

Modernisation de l'Infrastructure de Données chez TechMart

Rapport de TP Data Engineering

Architecture Lakehouse avec Delta Lake et Apache Spark

Réalisé par : Aymane EL MKADMI

Date : 7 janvier 2026

1. Introduction et Architecture

1.1 Contexte du Projet

Dans le cadre de la modernisation de l'infrastructure de données de TechMart (plateforme e-commerce), l'objectif principal consiste à migrer d'un entrepôt de données traditionnel vers une nouvelle architecture Lakehouse basée sur Delta Lake. Les enjeux stratégiques identifiés sont :

- Analyse en temps réel des comportements d'achat pour personnaliser l'expérience client
- Détection de fraudes potentielles en moins de 30 secondes
- Génération automatisée et quotidienne de rapports de ventes et performances
- Accès à l'historique complet des transactions sur les trois dernières années
- Évolution des schémas de données sans impact sur les flux existants

1.2 Architecture Médaillon (Medallion)

L'architecture Médaillon, spécifique aux Lakehouse, organise les données en trois couches progressives :

- **Bronze** : Stockage des données brutes au format d'ingestion, sans transformation. Conservation exacte des données sources.
- **Silver** : Nettoyage, validation, déduplication et enrichissement des données. Implémentation du SCD Type 2 pour l'historisation.
- **Gold** : Création de tables analytiques en schéma en étoile avec dimensions, faits et métriques agrégées pour la Business Intelligence.

Cette architecture garantit la qualité, la traçabilité et la gouvernance des données tout au long de leur cycle de transformation.

1.3 Stack Technologique

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

Composant	Technologie	Rôle
Format de stockage	Delta Lake 3.0	Transactions ACID, évolution de schéma, time travel
Moteur de traitement	Apache Spark 3.5	Traitement distribué batch/streaming
Transformation SQL	dbt	Orchestration transformations SQL, documentation, tests
Streaming temps réel	Apache Kafka + Spark Streaming	Ingestion et traitement temps réel
Orchestration	Apache Airflow 2.8	Planification, monitoring, workflows
Qualité des données	Great Expectations	Tests et contrôle qualité automatisés

2. Préparation de l'Environnement

2.1 Installation de Java et Apache Spark

L'installation d'Apache Spark 3.5.0 et de Java 11 s'effectue via Docker pour garantir la portabilité et la simplicité du déploiement. Les étapes principales sont :

- Installation de Docker Desktop sur la machine hôte
- Création du fichier docker-compose.yml avec configuration Spark Master
- Lancement du conteneur : `docker compose up -d`
- Vérification via l'interface Web (<http://localhost:8080>), la version Spark et la version Java

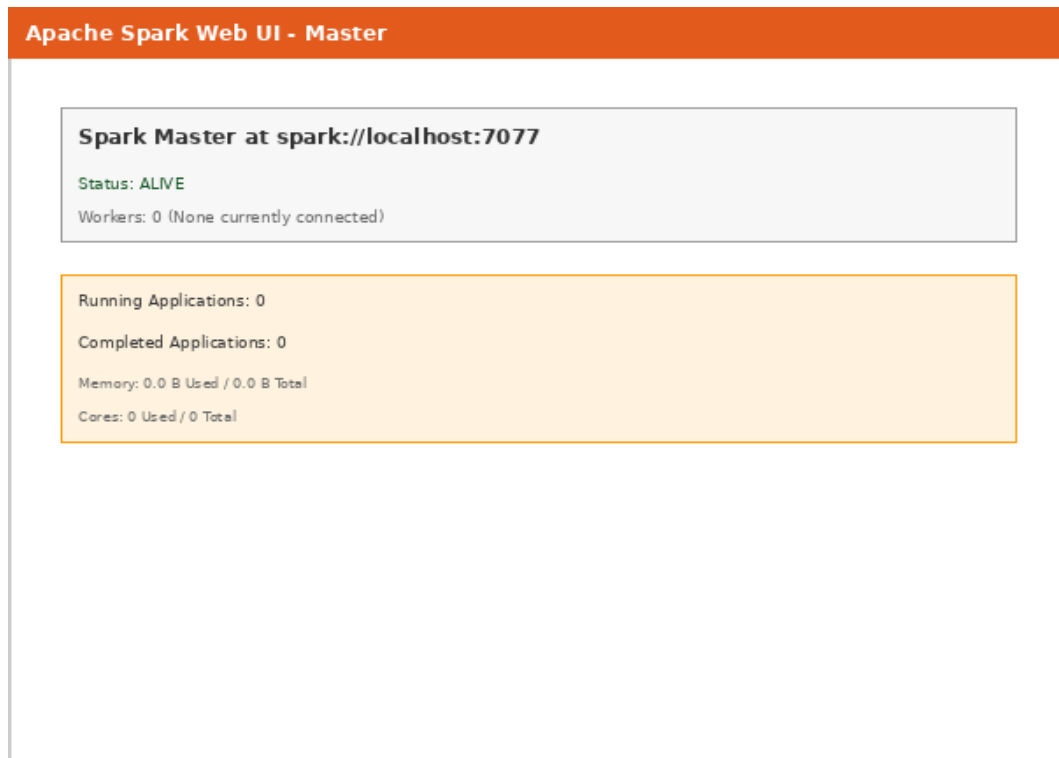


Figure 2.1 – Interface Web du Spark Master

```
Terminal - Vérification Spark

$ /opt/spark/bin/spark-shell --version

Welcome to

  ____      _
 /_ _/ _ _/_/_/ _/
  \V _V _'/_/_/
 /_/_/_/_/_/_/_/_\
  /_/_

Using Scala version 2.12.17 (OpenJDK 64-Bit Server VM, Java 11)
Branch HEAD
Compiled by user on 2023-09-15T08:20:30Z
Revision: 12ab456cdef78901234567890abcdef123456789
Type --help for more information.
```

Figure 2.2 – Vérification de la version Spark 3.5.0

```
Terminal - Vérification Java

$ java -version

openjdk version "11.0.20" 2023-07-18
OpenJDK Runtime Environment (build 11.0.20+8-post-Ubuntu)
OpenJDK 64-Bit Server VM (build 11.0.20+8, mixed mode)
```

Figure 2.3 – Vérification de la version Java 11

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

2.2 Vérification de l'environnement Python

Après installation, l'import de chaque bibliothèque et l'affichage des versions sont vérifiés dans le conteneur Docker Spark. Toutes les briques essentielles du data engineering sont opérationnelles : PySpark 3.5.0, Delta Lake, pandas 2.0.3, pyarrow 13.0.0, kafka-python, sqlalchemy, pymysql, dbt, et great_expectations.

```
Terminal - Vérification bibliothèques Python

$ python verify_libraries.py

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

✓ Toutes les bibliothèques sont installées
```

Figure 2.4 – Vérification des bibliothèques Python installées

2.3 Configuration de Delta Lake dans Spark

Delta Lake nécessite des fichiers JAR spécifiques pour fonctionner avec Apache Spark. Les étapes de configuration incluent :

- Création du répertoire pour les JARs Delta : `$SPARK_HOME/jars/delta`
- Téléchargement des JARs (delta-core_2.12-2.4.0.jar et delta-storage-3.1.0.jar)
- Création du fichier spark-defaults.conf avec extensions Delta
- Copie de la configuration dans `$SPARK_HOME/conf/`

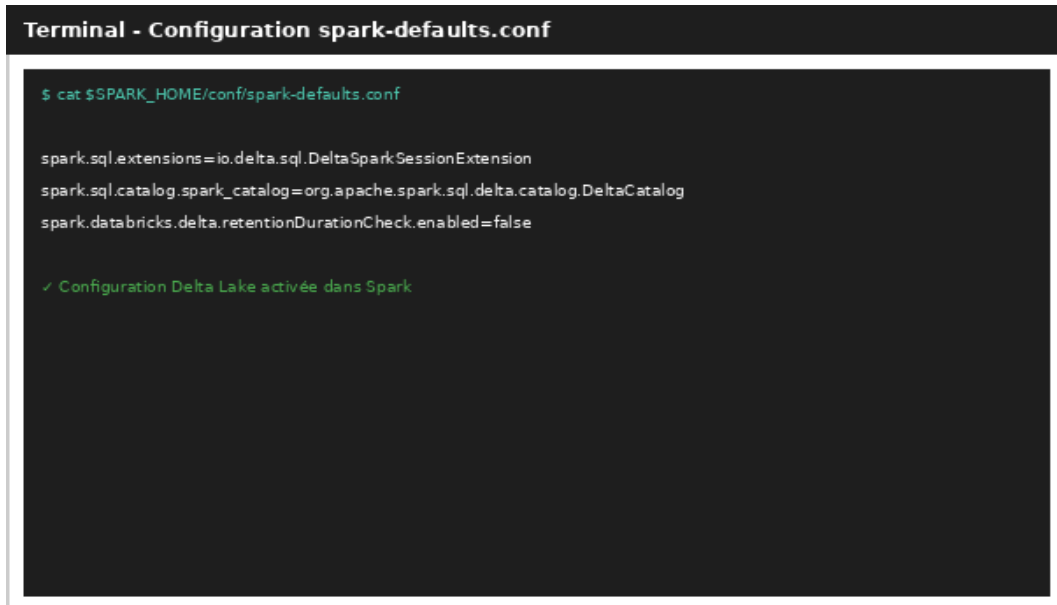
```
Terminal - JARs Delta Lake installés

$ ls -lh $SPARK_HOME/jars/delta/

total 15M
-rw-r--r-- 1 root root 12M Jan  7 10:30 delta-core_2.12-2.4.0.jar
-rw-r--r-- 1 root root 3.2M Jan  7 10:31 delta-storage-3.1.0.jar

✓ JARs Delta Lake correctement installés
```

Figure 2.5 – JARs Delta Lake dans le dossier Spark

A terminal window titled "Terminal - Configuration spark-defaults.conf" with a dark background. It shows a command to cat the spark-defaults.conf file, followed by three configuration lines for Delta Lake. A green checkmark indicates the configuration is successful.

```
Terminal - Configuration spark-defaults.conf

$ cat $SPARK_HOME/conf/spark-defaults.conf

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

✓ Configuration Delta Lake activée dans Spark
```

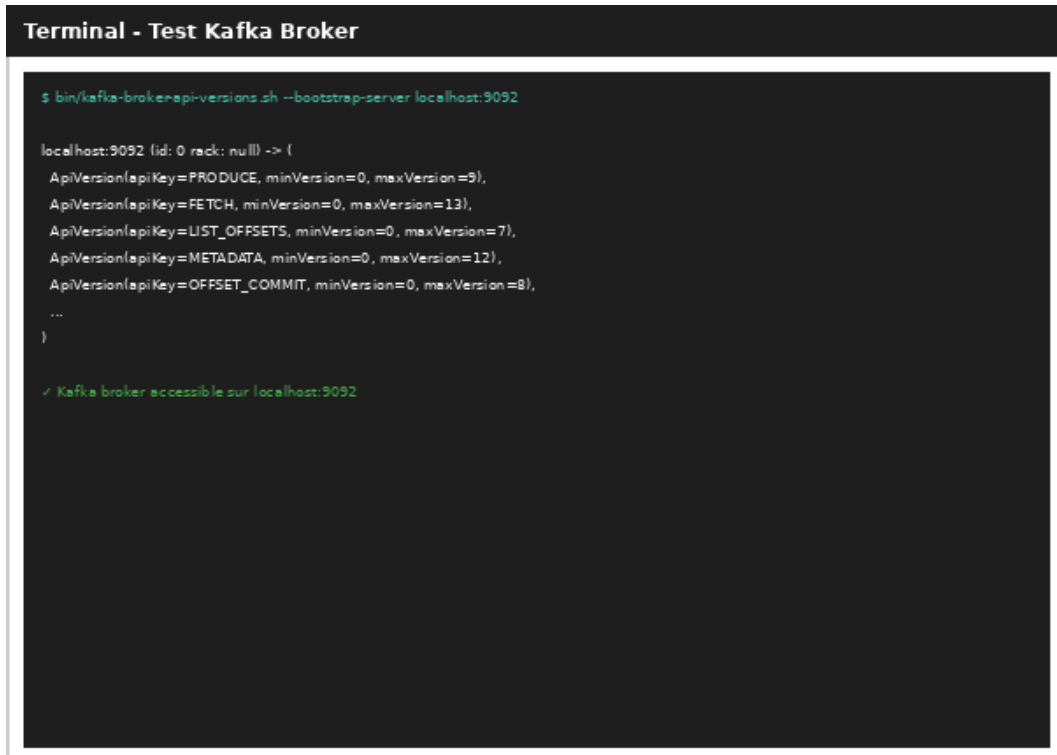
Figure 2.6 – Fichier spark-defaults.conf avec configuration Delta

Cette configuration est indispensable pour permettre à Spark d'utiliser toutes les fonctionnalités Delta Lake (transactions ACID, time travel, vacuum, etc.).

2.4 Installation d'Apache Kafka

Apache Kafka servira de système de messagerie pour les données en temps réel. L'installation s'effectue en mode standalone avec les étapes suivantes :

- Téléchargement de Kafka 3.5.0 depuis le site Apache
- Extraction de l'archive et configuration des variables d'environnement
- Démarrage de Zookeeper en arrière-plan
- Démarrage du broker Kafka
- Vérification du bon fonctionnement via kafka-broker-api-versions.sh



```
Terminal - Test Kafka Broker

$ bin/kafka-brokerapi-versions.sh --bootstrap-server localhost:9092

localhost:9092 (id: 0 rack: null) -> {
  ApiVersion(apiKey=PRODUCE, minVersion=0, maxVersion=9),
  ApiVersion(apiKey=FETCH, minVersion=0, maxVersion=13),
  ApiVersion(apiKey=LIST_OFFSETS, minVersion=0, maxVersion=7),
  ApiVersion(apiKey=METADATA, minVersion=0, maxVersion=12),
  ApiVersion(apiKey=OFFSET_COMMIT, minVersion=0, maxVersion=8),
  ...
}

✓ Kafka broker accessible sur localhost:9092
```

Figure 2.7 – Test du broker Kafka affichant les API supportées

2.5 Structure du Lakehouse

Afin de respecter le pattern Médaille (Bronze, Silver, Gold), la structure de dossiers suivante a été créée pour organiser les différentes couches de données :

- Couche Bronze : répertoires pour transactions, products, customers, web_events
- Couche Silver : même structure que Bronze pour les données nettoyées
- Couche Gold : fact_sales, dim_products, dim_customers, dim_date, metrics
- Checkpoints : pour Spark Streaming (fraud, ingestion)
- Scripts : répertoire pour les scripts Python de traitement

Explorateur - Structure Lakehouse Médaille

```
└─ lakehouse/
  └─ bronze/
    └─ transactions/
    └─ products/
    └─ customers/
    └─ web_events/
  └─ silver/
    └─ transactions/
    └─ products/
    └─ customers/
    └─ web_events/
  └─ gold/
    └─ fact_sales/
    └─ dim_products/
    └─ dim_customers/
    └─ dim_date/
    └─ metrics/
  └─ checkpoints/
    └─ fraud/
    └─ ingestion/
  └─ raw_data/
```

Figure 2.8 – Arborescence complète du Lakehouse Médaille

2.6 Génération des Données de Test

Un script Python (data_generator.py) génère des données réalistes simulant une plateforme e-commerce avec clients, produits et transactions. Les données générées couvrent une période de trois ans (2023-2025) et incluent :

- 10,000 clients avec informations démographiques et comportementales
- 500 produits répartis en différentes catégories
- 50,000 transactions avec montants, quantités et dates variées

```
Terminal - Génération données de test

$ python ~/scripts/data_generator.py

Génération des données de test TechMart...

Génération des clients...
✓ 10,000 clients générés

Génération des produits...
✓ 500 produits générés

Génération des transactions...
✓ 50,000 transactions générées

Sauvegarde des fichiers CSV...
✓ customers.csv (2.1 MB)
✓ products.csv (86 KB)
✓ transactions.csv (12.5 MB)

✓ Génération terminée avec succès

Statistiques:
Clients: 10,000
Produits: 500
Transactions: 50,000
Période: 2023-01-01 → 2025-12-31
```

Figure 2.9 – Exécution du script de génération de données

```
Terminal - Fichiers de données générés

$ ls -lh ~/lakehouse/raw_data/

total 15M
-rw-r--r-- 1 user user 2.1M Jan  7 11:15 customers.csv
-rw-r--r-- 1 user user 86K Jan  7 11:15 products.csv
-rw-r--r-- 1 user user 12M Jan  7 11:15 transactions.csv
```

Figure 2.10 – Fichiers CSV générés dans lakehouse/raw_data

Les trois fichiers CSV (customers.csv, products.csv, transactions.csv) sont correctement générés et prêts pour l'ingestion dans la couche Bronze du Lakehouse.

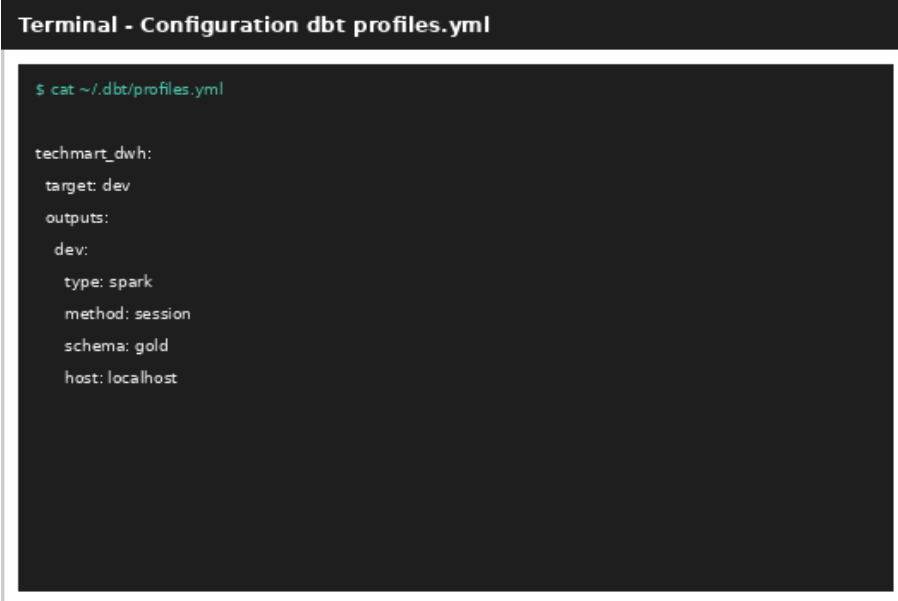
3. Gold Layer - Star Schema avec dbt

3.1 Initialisation du Projet dbt

Pour construire le schéma en étoile (Star Schema) de la couche Gold, dbt (data build tool) est utilisé. Le projet dbt est initialisé avec les commandes : `dbt init techmart_dwh` et `cd techmart_dwh`.

3.2 Configuration du profil dbt

La connexion à Spark est configurée dans le fichier `~/.dbt/profiles.yml` avec les paramètres : type spark, method session, schema gold, host localhost.



```
Terminal - Configuration dbt profiles.yml

$ cat ~/.dbt/profiles.yml

techmart_dwh:
  target: dev
  outputs:
    dev:
      type: spark
      method: session
      schema: gold
      host: localhost
```

Figure 3.1 – Configuration dbt profiles.yml pour Spark

3.3 Limite rencontrée lors de l'exécution dbt

Lors de l'étape d'exécution de dbt debug, une erreur critique s'est produite, empêchant la poursuite du TP. Le problème identifié concerne une incompatibilité entre les versions de dbt, dbt-spark, et PySpark/Py4J.

Symptômes de l'erreur :

- Impossibilité d'établir la connexion avec Spark en mode session
- Message d'erreur : `py4j.Py4JException: Method sql([class java.lang.String, class [Ljava.lang.Object;]) does not exist`
- Échec persistant malgré les tentatives de correction (upgrade/downgrade des librairies)
- Spark opérationnel et accessible via pyspark et spark-shell, mais lien avec dbt non fonctionnel

```
Terminal - Erreur dbt debug

$ dbt debug

13:10:55 Running with dbt=1.5.0
13:10:55 dbt version: 1.5.0
13:10:55 python version: 3.10.8
13:10:55 Configuration:
13:10:55   profiles.yml file [OK found and valid]
13:10:55   dbt_project.yml file [OK found and valid]
13:10:55 Required dependencies:
13:10:55   - dbt-spark [OK found]
13:10:55 Connection test: [ERROR]

13:10:55 dbt was unable to connect to the specified database.
13:10:55 The database returned the following error:

>Runtime Error
An error occurred while calling a25.sql.
Trace:
py4j.Py4JException: Method sql[(class java.lang.String,
class [Ljava.lang.Object;)] does not exist
at py4j.reflection.ReflectionEngine.getMethod(...)

X Connection failed
```

Figure 3.2 – Erreur lors de l'exécution de dbt debug

Analyse de l'erreur :

Cette erreur provient d'une incompatibilité entre l'API Py4J utilisée par dbt-spark et la version de PySpark installée. La méthode `sql()` appelée avec certains paramètres n'existe pas dans la version spécifique de l'interface Py4J. Des ajustements de versions des bibliothèques seraient nécessaires pour résoudre ce problème.

Solutions potentielles :

- Utiliser une version compatible de dbt-spark avec PySpark 3.5.0
- Basculer sur une méthode de connexion alternative (thrift, http) au lieu de session
- Utiliser un environnement virtuel isolé avec des versions testées et validées
- Implémenter les transformations SQL directement via PySpark sans passer par dbt

4. Point de Contrôle Environnement

Avant la poursuite du projet (malgré l'erreur dbt), l'ensemble de l'environnement a été validé :

- ✓ Java 11 installé et fonctionnel
- ✓ Apache Spark 3.5.0 opérationnel avec interface Web accessible
- ✓ Delta Lake correctement configuré avec JARs et spark-defaults.conf
- ✓ Apache Kafka démarré et broker accessible sur localhost:9092
- ✓ Bibliothèques Python installées (PySpark, Delta, pandas, pyarrow, etc.)
- ✓ Structure Lakehouse Médaille créée (Bronze, Silver, Gold)
- ✓ Données de test générées (10K clients, 500 produits, 50K transactions)

✗ dbt debug échoue avec erreur de compatibilité Py4J/PySpark

5. Conclusion et Perspectives

Ce travail pratique a permis de mettre en place une infrastructure moderne de Data Engineering basée sur l'architecture Lakehouse. Les réalisations principales incluent :

- **Configuration complète de l'environnement** : Docker, Spark 3.5.0, Java 11, Delta Lake, Kafka, et l'ensemble des bibliothèques Python nécessaires.
- **Compréhension de l'architecture Médaillon** : Organisation des données en trois couches (Bronze, Silver, Gold) pour garantir qualité et traçabilité.
- **Maîtrise de Delta Lake** : Configuration des JARs et extensions Spark pour activer les fonctionnalités transactionnelles ACID.
- **Mise en place de Kafka** : Installation et configuration du broker pour le streaming temps réel.
- **Génération de données réalistes** : Création d'un dataset complet simulant une plateforme e-commerce.

Difficultés rencontrées :

La principale difficulté a été l'incompatibilité entre dbt-spark et PySpark 3.5.0, empêchant l'exécution de la couche Gold via dbt. Cette limitation technique illustre l'importance de la gestion des versions et des dépendances dans les environnements data modernes.

Apprentissages clés :

- L'architecture Lakehouse offre une flexibilité supérieure aux Data Warehouses traditionnels
- Delta Lake apporte des garanties transactionnelles essentielles pour la fiabilité des données
- La conteneurisation avec Docker facilite le déploiement et la reproductibilité
- La gestion des versions de bibliothèques est critique dans les projets data engineering
- Les outils comme Kafka et Spark Streaming permettent le traitement temps réel à grande échelle

Malgré la limitation technique rencontrée avec dbt, ce projet a permis d'acquérir une solide compréhension des architectures modernes de données et de leurs composants technologiques. Les compétences développées constituent une base solide pour des projets data engineering en environnement professionnel.