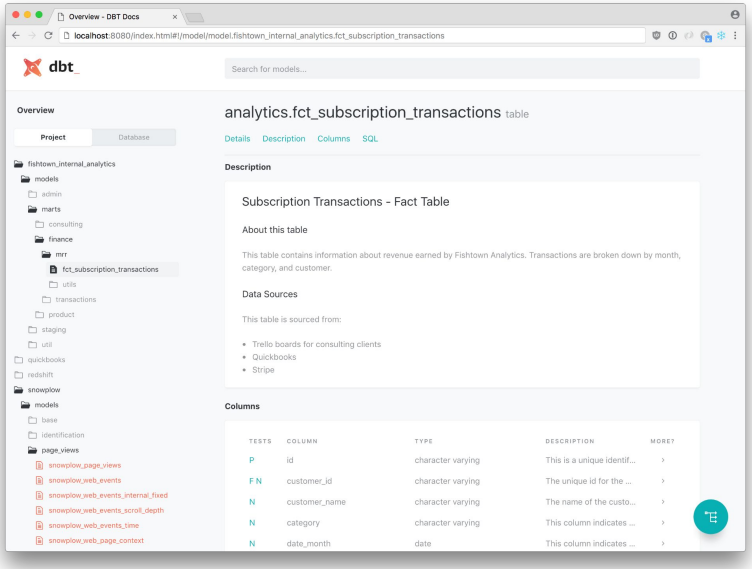
- DBT génère une documentation interactive avec la commande :

  dbt docs generate

- Lancer un serveur local pour visualiser la documentation :

  dbt docs serve

- Permet de naviguer et comprendre les dépendances entre modèles DBT.



4. Bonnes pratiques et déploiement

Tests, documentation et modularité dans DBT

Documentation et génération des rapports

Gestion et partage des documentations

- Stocker la documentation générée sur un serveur interne ou dans un outil de Data Catalog.
- Exporter la documentation statique en HTML.
- Intégration possible avec des outils comme DataHub ou Metabase.

4. Bonnes pratiques et déploiement

Utilisation des macros et modularité dans DBT

Jinja et DBT – Fonctionnalités avancées

- DBT utilise **Jinja** pour automatiser et paramétrer les modèles SQL.
- Exécution dynamique de requêtes avec des boucles et conditions.
- Exemple d’utilisation de Jinja :

```
SELECT *
FROM {{ ref('stg_orders') }}
WHERE order_status = '{{ var("status_filter", "completed") }}'
```

4. Bonnes pratiques et déploiement

Utilisation des macros et modularité dans DBT

Création de macros personnalisées

- Macros = Fonctions réutilisables écrites en SQL + Jinja.
- Exemple de macro pour calculer le total des ventes :

```
{% macro total_sales() %}
SELECT SUM(total_amount) AS total_sales FROM {{ ref('stg_orders') }}
{% endmacro %}
```

- Utilisation dans un modèle :

```
SELECT {{ total_sales() }}
```

4. Bonnes pratiques et déploiement

Utilisation des macros et modularité dans DBT

Modularisation et optimisation SQL avec des macros

- Réduction de la redondance en factorisant du code SQL récurrent.
- Exemple de macro pour normaliser les dates :

```
{% macro convert_date(column_name) %}
CAST({{ column_name }} AS DATE)
{% endmacro %}
```

- Utilisation dans un modèle :

```
SELECT {{ convert_date('created_at') }} FROM {{ ref('stg_orders') }}
```

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Optimisation des performances SQL dans DBT

Optimisation des requêtes pour les Data Warehouses

- Exploiter les fonctionnalités spécifiques des entrepôts de données :
  - **BigQuery** : Utiliser partition_by et cluster_by.
  - **Snowflake** : Optimiser l'utilisation du cache et du clustering automatique.
  - **Redshift** : Privilégier les **sort keys** et **dist keys**.
- Limiter le volume de données traité avec **SELECT spécifique** plutôt que SELECT *.
- Optimiser les jointures en réduisant le nombre de colonnes chargées.

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Optimisation des performances SQL dans DBT

Bonnes pratiques pour écrire des requêtes SQL performantes

- Privilégier les **CTE (WITH statements)** pour améliorer la lisibilité et éviter les sous-requêtes complexes.
- Utiliser des **filtres précoces** pour restreindre le nombre de lignes dès le départ.
- Indexer et partitionner les tables pour accélérer les requêtes analytiques.

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Optimisation des performances SQL dans DBT

Indexation et partitionnement des tables

- Partitionner les tables sur des colonnes temporelles (partition_by) pour limiter le scan des données :

```
{{ config(
    materialized='table',
    partition_by={'field': 'order_date', 'data_type': 'date'}
) }}
```

- Clustering (cluster_by) pour regrouper les données fréquemment filtrées :

```
{{ config(
    materialized='table',
    cluster_by=['customer_id']
) }}
```

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Optimisation des performances SQL dans DBT

Matérialisation incrémentale – Stratégies avancées

- **merge** : Mise à jour automatique des lignes modifiées (Snowflake, BigQuery) :

```
{{ config(
    materialized='incremental',
    unique_key='order_id',
    incremental_strategy='merge'
) }}
```

- **delete+insert** : Suppression des anciennes données avant insertion (PostgreSQL, Redshift) :

```
{{ config(
    materialized='incremental',
    unique_key='customer_id',
    incremental_strategy='delete+insert'
) }}
```

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Surveillance et monitoring des modèles DBT

Suivi des performances DBT (dbt debug, dbt compile)

- Vérification de la configuration et des connexions :

  dbt debug

- Compilation des modèles pour identifier les problèmes avant exécution :

  dbt compile

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Surveillance et monitoring des modèles DBT

Intégration avec des outils de monitoring

- **DataDog** : Surveillance des performances SQL et des requêtes DBT.
- **Prometheus & Grafana** : Création de dashboards pour analyser les latences et erreurs d’exécution.
- **Dagster UI** : Visualisation des exécutions et suivi des performances des jobs DBT.

4. Bonnes pratiques et déploiement

Optimisation et bonnes pratiques en production

Surveillance et monitoring des modèles DBT

Gestion des erreurs et stratégies de rollback

- Surveiller et loguer les erreurs dans **logs/dbt.log**.
- Implémenter un rollback automatique en cas d'échec :

  dbt run --target prod || dbt run --target rollback

- Envoyer des alertes en cas d'échec via Slack ou email.

4. Bonnes pratiques et déploiement