Groupe:

- AMAR HASNAOUI, AMINE NAIT SIDHOUM, AYMEN MEZIANE

Documentation Technique : Projet BDF3

Présentation du projet

L’objectif de ce projet est de concevoir et mettre en place une architecture Big Data en médaillon (Bronze / Silver / Gold) afin de traiter et valoriser un jeu de données de transactions. Cette architecture permettra d’assurer un traitement structuré, fiable et évolutif des données, depuis l’ingestion brute jusqu’à leur exploitation analytique.

Architecture

1. Mise en place

- Nous avons installer la meme image que le professeur nous a fournie pour realiser le projet **BDF1** avec hadoop, spark et hive.
- une image mysql
- Dans le rendu bdf3 vous allez trouver des sous dossiers, **Traitement**, **ML**, **Source**, **lib**, **logs**.
- Traitement contient les trois fichiers : **feeder.py** | **processor.py** | **gold.py**
- ML contient le fichier : **detect_fraud.py**
- Source contient les fichier Source
- lib contient je jbdc mysql
- logs dossier pour stocker les logs
- Dans notre conteneur spark à l'aide de docker cp nous avons transferer c'est dossiers dans un dossier **projet_bdf3**

capture d'écran tree sur conteneur

Pipeline de Données vers Zone Bronze – Composant Feeder

1. Objectif du Script

Ce script Spark en Python a pour but de :

- **Extraire** les données brutes de différentes sources (MySQL, CSV, JSON).
- **Nettoyer, transformer** et **partitionner** ces données.
- **Stocker** les données formatées dans la **zone Bronze** du data lake, en format **Parquet partitionné** par date (`year/month/day`).

Cette étape est appelée **"Feeder"**, car elle alimente la zone Bronze à partir des sources brutes.

---

2. Architecture du Script (Structure orientée objet)

Le script est structuré en plusieurs **classes** qui séparent clairement les responsabilités :

Classe	Rôle
`SparkSessionManager`	Crée et configure la session Spark.
`LoggerManager`	Configure les logs pour le suivi des traitements.
`BaseProcessor`	Contient les méthodes et dates communes aux autres classes.
`MySQLProcessor`	Lit les données `transactions` depuis MySQL par batchs.
`CSVProcessor`	Transforme et charge les fichiers CSV (`cards`, `users`).
`JSONProcessor`	Transforme et charge les fichiers JSON (`mcc_codes`, `train_labels`).

---

3. Traitement par Source

a) Données SQL (transactions)

- **Source** : base MySQL `Source`, table `transactions`.
- **Traitement** :
  - Lecture **par lot de 4 millions** d'IDs pour éviter les surcharges mémoire.
  - Conversion de la colonne `situation_date` au format date.
  - Ajout de colonnes `year`, `month`, `day` pour la partition.
  - Sauvegarde en **Parquet partitionné** dans `/Bronze/transactions/`.
- **Format final** : Parquet, partitionné par `year`, `month`, `day`.

---

b) Données CSV (`cards`, `users`)

- **Source** : fichiers plats CSV.
- **Traitement** :
  - Lecture avec en-tête (`header=True`) et séparateur `,`.
  - Conversion directe au format Parquet.
  - Sauvegarde partitionnée dans `/Bronze/<table>/year=YYYY/month=MM/day=DD/`.
- **Format final** : Parquet partitionné.

---

c) Données JSON (`mcc_codes`, `train_fraud_labels`)

- `mcc_codes.json`, `train_fraud_labels.json` : fichier au format `clé:valeur`, transformé en deux colonnes.
- **Traitement** :
  - Chargement des objets JSON.
  - Sauvegarde partitionnée en Parquet comme les autres.

---

4. Bonnes pratiques de Data Management appliquées

Bonne pratique	Application dans le script
Partitionnement temporel	`year/month/day` utilisé dans tous les formats pour requêtes efficaces.

Format optimisé	Utilisation du **format Parquet** pour la compression + performance.
Modularité	Le script est orienté objet avec des classes indépendantes.
Logs et monitoring	Chaque étape est loggée avec niveau `INFO` ou `ERROR`.
Gestion mémoire	Utilisation de `.cache()`, `.unpersist()`, `gc.collect()`.
Lecture incrémentale SQL	Lecture par **batch** via des plages d'ID.
Séparation des responsabilités	Code découpé en classes spécialisées.

---

5. Organisation des Données – Zone Bronze

Les données sont organisées dans `/projet_bdf3/Bronze/<table>/year=YYYY/month=MM/day=DD/` :

Exemple : capture d'écran tree

Ce script exécute le traitement des données issues du niveau **Bronze** pour les transformer en **niveau Silver**. Il applique des étapes de **normalisation**, **nettoyage**, **typage**, et enregistre les données :

- dans **Hive** (via des tables partitionnées)
- et dans des fichiers **Parquet** sur le **HDFS local**

Pipeline de Données vers Zone Silver – Composant processor

Architecture du script

Classes

`PipelineProcessor`

Classe principale orchestrant les traitements Spark pour les différentes tables Bronze.

Méthodes internes

- `get_latest_local_path(base_path: str) -> str | None`
  → Retourne le chemin le plus récent (par date) dans la hiérarchie `year=/month=/day=`.
- `extract_date_parts_from_path(path: str) -> tuple[int, int, int]`
  → Extrait `year`, `month`, `day` à partir d'un chemin formaté.
- `init_spark()`
  → Initialise une `SparkSession` avec support Hive (via `hive.metastore.uris`).

Méthodes de traitement par table

`process_transactions()`

- **Source** : `/projet_bdf3/Bronze/transactions/`
- **Nettoyage** :
  - Nettoyage des colonnes `amount` → `amount_usd`, en enlevant `$`
  - Conversion des types : `id`, `zip`, `mcc`, etc.
- **Écriture** :
  - Hive : `silver.transactions`
  - Parquet : `/projet_bdf3/Silver/transactions/`

`process_cards()`

- **Source** : `/projet_bdf3/Bronze/cards/`
- **Nettoyage** :
  - Conversion de chaînes `"YES"` / `"NO"` en booléens (`has_chip`, `card_on_dark_web`)
  - `credit_limit` → `credit_limit_usd` en enlevant `$`
- **Écriture** :
  - Hive : `silver.cards`
  - Parquet : `/projet_bdf3/Silver/cards/`

`process_users()`

- **Source** : `/projet_bdf3/Bronze/users/`
- **Nettoyage** :
  - `gender` : `"Male"` / `"Female"` → `"M"` / `"F"`
  - `per_capita_income` → `per_capita_income_usd`
- **Écriture** :
  - Hive : `silver.users`
  - Parquet : `/projet_bdf3/Silver/users/`

`process_mcc_codes()`

- **Source** : `/projet_bdf3/Bronze/mcc_codes/`
- **Nettoyage** :
  - `mcc_code` converti en entier
- **Écriture** :
  - Hive : `silver.mcc`

- Parquet : `/projet_bdf3/Silver/mcc_codes/`

Logs

- Fichier log : `/projet_bdf3/logs/pipeline_processor.log`
- Les logs sont aussi affichés dans la console (via `StreamHandler`)
- Log des étapes :
  - Lecture
  - Normalisation
  - Écriture
  - Durée du traitement

Nettoyage mémoire

Chaque `DataFrame` est :

- `unpersist()`
- Supprimé via `del df`
- Collecté manuellement avec `gc.collect()`

capture d'écran tree silver
capture d'écran Hive
capture d'écran Hive
capture d'écran Hive
capture d'écran Hive

Pipeline de Données vers Zone Gold – Composant gold

Objectif

Le script Spark vise à transformer des données Silver (stockées dans Hive) en plusieurs **datamarts** métier dans la zone Gold (MySQL). Chaque datamart répond à une problématique analytique spécifique liée aux transactions bancaires, à la fraude, aux cartes et aux clients.

Architecture de la classe `Datamarts`

class Datamarts: Cette classe contient 5 méthodes qui encapsulent chacune un datamart spécifique. Chaque méthode prend un ou plusieurs DataFrames en entrée, effectue des transformations Spark (groupBy, joins, aggregations), et retourne un DataFrame prêt à être écrit dans MySQL.

Détail des Datamarts créés

1. dm_fraude_detection(df: DataFrame) But : Identifier le volume de fraudes par ville. Traitement :
   - Groupement par merchant_city et is_fraud.

- Agrégation : nombre total de transactions (count(*)).
- Résultat : capture d'écran Mysql

2. dm_clients(df_users: DataFrame, df_cards: DataFrame) But : Analyser le profil des clients avec leurs cartes bancaires. Traitement :
   - Jointure entre users et cards (via client_id).
   - Colonnes extraites : user_id, gender, credit_limit_usd, num_credit_cards.
   - Résultat : capture d'écran Mysql
3. dm_transactions_par_region(df: DataFrame) But : Observer les montants de transactions par État. Traitement :
   - Groupement par merchant_state.
   - Agrégation sur la somme des amount_usd.
   - Résultat : capture d'écran Mysql
4. dm_cartes(df: DataFrame) But : Analyser la répartition des cartes par type et marque. Traitement :
   - Groupement par card_brand et card_type.
   - Comptage du nombre de cartes.
   - Résultat : capture d'écran Mysql
5. dm_mcc_categories(df_trans: DataFrame, df_mcc: DataFrame) But :Comprendre les transactions par catégorie MCC (Merchant Category Code). Traitement :
   - Jointure avec la table mcc_codes sur le champ mcc.
   - Groupement par description.
   - Comptage du nombre de transactions.
   - Résultat : capture d'écran Mysql

Stockage final : MySQL (Zone Gold)

Toutes les sorties des datamarts sont écrites dans une base MySQL gold, en mode overwrite, via JDBC

```
df.write.mode("overwrite").jdbc(url=MYSQL_URL, table=table_name,
properties=MYSQL_PROPERTIES)
```

Points de clarté

Critère	Évaluation
Structure claire des datamarts	Classe dédiée `Datamarts`, méthodes bien nommées
Lisibilité des transformations Spark	Fonctions explicites avec `groupBy`, `agg`, `join`
Objectif métier identifiable	Chaque datamart répond à un besoin analytique concret
Séparation logique des étapes	Lecture, transformation, écriture séparées proprement
Commentaires / Logs	Logs présents

Execution

```
spark-submit \
  --jars lib/mysql-connector-j-8.3.0.jar \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 2 \
  --executor-memory 2G \
  --driver-memory 2G \
  --executor-cores 2 \
  Traitement/"traitement".py
```

Étant donné que ma machine dispose de 8 Go de RAM et 4 cœurs, j’ai ajusté les ressources comme suit pour assurer un traitement parallèle tout en évitant la saturation mémoire :

- num-executors 2 : J’ai choisi de lancer 2 exécuteurs pour permettre le parallélisme sans dépasser les ressources disponibles.
- executor-cores 2 : Chaque exécuteur utilise 2 cœurs, ce qui utilise l'ensemble des 4 cœurs disponibles (2 exécuteurs × 2 cœurs).
- executor-memory 2G : J’alloue 2 Go de RAM à chaque exécuteur. Avec 2 exécuteurs, cela fait 4 Go utilisés par les exécuteurs.
- driver-memory 2G : Le driver reçoit également 2 Go, ce qui fait un total de 6 Go alloués.
- Je garde une marge de 2 Go libre pour le système d’exploitation et les processus Spark internes, ce qui est important pour éviter les erreurs OutOfMemory.

capture d'écran yarn
capture d'écran spark ui

Machine Learning

Détection de Fraude avec Spark et GBTClassifier

Objectif

Ce module vise à entraîner un modèle de machine learning pour **détecter les fraudes bancaires** à partir des transactions présentes dans la zone Silver de notre Data Lake. L’approche est supervisée, avec un apprentissage à partir des données labellisées et un équilibrage des classes.

Technologies utilisées

- **Apache Spark** (PySpark) pour le traitement distribué
- **Hive** pour le stockage des tables Silver
- **MLlib** (Spark ML) pour l'entraînement du modèle
- **GBTClassifier** (Gradient Boosted Trees)

Pipeline de traitement

1. **Chargement des données depuis Hive** :
   - `silver.transactions`
   - `silver.train_fraud_labels`
   - `silver.cards`
   - `silver.mcc_codes`
2. **Préparation des données** :
   - Jointure des tables via les clés `id`, `card_id`, `mcc`
   - Conversion des colonnes nécessaires (`has_chip`, `fraud_label`) en types numériques
   - Filtrage pour ne conserver que les transactions ayant un label
3. **Échantillonnage équilibré** :
   - Toutes les fraudes (`label = Yes`)
   - Échantillon aléatoire de 10 % des non-fraudes (`label = No`)
   - Objectif : limiter le biais de classe majoritaire
4. **Traitement des variables catégorielles** :
   - Colonnes : `merchant_state`, `card_brand`, `card_type`, `has_chip`, `description`
   - Transformation avec `StringIndexer`
5. **Assemblage des features** :
   - Numériques : `amount_usd`, `credit_limit_usd`
   - Catégorielles indexées
   - Utilisation de `VectorAssembler`
6. **Entraînement du modèle** :
   - Modèle : `GBTClassifier` (30 itérations, 200 bins max)
   - Split 80/20 pour train/test
   - Vérification que l'ensemble d'entraînement n'est pas vide
7. **Évaluation** :
   - Métrique : AUC (Area Under ROC Curve)
   - Prédictions sur le jeu de test
8. **Sauvegarde du modèle** :
   - Répertoire : `ML/saved_model_balanced`
   - Format Spark ML

Résultat

- Le modèle a été entraîné avec succès sur un échantillon équilibré.
- Une AUC a été calculée = 0.98 pour évaluer la capacité du modèle à distinguer les fraudes.
- Le modèle a été sauvegardé pour réutilisation dans la phase de scoring ou d'inférence.


capture d'écran entrainement model

Test du model

Le model a été testé sur deux transactions via code python **test_model.py**. capture d'écran test model

---

Documentation API – Accès sécurisé aux données MySQL (couche Gold)

Contexte

Dans le cadre de notre architecture orientée données, cette API légère permet d'exécuter dynamiquement des requêtes SQL sur la base MySQL de la **zone gold**, tout en assurant un **accès sécurisé**, une **traçabilité des connexions**, et un **enregistrement des résultats**.

Objectifs

- Fournir une interface REST pour interroger la base de données MySQL.
- Authentifier les utilisateurs via un token unique.
- Consigner toutes les requêtes dans un fichier de logs.
- Permettre une exportation automatique des résultats au format CSV.

Technologies utilisées

Composant	Détail
Framework	Django
Base de données	MySQL (zone gold, conteneur Docker)
Authentification	Token UUID généré dynamiquement
Fichier de logs	`logs/access.log` (texte brut horodaté)
Export	Résultats enregistrés en CSV dans `query_results/`
Outils de test	Postman

Authentification

Route : `/login` (POST)

Permet d'obtenir un **token d'accès** unique pour l'utilisateur autorisé `aminamar`.

Requête

```
POST /login
Content-Type: application/json

{
```

```
  "username": "aminamar",
  "password": "admin123"
}
```

Réponse (succès)

```
{
  "message": "Connexion réussie",
  "token": "1d8a9e10-4d30-4f43-9b12-2b8f3f3d5478"
}
```

capture d'écran token

Réponses d’erreur possibles

- `403 FORBIDDEN` : Vous n'avez pas accès.
- `401 UNAUTHORIZED` : Mot de passe incorrect.

capture d'écran utilisateur incorect capture d'écran mot de passe incorect

---

Exécution de requêtes SQL

Route : `/query` (POST)

Permet de soumettre une requête SQL (type `SELECT`) à la base de données MySQL.

En-tête requis

```
Authorization: Bearer <your_token_here>
```

capture d'écran token

Requête

```
POST /query
Content-Type: application/json

{
  "query": "SELECT * FROM dm_cartes"
}
```

Réponse (succès)

```
{
  "message": "Query executed and saved to CSV successfully.",
  "results": [
    {
      "id": 1,
      "amount": 500,
      "date": "2023-01-01"
    }
  ],
```

```
  "file": "query_results/query_2025-05-30_17-43-21.csv"
}
```

capture d'écran reultat requete
capture d'écran reultat fichier
capture d'écran reultat requete
capture d'écran reultat fichier

Réponses d’erreur possibles

- `400 BAD REQUEST` : Requête SQL vide.
- `400 BAD REQUEST` : Erreur SQL levée (malformée ou non autorisée).
- `403 FORBIDDEN` : Token manquant ou invalide.

capture d'écran Requête SQL vide
capture d'écran Erreur SQL
capture d'écran Token manquant

Journalisation des accès

Chaque requête `/query` est consignée dans le fichier local suivant :

```
logs/access.log
```

Sécurité & validation

- *Vérification du token** via décorateur `@token_required`.
- Export automatique des résultats en CSV dans un dossier sécurisé.
- Traitement des erreurs avec des réponses structurées côté client.

Documentation – Visualisation et Recommandations (2/2)

Objectif

Analyser les transactions bancaires par zone géographique, genre, type de carte, ville marchande, et catégorie de dépense afin d’identifier les tendances, les comportements clients, et proposer des axes d’optimisation.

1. Visualisation 1 – Analyse géographique et par ville marchande Carte des États-Unis (par merchant_state)

   Représente la somme de total_usd par État. Une échelle de couleurs montre les États avec les volumes de transaction les plus élevés (plus foncé = plus élevé). Bar chart – Nombre de transactions par merchant_city La ville "ONLINE" est largement dominante avec près de 2M de transactions, bien devant Houston, Miami, Brooklyn, etc. Cela suggère une forte part des transactions.

Recommandations

Segmentation : créer une catégorie spécifique pour les transactions "ONLINE" afin d’en distinguer l’analyse. Focus marketing : investir dans les villes physiques à fort volume (Houston, Miami, etc.) pour stimuler encore plus l'activité. Analyse par État : les États les plus foncés (ex. : NY, CA, FL) méritent une attention particulière pour l’allocation de ressources.

2. Visualisation 2 – Analyse par genre, carte, type de dépenses, et localisation
   Crédit disponible par genre (credit_limit_usd)

   Répartition assez équilibrée entre les femmes (F) et les hommes (M). Légère avance du crédit total disponible pour les femmes. Nombre de transactions par description Catégories principales : épiceries, stations-service, restaurants, pharmacies, etc. Répartition logique avec les besoins de consommation courante. Mastercard domine, suivie de Visa. Débit est le type de carte le plus répandu, devant le crédit et le prépayé. total_usd par État marchand L'État Californie affiche un total très élevé. Suivi par New York, etc.

Recommandations

Produit bancaire ciblé : promouvoir davantage les cartes de crédit (moins utilisées) pour augmenter les marges bancaires. Analyse détaillée des dépenses : les épiceries et services alimentaires concentrent le plus de transactions → opportunité de partenariats. Étude de genre : comprendre pourquoi les femmes ont un crédit disponible légèrement plus élevé – potentiel d’approfondissement.
