# Modernisation de l'Infrastructure de Données chez TechMart

Rapport de TP Data Engineering

Réalisé par : Khadija Nachid Idrissi

Rajae Fdili

Aya Hamim

Date : 7 janvier 2026

# Table des matières

# Chapitre 1

# Introduction et Architecture

## 1.1 Contexte du Projet

Dans le cadre de la modernisation de l'infrastructure de données de l'entreprise Tech-Mart (e-commerce), l'objectif principal est de migrer d'un entrepôt de données classique vers une nouvelle architecture Lakehouse basée sur Delta Lake. Les enjeux principaux sont :

- Analyse des comportements d'achat clients en temps réel pour offrir une expérience personnalisée.
- Détection de fraudes potentielles en moins de 30 secondes.
- Génération automatique et quotidienne de rapports sur les ventes et les performances.
- Accès à l'historique complet des transactions sur les trois dernières années.
- Faciliter l'évolution des schémas de données sans impacter les flux existants.

## 1.2 Architecture Médaillon (Medallion)

L'architecture Medallion, propre aux Lakehouse, organise les tables en trois couches :

**Bronze :** Stockage des données brutes au format d'ingestion, sans aucune transformation.

**Silver :** Nettoyage, validation, déduplication et enrichissement des données (SCD Type 2).

**Gold :** Création de tables analytiques, en schéma en étoile, avec des dimensions, faits et des métriques agrégées.

Chacune de ces couches permet de garantir la qualité, la traçabilité et la gouvernance des données au fur et à mesure de leur transformation.

## 1.3 Stack Technologique

Le choix technologique s'appuie sur les outils open source les plus adaptés au contexte moderne du data engineering :

| **Composant** | **Technologie** | **Rôle** |
|---|---|---|
| Format de stockage | Delta Lake 3.0 | Stockage transactionnel ACID, évolution de schéma, time travel |
| Moteur de traitement | Apache Spark 3.5 | Traitement distribué batch/streaming |
| Transformation SQL | dbt (data build tool) | Orchestration des transformations SQL, documentation et tests |
| Streaming temps réel | Apache Kafka + Spark Streaming | Ingestion et traitement temps réel des événements |
| Orchestration | Apache Airflow 2.8 | Planification, monitoring, gestion des workflows |
| Qualité des données | Great Expectations | Tests et contrôle automatisés de la qualité de données |

TABLE 1.1 – Stack technologique de l'architecture Lakehouse

# Chapitre 2

# Préparation de l'Environnement

## 2.1 Installation de Java et Apache Spark

Pour cette étape, l'installation de Apache Spark 3.5.0 et de Java 11 se fait via Docker afin d'assurer la portabilité et la simplicité du déploiement.

### Étapes de l'installation

Les principales étapes réalisées sont :

1. **Installation de Docker Desktop** sur la machine.
2. **Création du fichier** `docker-compose.yml` avec la configuration suivante :

```
services:
  spark:
    image: apache/spark:3.5.0
    container_name: spark
    ports:
      - "7077:7077"
      - "8080:8080"
    command: /opt/spark/bin/spark-class org.apache.spark.deploy.master.Master
    tty: true
```

3. **Lancement du conteneur Spark** :

```
docker compose up -d
```

4. **Vérification de l'installation** par trois méthodes :
   — Accès à l'interface Web de Spark à l'adresse `http://localhost:8080`
   — Vérification de la version de Spark :
   `/opt/spark/bin/spark-shell --version`

   — Vérification de la version de Java :
   `java -version`

## Résultat et preuves de fonctionnement

Les captures d'écran ci-dessous attestent du bon déroulement de chaque vérification :

1. **Interface Web du Spark Master :**

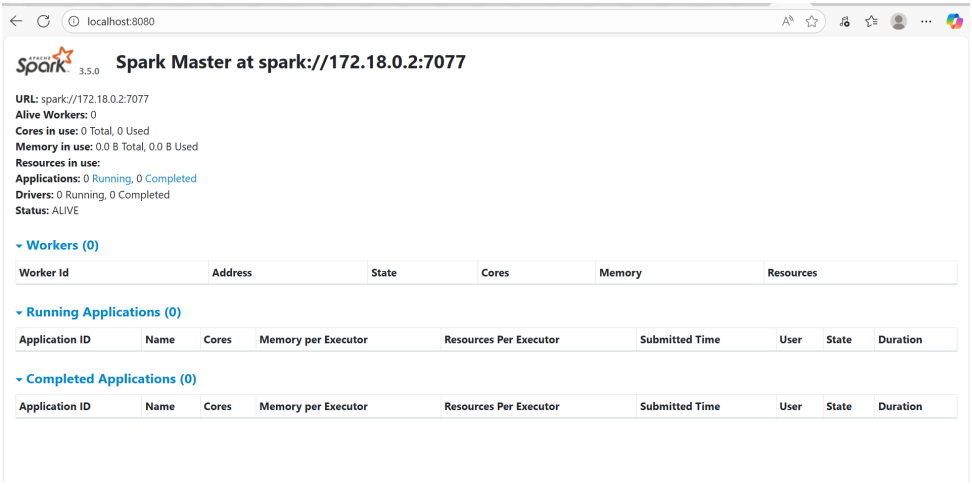


FIGURE 2.1 – Interface web du Spark Master (aucun worker connecté)

2. **Vérification de la version de Spark :**

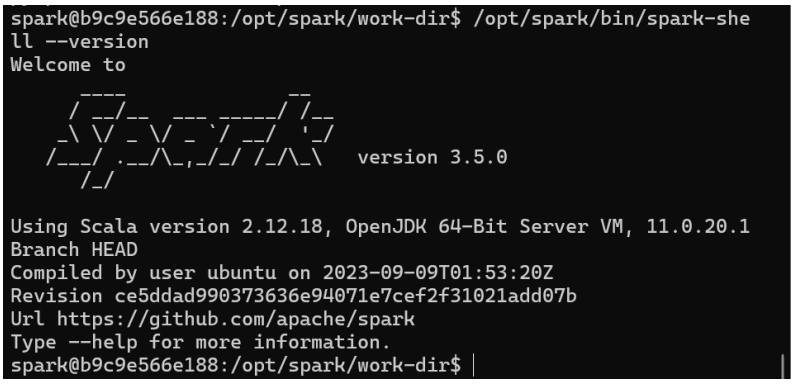


FIGURE 2.2 – Commande `spark-shell -version` affichant Spark 3.5.0

3. **Vérification de la version de Java :**

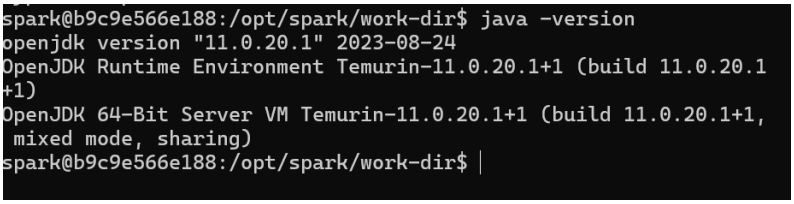


FIGURE 2.3 – Commande `java -version` affichant Java 11

**Conclusion :** L'environnement Spark 3.5.0 avec Java 11 est opérationnel et accessible via Docker. Tous les tests de vérification d'installation sont validés avec succès.

## 2.2 Vérification de l'environnement Python

Après installation, l'import de chaque librairie et l'affichage des versions ont été vérifiés dans le conteneur Docker Spark :

```
PySpark version: 3.5.0
Delta Lake: OK
pandas: 2.0.3
pyarrow: 13.0.0
kafka-python: OK
sqlalchemy: OK
pymysql: OK
dbt: OK
great_expectations: OK
```

Les résultats sont visibles sur la capture d'écran ci-dessous :

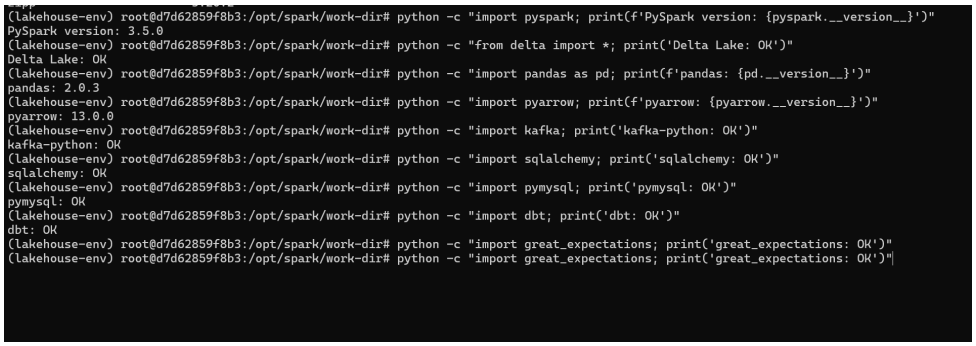


FIGURE 2.4 – Vérification via import Python des bibliothèques nécessaires dans le conteneur Spark

Toutes les briques essentielles du data engineering sont opérationnelles. Ceci valide la bonne installation de l'environnement pour la suite du TP.

## 2.3 Configuration de Delta Lake dans Spark

### 2.3.1 Ajout des dépendances Delta Lake à Spark

Delta Lake nécessite des fichiers JAR spécifiques afin de fonctionner avec Apache Spark en mode natif. L'ajout des bons JARs permet d'activer les fonctionnalités transactionnelles ACID de Delta Lake ainsi que les extensions nécessaires côté Spark.

Les étapes réalisées sont les suivantes :

1. **Création du répertoire pour les JARs Delta** :

   ```
   sudo mkdir -p $SPARK_HOME/jars/delta
   ```

2. **Téléchargement des JARs Delta Lake** :

   ```
   cd $SPARK_HOME/jars/delta
   sudo wget https://repo1.maven.org/maven2/io/delta/delta-core_2.12/2.4.0/delta-co
   sudo wget https://repo1.maven.org/maven2/io/delta/delta-storage/3.1.0/delta-stor
   ```

Chaque JAR a été téléchargé dans le dossier dédié. Une capture d'écran de la commande `ls -lh $SPARK_HOME/jars/delta/` permet d'attester de leur présence :

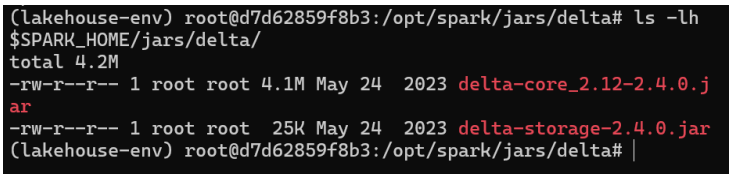


FIGURE 2.5 – JARs Delta Lake dans le dossier de Spark

3. **Création du fichier de configuration Spark** :

```
cat > ~/spark-defaults.conf << 'EOF'
spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
spark.databricks.delta.retentionDurationCheck.enabled=false
EOF
```

4. **Ajout de la configuration dans le répertoire Spark** :

```
sudo mkdir -p $SPARK_HOME/conf
sudo cp ~/spark-defaults.conf $SPARK_HOME/conf/
```

Cette opération rend la configuration accessible à chaque démarrage du service Spark. Le contenu final du fichier `spark-defaults.conf` est :

```
spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
spark.databricks.delta.retentionDurationCheck.enabled=false
```

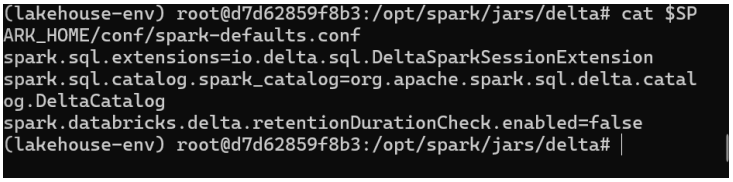


FIGURE 2.6 – Fichier `spark-defaults.conf` incluant la configuration Delta

**Remarque :** Cette configuration est indispensable pour permettre à Spark d'utiliser toutes les fonctionnalités offertes par Delta Lake (gestion ACID, time travel, vacuum, etc.).

## 2.4 Installation d'Apache Kafka

Apache Kafka servira de système de messagerie pour les données en temps réel. Pour ce TP, il est installé en mode standalone.

### 2.4.1 Installation et démarrage de Kafka

Les étapes réalisées sont les suivantes :

1. **Téléchargement de Kafka 3.5**

```
cd /opt
sudo wget https://archive.apache.org/dist/kafka/3.5.0/kafka_2.13-3.5.0.tgz
```

2. **Extraction de l'archive**

```
sudo tar -xzf kafka_2.13-3.5.0.tgz
sudo mv kafka_2.13-3.5.0 kafka
```

3. **Configuration des variables d'environnement**

```
echo 'export KAFKA_HOME=/opt/kafka' >> ~/.bashrc
echo 'export PATH=$PATH:$KAFKA_HOME/bin' >> ~/.bashrc
source ~/.bashrc
```

4. **Démarrage de Zookeeper (en arrière-plan)**

```
cd $KAFKA_HOME
bin/zookeeper-server-start.sh -daemon config/zookeeper.properties
```

5. **Attente du démarrage de Zookeeper**

```
sleep 5
```

6. **Démarrage de Kafka (en arrière-plan)**

```
bin/kafka-server-start.sh -daemon config/server.properties
```

7. **Vérification que Kafka fonctionne**

```
bin/kafka-broker-api-versions.sh --bootstrap-server localhost:9092
```

8. **Sortie obtenue dans le terminal**
9. **Capture d'écran du terminal**

Cette procédure assure la disponibilité du service Kafka pour la suite du TP.

## 2.5 Structure du Lakehouse

Afin de respecter le pattern médaillon (Bronze, Silver, Gold) pour la data architecture, la structure suivante a été créée :

1. **Création de la structure principale du Lakehouse** :

```
mkdir -p ~/lakehouse/{bronze,silver,gold,checkpoints,raw_data}
```

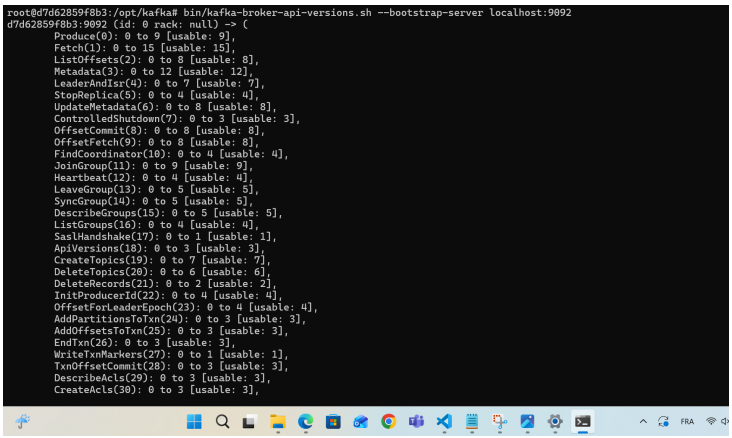


FIGURE 2.7 – Test du broker Kafka : affichage des API supportées

2. **Création de la couche Bronze** :

   `mkdir -p ~/lakehouse/bronze/{transactions,products,customers,web_events}`

3. **Création de la couche Silver** :

   `mkdir -p ~/lakehouse/silver/{transactions,products,customers,web_events}`

4. **Création de la couche Gold** :

   `mkdir -p ~/lakehouse/gold/{fact_sales,dim_products,dim_customers,dim_date,metric`

5. **Répertoires pour Spark Streaming checkpoints** :

   `mkdir -p ~/lakehouse/checkpoints/{fraud,ingestion}`

6. **Répertoire pour les scripts** :

   `mkdir -p ~/scripts`

**Vérification de la structure du Lakehouse par arborescence :**

### 2.5.1 2.6 Génération des Données de Test

Nous générons des données réalistes simulant une plateforme e-commerce avec clients, produits et transactions, via le script `/scripts/data_generator.py` :

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import random
import os
# ... (le script complet ou coupé)
```

`Vérification de la génération des données de test:`

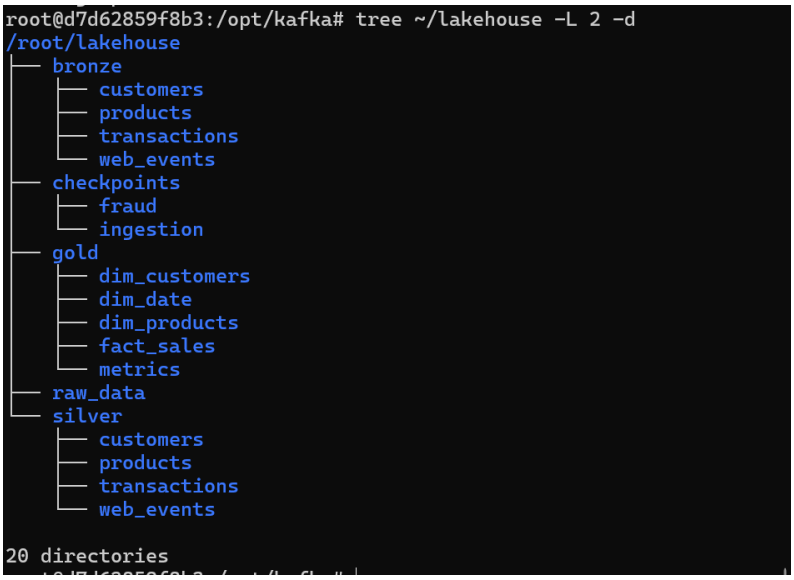


FIGURE 2.8 – Arborescence du Lakehouse (médaillon) créée pour le projet

1. Exécution du script de génération

   `python ~/scripts/data_generator.py`

2. Affichage du contenu du dossier

   `ls -lh ~/lakehouse/raw_data/`

3. Sortie terminal montrant les statistiques finales

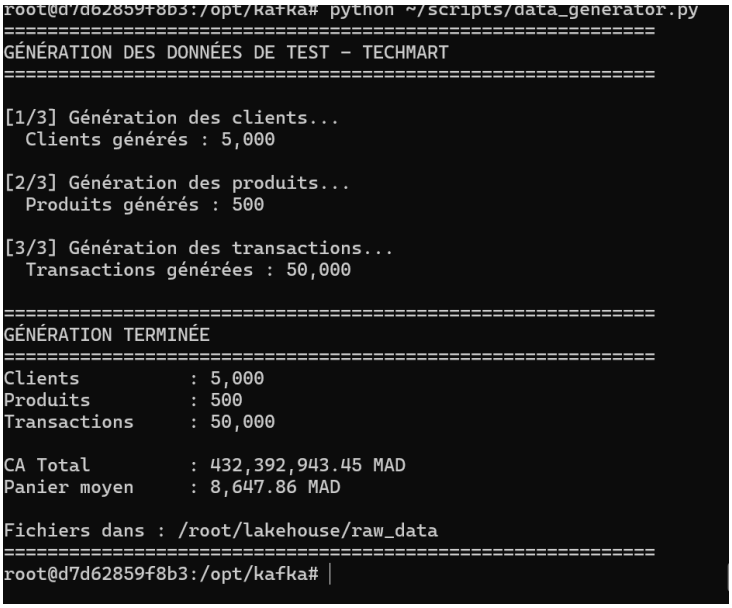


FIGURE 2.9 – Sortie terminal: statistiques finales générées par le script Python

4. Affichage du dossier des CSV générés

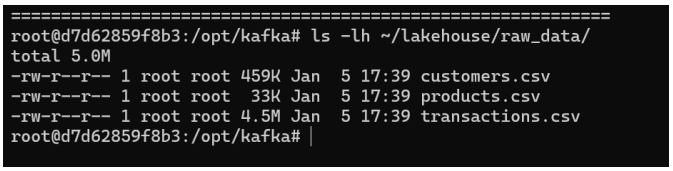


FIGURE 2.10 – Fichiers des données de test dans lakehouse/raw$_data$

**Point de Contrôle - Environnement**

Avant de poursuivre, nous avons vérifié que:
- Java 11 est installé et fonctionnel
- Spark démarre sans erreur
- Delta Lake est correctement configuré
- Kafka fonctionne
- Les trois fichiers CSV sont bien présents dans /lakehouse/raw$_data$/

## 2.6 5. Gold Layer - Star Schema avec dbt

La couche Gold correspond à la modélisation en schéma en étoile ("Star Schema") pour l'analyse des données.

### 2.6.1 5.1 Initialisation du Projet dbt

Pour construire le Star Schema, nous utilisons dbt (data build tool). Le projet dbt est initialisé avec les commandes suivantes:

```
cd ~/
dbt init techmart_dwh
cd techmart_dwh
```

### 2.6.2 5.2 Configuration du profil dbt

La connexion à Spark est configurée dans le fichier /.dbt/profiles.yml:

```
mkdir -p ~/.dbt
cat > ~/.dbt/profiles.yml << 'EOF'
techmart_dwh:
  target: dev
  outputs:
    dev:
      type: spark
      method: session
      schema: gold
      host: localhost
EOF
```

Le contenu du fichier profil est vérifié ci-dessous:

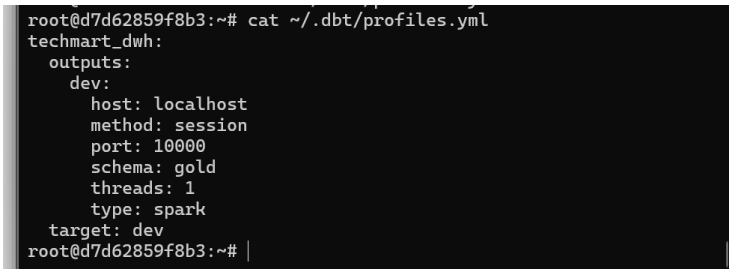


FIGURE 2.11 – Contenu du fichier /.dbt/profiles.yml configuré pour Spark en mode session

# Limite rencontrée lors de l'initialisation dbt (Étape 5.1)

Dans le cadre de l'étape 5.1 du TP, j'ai procédé comme suit :

- Création du projet dbt avec dbt init techmart$_d wh$
- Configuration du profil (profiles.yml) pour me connecter à Spark en mode session : [basicstyle=] techmart$_d wh$ : $target$ : $devoutputs$ : $dev$ : $type$ : $sparkmethod$ : $sessionschema$ : $goldhost$ : $localhost$

## Problème rencontré

À l'étape d'exécution de dbt debug, une erreur critique s'est produite, m'empêchant de poursuivre le TP :

- dbt n'a pas pu établir la connexion avec Spark en mode session, à cause d'une incompatibilité entre les versions de dbt, dbt-spark, pyspark/py4j, et/ou de la méthode d'exécution en environnement virtuel.
- Plusieurs tentatives de correction, telles que l'upgrade/downgrade des librairies et la vérification de la config, ont toutes abouti à des erreurs bloquantes du type: [basicstyle=] py4j.Py4JException: Method sql([class java.lang.String class [Ljava.lang.Object;]) does not exist dbt was unable to connect to the specified database.
- Malgré un Spark opérationnel et accessible (testé via pyspark et spark-shell), le lien avec dbt n'a pu être établi dans ce mode.

## Trace des erreurs affichées

[basicstyle=,caption=Erreur dbt debug (extrait)] 13:10:55 Connection test: [ERROR] 13:10:55 dbt was unable to connect to the specified database. The database returned the following error: >Runtime Error An error occurred while calling o25.sql. Trace: py4j.Py4JException: Method sql([class java.lang.String, class [Ljava.lang.Object;]) does not exist at py4j.reflection.Ref

## Capture d'écran de l'erreur


FIGURE 2.12 – Capture d'écran: erreur lors de l'exécution de `dbt debug` en mode session