Correction du TP4 : Configuration et Implémentation d'un Pipeline de Données avec Docker Compose et Outils Big Data

Introduction

Cette correction détaillée du TP4 présente la configuration et l'implémentation d'un pipeline de données complet utilisant Docker Compose et des outils Big Data (Hadoop, Spark, Flink, Kafka, Cassandra). Le pipeline est conçu pour gérer à la fois des traitements batch et temps réel, offrant ainsi une architecture Lambda complète.

La correction est structurée en sept parties principales, couvrant l'ensemble du cycle de vie du pipeline, de la mise en place de l'environnement de développement jusqu'au déploiement en production, en passant par l'implémentation des différents composants et les tests.

Chaque partie contient des explications détaillées, des exemples de code, des configurations et des bonnes pratiques pour vous guider dans la mise en œuvre de votre propre pipeline de données.

Table des matières

Commençons par la mise en place de l'environnement de développement.

Correction Partie 1 : Mise en place de l'environnement de développement avec Docker Compose

La mise en place d'un environnement de développement robuste, reproductible et proche de la production est une étape fondamentale pour tout projet d'ingénierie de données. Dans cette partie, nous allons utiliser Docker Compose pour créer un environnement complet intégrant tous les outils Big Data nécessaires à notre pipeline.

Avantages de Docker Compose pour un environnement Big Data

L'utilisation de Docker Compose pour un environnement Big Data présente de nombreux avantages :

1. **Isolation** : Chaque composant (Hadoop, Spark, Flink, Kafka, Cassandra) s'exécute dans son propre conteneur, évitant les conflits de dépendances.
2. **Reproductibilité** : L'environnement est défini comme code, garantissant que tous les développeurs travaillent dans des conditions identiques.
3. **Proximité avec la production** : L'environnement de développement peut être configuré pour refléter fidèlement l'environnement de production.
4. **Facilité de démarrage** : Un simple `docker-compose up` suffit pour démarrer l'ensemble de l'écosystème.
5. **Gestion des ressources** : Les ressources allouées à chaque service peuvent être contrôlées précisément.

Structure du projet

Commençons par définir une structure de projet adaptée à notre pipeline Big Data :

```
data-pipeline/
├── docker/                          # Configuration Docker Compose
│   ├── base.yml                     # Configuration de base
│   ├── dev.yml                      # Configuration de
développement
│   ├── test.yml                     # Configuration de test
│   ├── prod.yml                     # Configuration de production
│   └── config/
# Fichiers de configuration des services
```

```
│       ├── hadoop/              # Configuration Hadoop
│       ├── spark/               # Configuration Spark
│       ├── flink/               # Configuration Flink
│       ├── kafka/               # Configuration Kafka
│       ├── cassandra/           # Configuration Cassandra
│       └── airflow/             # Configuration Airflow
├── src/                         # Code source du pipeline
│   ├── extractors/              # Extracteurs de données
│   ├── transformers/            # Transformateurs de données
│   │   ├── batch/               # Transformations batch (Spark)
│   │   └── streaming/           # Transformations streaming
(Flink)
│   └── loaders/                 # Chargeurs de données
├── dags/                        # DAGs Airflow
├── notebooks/                   # Notebooks Jupyter pour
l'exploration
├── scripts/                     # Scripts utilitaires
├── test/                        # Tests
│   ├── unit/                    # Tests unitaires
│   ├── integration/             # Tests d'intégration
│   └── data/                    # Données de test
└── README.md                    # Documentation du projet
```

Installation des prérequis

Avant de commencer, assurez-vous que les outils suivants sont installés sur votre machine de développement :

1. **Docker** : Pour exécuter les conteneurs
2. **Docker Compose** : Pour orchestrer les conteneurs
3. **Git** : Pour la gestion du code source

Installation sur Ubuntu

```
# Mise à jour du système
sudo apt update && sudo apt upgrade -y

# Installation de Docker
sudo apt install -y apt-transport-https ca-certificates curl
software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://
download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt update
sudo apt install -y docker-ce
```

```
# Ajout de l'utilisateur courant au groupe docker
sudo usermod -aG docker $USER
newgrp docker

# Installation de Docker Compose
sudo curl -L "https://github.com/docker/compose/releases/
download/v2.15.1/docker-compose-$(uname -s)-$(uname -m)" -o /
usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Installation de Git
sudo apt install -y git
```

Installation sur macOS

```
 # Installation de Homebrew (si non installé)
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/HEAD/install.sh)"

# Installation de Docker Desktop (inclut Docker Compose)
brew install --cask docker

# Installation de Git
brew install git
```

Installation sur Windows

1. Téléchargez et installez Docker Desktop
2. Téléchargez et installez Git

Configuration de l'environnement Docker Compose

Création des fichiers Docker Compose

Commençons par créer les fichiers Docker Compose nécessaires. Créez d'abord le répertoire du projet et les sous-répertoires :

```
 mkdir -p data-pipeline/docker/config/
{hadoop,spark,flink,kafka,cassandra,airflow}
cd data-pipeline
```

Fichier `docker/base.yml`

Ce fichier contient la configuration de base commune à tous les environnements :

```yaml
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.3.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    volumes:
      - zookeeper_data:/var/lib/zookeeper/data
      - zookeeper_log:/var/lib/zookeeper/log
    networks:
      - data_pipeline_network

  kafka:
    image: confluentinc/cp-kafka:7.3.0
    hostname: kafka
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:
9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    volumes:
      - kafka_data:/var/lib/kafka/data
    networks:
      - data_pipeline_network

  schema-registry:
    image: confluentinc/cp-schema-registry:7.3.0
    hostname: schema-registry
    container_name: schema-registry
    depends_on:
      - kafka
    ports:
      - "8081:8081"
    environment:
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
```

```
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: kafka:9092
    networks:
      - data_pipeline_network

  namenode:
    image: bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8
    container_name: namenode
    restart: always
    ports:
      - "9870:9870"
      - "9000:9000"
    environment:
      - CLUSTER_NAME=hadoop-cluster
    env_file:
      - ./config/hadoop/hadoop.env
    volumes:
      - hadoop_namenode:/hadoop/dfs/name
    networks:
      - data_pipeline_network

  datanode:
    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
    container_name: datanode
    restart: always
    depends_on:
      - namenode
    environment:
      - SERVICE_PRECONDITION=namenode:9870
    env_file:
      - ./config/hadoop/hadoop.env
    volumes:
      - hadoop_datanode:/hadoop/dfs/data
    networks:
      - data_pipeline_network

  spark-master:
    image: bde2020/spark-master:3.3.0-hadoop3.3
    container_name: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
    environment:
      - INIT_DAEMON_STEP=setup_spark
    volumes:
      - ./config/spark/spark-defaults.conf:/spark/conf/spark-
defaults.conf
      - spark_data:/spark/data
    networks:
      - data_pipeline_network

  spark-worker:
    image: bde2020/spark-worker:3.3.0-hadoop3.3
```

```
    container_name: spark-worker
    depends_on:
      - spark-master
    environment:
      - SPARK_MASTER=spark://spark-master:7077
      - SPARK_WORKER_CORES=2
      - SPARK_WORKER_MEMORY=2G
    volumes:
      - spark_data:/spark/data
    networks:
      - data_pipeline_network

  jobmanager:
    image: flink:1.16.0
    container_name: jobmanager
    ports:
      - "8081:8081"
    command: jobmanager
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager
    volumes:
      - ./config/flink/flink-conf.yaml:/opt/flink/conf/flink-
conf.yaml
      - flink_data:/opt/flink/data
    networks:
      - data_pipeline_network

  taskmanager:
    image: flink:1.16.0
    container_name: taskmanager
    depends_on:
      - jobmanager
    command: taskmanager
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager
    volumes:
      - ./config/flink/flink-conf.yaml:/opt/flink/conf/flink-
conf.yaml
      - flink_data:/opt/flink/data
    networks:
      - data_pipeline_network

  cassandra:
    image: cassandra:4.1.0
    container_name: cassandra
    ports:
      - "9042:9042"
    environment:
      - CASSANDRA_CLUSTER_NAME=data-pipeline-cluster
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
      - CASSANDRA_DC=datacenter1
    volumes:
```

```
      - cassandra_data:/var/lib/cassandra
    networks:
      - data_pipeline_network

  airflow-webserver:
    image: apache/airflow:2.5.1
    container_name: airflow-webserver
    depends_on:
      - postgres
    environment:
      - AIRFLOW__CORE__EXECUTOR=LocalExecutor
      - AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgresql+psycopg2://
airflow:airflow@postgres/airflow
      -
AIRFLOW__CORE__FERNET_KEY=46BKJoQYlPPOexq0OhDZnIlNepKFf87WFwLbfzqDDho=
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
      -
AIRFLOW__API__AUTH_BACKENDS=airflow.api.auth.backend.basic_auth
    volumes:
      - ./dags:/opt/airflow/dags
      - ./logs:/opt/airflow/logs
      - ./plugins:/opt/airflow/plugins
      - ./config/airflow/airflow.cfg:/opt/airflow/airflow.cfg
    ports:
      - "8082:8080"
    command: webserver
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8080/
health"]
      interval: 30s
      timeout: 10s
      retries: 5
    networks:
      - data_pipeline_network

  airflow-scheduler:
    image: apache/airflow:2.5.1
    container_name: airflow-scheduler
    depends_on:
      - postgres
    environment:
      - AIRFLOW__CORE__EXECUTOR=LocalExecutor
      - AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgresql+psycopg2://
airflow:airflow@postgres/airflow
      -
AIRFLOW__CORE__FERNET_KEY=46BKJoQYlPPOexq0OhDZnIlNepKFf87WFwLbfzqDDho=
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
    volumes:
      - ./dags:/opt/airflow/dags
      - ./logs:/opt/airflow/logs
      - ./plugins:/opt/airflow/plugins
      - ./config/airflow/airflow.cfg:/opt/airflow/airflow.cfg
```

```
    command: scheduler
    networks:
      - data_pipeline_network

  postgres:
    image: postgres:14
    container_name: postgres
    environment:
      - POSTGRES_USER=airflow
      - POSTGRES_PASSWORD=airflow
      - POSTGRES_DB=airflow
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - data_pipeline_network

networks:
  data_pipeline_network:
    driver: bridge

volumes:
  zookeeper_data:
  zookeeper_log:
  kafka_data:
  hadoop_namenode:
  hadoop_datanode:
  spark_data:
  flink_data:
  cassandra_data:
  postgres_data:
```

Fichier `docker/dev.yml`

Ce fichier étend la configuration de base pour l'environnement de développement :

```
version: '3.8'

services:
  # Surcharge des services pour le développement
  kafka:
    environment:
      # Configuration supplémentaire pour le développement
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
      KAFKA_DELETE_TOPIC_ENABLE: "true"
      KAFKA_LOG_RETENTION_HOURS: 24

  # Outils supplémentaires pour le développement
  kafka-ui:
```

```
    image: provectuslabs/kafka-ui:latest
    container_name: kafka-ui
    depends_on:
      - kafka
      - schema-registry
    ports:
      - "8083:8080"
    environment:
      - KAFKA_CLUSTERS_0_NAME=local
      - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=kafka:9092
      - KAFKA_CLUSTERS_0_SCHEMAREGISTRY=http://schema-registry:
8081
    networks:
      - data_pipeline_network

  hadoop-ui:
    image: bde2020/hdfs-filebrowser:3.2.1
    container_name: hadoop-ui
    depends_on:
      - namenode
    environment:
      - NAMENODE_HOST=namenode
      - NAMENODE_PORT=9870
    ports:
      - "8084:8085"
    networks:
      - data_pipeline_network

  cassandra-web:
    image: delermando/cassandra-web:latest
    container_name: cassandra-web
    depends_on:
      - cassandra
    ports:
      - "8085:8083"
    environment:
      - CASSANDRA_HOST=cassandra
      - CASSANDRA_PORT=9042
    networks:
      - data_pipeline_network

  jupyter:
    image: jupyter/pyspark-notebook:spark-3.3.0
    container_name: jupyter
    ports:
      - "8888:8888"
    environment:
      - SPARK_OPTS=--master=spark://spark-master:7077
    volumes:
      - ./notebooks:/home/jovyan/work
    networks:
      - data_pipeline_network
```

Création des fichiers de configuration

Configuration Hadoop

Créez le fichier `docker/config/hadoop/hadoop.env` :

```
CORE_CONF_fs_defaultFS=hdfs://namenode:9000
CORE_CONF_hadoop_http_staticuser_user=root
CORE_CONF_hadoop_proxyuser_hue_hosts=*
CORE_CONF_hadoop_proxyuser_hue_groups=*
CORE_CONF_io_compression_codecs=org.apache.hadoop.io.compress.SnappyCodec

HDFS_CONF_dfs_webhdfs_enabled=true
HDFS_CONF_dfs_permissions_enabled=false
HDFS_CONF_dfs_namenode_datanode_registration_ip___hostname___check=false
```

Configuration Spark

Créez le fichier `docker/config/spark/spark-defaults.conf` :

```
spark.master                     spark://spark-master:7077
spark.driver.memory              2g
spark.executor.memory            2g
spark.executor.cores             2
spark.default.parallelism        4
spark.sql.shuffle.partitions     4
spark.serializer
org.apache.spark.serializer.KryoSerializer
spark.hadoop.fs.defaultFS        hdfs://namenode:9000
```

Configuration Flink

Créez le fichier `docker/config/flink/flink-conf.yaml` :

```
jobmanager.rpc.address: jobmanager
jobmanager.rpc.port: 6123
jobmanager.memory.process.size: 1600m
taskmanager.memory.process.size: 1728m
taskmanager.numberOfTaskSlots: 2
parallelism.default: 2
state.backend: filesystem
state.checkpoints.dir: file:///opt/flink/data/checkpoints
state.savepoints.dir: file:///opt/flink/data/savepoints
execution.checkpointing.interval: 10000
execution.checkpointing.mode: EXACTLY_ONCE
```

Démarrage de l'environnement de développement

Une fois tous les fichiers créés, vous pouvez démarrer l'environnement de développement :

```
# Se placer dans le répertoire du projet
cd data-pipeline

# Créer les répertoires nécessaires
mkdir -p dags logs plugins notebooks

# Démarrer l'environnement de développement
docker-compose -f docker/base.yml -f docker/dev.yml up -d
```

Vérification de l'environnement

Après le démarrage, vérifiez que tous les services sont opérationnels :

```
docker-compose -f docker/base.yml -f docker/dev.yml ps
```

Vous devriez voir tous les services avec le statut "Up".

Accès aux interfaces web

Voici les URL pour accéder aux différentes interfaces web :

- **Hadoop NameNode** : http://localhost:9870
- **Spark Master** : http://localhost:8080
- **Flink JobManager** : http://localhost:8081
- **Kafka UI** : http://localhost:8083
- **Hadoop UI** : http://localhost:8084
- **Cassandra Web** : http://localhost:8085
- **Jupyter Notebook** : http://localhost:8888
- **Airflow** : http://localhost:8082

Initialisation d'Airflow

Pour initialiser Airflow et créer un utilisateur administrateur :

```
# Exécuter la commande d'initialisation dans le conteneur
Airflow
docker exec -it airflow-webserver airflow db init
```

```
# Créer un utilisateur administrateur
docker exec -it airflow-webserver airflow users create \
  --username admin \
  --firstname Admin \
  --lastname User \
  --role Admin \
  --email admin@example.com \
  --password admin
```

Configuration des connexions Airflow

Pour que Airflow puisse interagir avec les autres services, configurez les connexions suivantes :

Connexion Spark

```
docker exec -it airflow-webserver airflow connections add
'spark_default' \
  --conn-type 'spark' \
  --conn-host 'spark://spark-master' \
  --conn-port '7077'
```

Connexion HDFS

```
docker exec -it airflow-webserver airflow connections add
'hdfs_default' \
  --conn-type 'hdfs' \
  --conn-host 'namenode' \
  --conn-port '9000' \
  --conn-extra '{"use_ssl": false}'
```

Connexion Kafka

```
docker exec -it airflow-webserver airflow connections add
'kafka_default' \
  --conn-type 'kafka' \
  --conn-host 'kafka' \
  --conn-port '9092'
```

Connexion Cassandra

```
docker exec -it airflow-webserver airflow connections add
'cassandra_default' \
  --conn-type 'cassandra' \
  --conn-host 'cassandra' \
  --conn-port '9042'
```

Création des structures de données initiales

Création des topics Kafka

```
# Créer les topics Kafka nécessaires
docker exec -it kafka kafka-topics --create --topic
transactions --bootstrap-server kafka:9092 --partitions 3 --
replication-factor 1
docker exec -it kafka kafka-topics --create --topic products --
bootstrap-server kafka:9092 --partitions 3 --replication-factor
1
docker exec -it kafka kafka-topics --create --topic customers --
bootstrap-server kafka:9092 --partitions 3 --replication-factor
1
```

Création des répertoires HDFS

```
# Créer les répertoires HDFS nécessaires
docker exec -it namenode hdfs dfs -mkdir -p /data/raw
docker exec -it namenode hdfs dfs -mkdir -p /data/processed
docker exec -it namenode hdfs dfs -mkdir -p /data/enriched
docker exec -it namenode hdfs dfs -chmod -R 777 /data
```

Création du keyspace Cassandra

```
# Créer le keyspace et les tables Cassandra
docker exec -it cassandra cqlsh -e "
CREATE KEYSPACE IF NOT EXISTS pipeline WITH REPLICATION =
{'class': 'SimpleStrategy', 'replication_factor': 1};

CREATE TABLE IF NOT EXISTS pipeline.transactions (
  transaction_id UUID PRIMARY KEY,
  customer_id TEXT,
  product_id TEXT,
  transaction_date TIMESTAMP,
```

```
  amount DECIMAL,
  quantity INT
);

CREATE TABLE IF NOT EXISTS pipeline.customer_kpis (
  customer_id TEXT,
  date DATE,
  total_amount DECIMAL,
  transaction_count INT,
  avg_transaction_value DECIMAL,
  PRIMARY KEY (customer_id, date)
);
"
```

Installation des dépendances Python

Pour faciliter le développement, créez un fichier `requirements.txt` à la racine du projet :

```
# Kafka
confluent-kafka==2.1.0
kafka-python==2.0.2
fastavro==1.7.0

# Spark
pyspark==3.3.0
delta-spark==2.3.0

# Flink
apache-flink==1.16.0
pyflink==1.16.0

# Cassandra
cassandra-driver==3.25.0

# HDFS
hdfs==2.7.0

# Airflow
apache-airflow==2.5.1
apache-airflow-providers-apache-spark==4.0.0
apache-airflow-providers-apache-hdfs==3.2.0
apache-airflow-providers-apache-cassandra==3.1.0
apache-airflow-providers-apache-kafka==1.0.0

# Utilitaires
pandas==1.5.3
```

```
numpy==1.24.2
python-dotenv==1.0.0
```

Installez ces dépendances dans le conteneur Jupyter :

```
docker exec -it jupyter pip install -r /home/jovyan/work/
requirements.txt
```

Création d'un notebook de test

Pour vérifier que tout fonctionne correctement, créez un notebook Jupyter qui teste les connexions à tous les services :

```
# Test de connexion à tous les services

# 1. Test de connexion à Spark
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("TestConnections") \
    .master("spark://spark-master:7077") \
    .getOrCreate()

print("Connexion à Spark établie :")
print(f"Version : {spark.version}")
print(f"Application ID : {spark.sparkContext.applicationId}")

# 2. Test de connexion à HDFS
from hdfs import InsecureClient

hdfs_client = InsecureClient('http://namenode:9870')
print("\nConnexion à HDFS établie :")
print(f"Contenu du répertoire racine : {hdfs_client.list('/')}")

# 3. Test de connexion à Kafka
from kafka import KafkaProducer, KafkaConsumer
import json

producer = KafkaProducer(
    bootstrap_servers=['kafka:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

print("\nConnexion à Kafka établie :")
producer.send('test-topic', {'message': 'Hello Kafka!'})
producer.flush()
print("Message envoyé à Kafka")
```

```
# 4. Test de connexion à Cassandra
from cassandra.cluster import Cluster

cluster = Cluster(['cassandra'])
session = cluster.connect()
print("\nConnexion à Cassandra établie :")
print(f"Keyspaces disponibles : {session.execute('SELECT
keyspace_name FROM system_schema.keyspaces')}")

# 5. Test de connexion à Flink (via REST API)
import requests

try:
    response = requests.get('http://jobmanager:8081/overview')
    flink_info = response.json()
    print("\nConnexion à Flink établie :")
    print(f"Version : {flink_info['flink-version']}")
except Exception as e:
    print(f"\nErreur de connexion à Flink : {e}")

# Nettoyage
spark.stop()
cluster.shutdown()
```

Arrêt de l'environnement

Pour arrêter l'environnement de développement :

```
docker-compose -f docker/base.yml -f docker/dev.yml down
```

Pour arrêter l'environnement et supprimer tous les volumes (attention, cela supprimera toutes les données) :

```
docker-compose -f docker/base.yml -f docker/dev.yml down -v
```

Bonnes pratiques pour le développement

Gestion des secrets

Pour éviter de stocker des informations sensibles dans les fichiers Docker Compose, utilisez un fichier `.env` :

```
# Créer un fichier .env à la racine du projet
cat > .env << EOF
POSTGRES_USER=airflow
POSTGRES_PASSWORD=airflow
POSTGRES_DB=airflow
AIRFLOW_FERNET_KEY=46BKJoQYlPPOexq0OhDZnIlNepKFf87WFwLbfzqDDho=
EOF
```

Scripts utilitaires

Pour faciliter l'utilisation quotidienne, créez des scripts shell :

`scripts/start-dev.sh`

```
#!/bin/bash
docker-compose -f docker/base.yml -f docker/dev.yml up -d
echo "Environnement de développement démarré. Interfaces
disponibles :"
echo "- Hadoop NameNode : http://localhost:9870"
echo "- Spark Master : http://localhost:8080"
echo "- Flink JobManager : http://localhost:8081"
echo "- Kafka UI : http://localhost:8083"
echo "- Hadoop UI : http://localhost:8084"
echo "- Cassandra Web : http://localhost:8085"
echo "- Jupyter Notebook : http://localhost:8888"
echo "- Airflow : http://localhost:8082 (admin/admin)"
```

`scripts/stop-dev.sh`

```
#!/bin/bash
docker-compose -f docker/base.yml -f docker/dev.yml down
echo "Environnement de développement arrêté."
```

`scripts/reset-dev.sh`

```
#!/bin/bash
docker-compose -f docker/base.yml -f docker/dev.yml down -v
echo "Environnement de développement réinitialisé (volumes
supprimés)."
```

N'oubliez pas de rendre ces scripts exécutables :

```
chmod +x scripts/*.sh
```

Conclusion

Vous avez maintenant un environnement de développement complet pour votre pipeline de données Big Data, avec tous les composants nécessaires (Hadoop, Spark, Flink, Kafka, Cassandra, Airflow) configurés et prêts à être utilisés. Cet environnement est :

1. **Reproductible** : Tous les développeurs peuvent avoir exactement le même environnement
2. **Isolé** : Chaque composant s'exécute dans son propre conteneur
3. **Configurable** : Les configurations peuvent être facilement modifiées
4. **Évolutif** : De nouveaux services peuvent être ajoutés selon les besoins

Dans les parties suivantes, nous utiliserons cet environnement pour implémenter les différentes étapes du pipeline : extraction, transformation et chargement des données, en tirant parti des capacités de traitement batch et temps réel offertes par Spark et Flink.

Correction Partie 2 : Implémentation des extracteurs avec Kafka et outils Big Data

L'extraction des données est la première étape cruciale de tout pipeline de données. Dans un environnement Big Data avec des besoins de traitement batch et temps réel, cette étape devient encore plus stratégique. Cette partie détaille l'implémentation d'extracteurs de données utilisant Kafka et d'autres outils Big Data pour alimenter notre pipeline.

Architecture d'extraction dans un contexte Big Data

Dans notre architecture Lambda, l'extraction des données doit répondre à deux besoins distincts :

1. **Extraction batch** : Récupération périodique de grands volumes de données pour des traitements différés
2. **Extraction temps réel** : Capture continue des événements pour un traitement immédiat

Kafka joue un rôle central dans cette architecture, servant à la fois de : - **Point d'entrée** pour les données en temps réel - **Buffer** pour absorber les pics de charge - **Source**

unique de vérité pour les données entrantes - **Interface** entre les systèmes sources et notre pipeline

Types d'extracteurs à implémenter

Nous allons implémenter trois types d'extracteurs, chacun adapté à une source de données spécifique :

1. **Extracteur de fichiers CSV vers HDFS** : Pour les données batch historiques
2. **Extracteur API REST vers Kafka** : Pour les données en temps réel ou périodiques
3. **Extracteur de base de données vers Kafka et HDFS** : Pour les données relationnelles, avec support du CDC (Change Data Capture)

Implémentation des extracteurs

1. Extracteur de fichiers CSV vers HDFS

Cet extracteur permet de charger des fichiers CSV dans HDFS pour un traitement batch ultérieur avec Spark.

Structure du code

```
 # src/extractors/csv_to_hdfs_extractor.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Extracteur de fichiers CSV vers HDFS.

Ce script permet de charger des fichiers CSV dans HDFS pour un
traitement batch.
Il supporte le partitionnement par date et la validation des
données.
"""

import os
import argparse
import logging
import json
from datetime import datetime
import pandas as pd
from hdfs import InsecureClient
from pyspark.sql import SparkSession

# Configuration du logging
```

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
)
logger = logging.getLogger(__name__)

class CsvToHdfsExtractor:
    """Extracteur de fichiers CSV vers HDFS."""

    def __init__(self, config_path):
        """
        Initialise l'extracteur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
        self.config = self._load_config(config_path)
        self.hdfs_client = InsecureClient(
            f'http://{self.config["hdfs_host"]}:
{self.config["hdfs_port"]}'
        )

        # Initialisation de Spark pour la validation et la
conversion en Parquet
        self.spark = SparkSession.builder \
            .appName("CsvToHdfsExtractor") \
            .master(self.config.get("spark_master", "local[*]"))
\
            .config("spark.hadoop.fs.defaultFS", f"hdfs://
{self.config['hdfs_host']}:9000") \
            .getOrCreate()

        logger.info(f"Extracteur initialisé avec la
configuration: {self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)
```

```
            # Validation des paramètres requis
            required_params = ['hdfs_host', 'hdfs_port',
'hdfs_target_dir']
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def extract(self, input_dir, date=None):
        """
        Extrait les fichiers CSV du répertoire spécifié et les
charge dans HDFS.

        Args:
            input_dir (str): Répertoire contenant les fichiers
CSV à extraire
            date (str, optional): Date des données au format
YYYY-MM-DD
                                  Si non spécifiée, utilise la
date du jour

        Returns:
            list: Liste des chemins HDFS où les fichiers ont été
chargés
        """
        # Déterminer la date à utiliser
        if date:
            try:
                process_date = datetime.strptime(date, '%Y-%m-
%d')
            except ValueError:
                logger.error(f"Format de date invalide: {date}.
Utilisation de la date du jour.")
                process_date = datetime.now()
        else:
            process_date = datetime.now()

        date_str = process_date.strftime('%Y-%m-%d')
        year, month, day = date_str.split('-')

        # Construire le chemin HDFS cible avec partitionnement
par date
        hdfs_target = os.path.join(
            self.config['hdfs_target_dir'],
            f"year={year}/month={month}/day={day}"
```

```
        )

        # S'assurer que le répertoire cible existe
        try:
            self.hdfs_client.makedirs(hdfs_target)
            logger.info(f"Répertoire HDFS créé: {hdfs_target}")
        except Exception as e:
            logger.warning(f"Erreur lors de la création du
répertoire HDFS (peut-être déjà existant): {e}")

        # Lister les fichiers CSV dans le répertoire source
        csv_files = [f for f in os.listdir(input_dir) if
f.endswith('.csv')]
        if not csv_files:
            logger.warning(f"Aucun fichier CSV trouvé dans
{input_dir}")
            return []

        logger.info(f"Fichiers CSV trouvés: {csv_files}")

        # Traiter chaque fichier CSV
        hdfs_paths = []
        for csv_file in csv_files:
            input_path = os.path.join(input_dir, csv_file)
            file_name = os.path.splitext(csv_file)[0]
            hdfs_path = os.path.join(hdfs_target,
f"{file_name}.parquet")

            try:
                # Charger le CSV avec Spark pour validation et
conversion en Parquet
                logger.info(f"Chargement du fichier
{input_path}")
                df = self.spark.read.csv(
                    input_path,
                    header=True,
                    inferSchema=True,
                    sep=self.config.get('csv_delimiter', ','),
                    nullValue=self.config.get('null_value', ''),
                    mode=self.config.get('parse_mode',
'PERMISSIVE')
                )

                # Appliquer les validations configurées
                if 'validations' in self.config:
                    df = self._apply_validations(df)

                # Écrire en format Parquet dans HDFS
                logger.info(f"Écriture du fichier {hdfs_path}")
                df.write.mode('overwrite').parquet(f"hdfs://
{self.config['hdfs_host']}:9000{hdfs_path}")
```

```
                hdfs_paths.append(hdfs_path)
                logger.info(f"Fichier chargé avec succès:
{hdfs_path}")

                # Collecter des métriques
                row_count = df.count()
                logger.info(f"Nombre de lignes chargées:
{row_count}")

            except Exception as e:
                logger.error(f"Erreur lors du traitement du
fichier {csv_file}: {e}")

        return hdfs_paths

    def _apply_validations(self, df):
        """
        Applique les validations configurées au DataFrame.

        Args:
            df (DataFrame): DataFrame Spark à valider

        Returns:
            DataFrame: DataFrame validé
        """
        validations = self.config['validations']

        # Filtrer les lignes nulles dans les colonnes spécifiées
        if 'required_columns' in validations:
            for column in validations['required_columns']:
                df = df.filter(df[column].isNotNull())
                logger.info(f"Validation: Filtrage des valeurs
nulles dans la colonne {column}")

        # Appliquer des filtres personnalisés
        if 'filters' in validations:
            for filter_expr in validations['filters']:
                df = df.filter(filter_expr)
                logger.info(f"Validation: Application du filtre
{filter_expr}")

        return df

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.spark:
            self.spark.stop()
            logger.info("Session Spark arrêtée")

def main():
    """Point d'entrée principal du script."""
    parser = argparse.ArgumentParser(description='Extracteur de
```

```
fichiers CSV vers HDFS')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--input', required=True,
help='Répertoire contenant les fichiers CSV')
    parser.add_argument('--date', help='Date des données (YYYY-
MM-DD)')

    args = parser.parse_args()

    try:
        extractor = CsvToHdfsExtractor(args.config)
        hdfs_paths = extractor.extract(args.input, args.date)
        extractor.close()

        if hdfs_paths:
            logger.info(f"Extraction terminée avec succès.
Fichiers chargés: {hdfs_paths}")
            return 0
        else:
            logger.warning("Extraction terminée, mais aucun
fichier n'a été chargé")
            return 1
    except Exception as e:
        logger.error(f"Erreur lors de l'extraction: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration de l'extracteur CSV

```
{
  "hdfs_host": "namenode",
  "hdfs_port": "9870",
  "hdfs_target_dir": "/data/raw/transactions",
  "spark_master": "spark://spark-master:7077",
  "csv_delimiter": ",",
  "null_value": "",
  "parse_mode": "PERMISSIVE",
  "validations": {
    "required_columns": ["transaction_id", "transaction_date",
"amount"],
    "filters": ["amount > 0"]
  }
}
```

2. Extracteur API REST vers Kafka

Cet extracteur permet de capturer des données depuis une API REST et de les publier dans Kafka pour un traitement en temps réel.

Structure du code

```
 # src/extractors/api_to_kafka_extractor.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Extracteur API REST vers Kafka.

Ce script permet de capturer des données depuis une API REST et
de les publier
dans Kafka pour un traitement en temps réel.
"""

import os
import argparse
import logging
import json
import time
import requests
from datetime import datetime
from confluent_kafka import Producer
from confluent_kafka.serialization import StringSerializer
from confluent_kafka.schema_registry import SchemaRegistryClient
from confluent_kafka.schema_registry.avro import AvroSerializer

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
)
logger = logging.getLogger(__name__)

class ApiToKafkaExtractor:
    """Extracteur API REST vers Kafka."""

    def __init__(self, config_path):
        """
        Initialise l'extracteur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
```

```
        self.config = self._load_config(config_path)
        self.producer = self._create_kafka_producer()

        # Initialiser le sérialiseur Avro si un schéma est
fourni
        if 'schema_registry_url' in self.config and
'avro_schema_path' in self.config:
            self.use_avro = True
            self.serializer = self._create_avro_serializer()
        else:
            self.use_avro = False
            self.serializer = StringSerializer()

        logger.info(f"Extracteur initialisé avec la
configuration: {self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)

            # Validation des paramètres requis
            required_params = ['api_base_url',
'kafka_bootstrap_servers', 'kafka_topic']
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def _create_kafka_producer(self):
        """
        Crée et configure un producteur Kafka.

        Returns:
            Producer: Instance du producteur Kafka configuré
```

```
        """
        producer_config = {
            'bootstrap.servers':
self.config['kafka_bootstrap_servers'],
            'client.id': self.config.get('client_id', 'api-
extractor'),
            'acks': self.config.get('acks', 'all'),
            'retries': self.config.get('retries', 5),
            'retry.backoff.ms':
self.config.get('retry_backoff_ms', 500)
        }

        return Producer(producer_config)

    def _create_avro_serializer(self):
        """
        Crée un sérialiseur Avro pour les messages Kafka.

        Returns:
            AvroSerializer: Instance du sérialiseur Avro
        """
        try:
            # Charger le schéma Avro
            with open(self.config['avro_schema_path'], 'r') as
f:
                avro_schema_str = f.read()

            # Configurer le client Schema Registry
            schema_registry_conf = {'url':
self.config['schema_registry_url']}
            schema_registry_client =
SchemaRegistryClient(schema_registry_conf)

            # Créer le sérialiseur Avro
            return AvroSerializer(schema_registry_client,
avro_schema_str)
        except Exception as e:
            logger.error(f"Erreur lors de la création du
sérialiseur Avro: {e}")
            self.use_avro = False
            return StringSerializer()

    def _delivery_report(self, err, msg):
        """
        Callback appelé pour chaque message produit pour
confirmer la livraison.

        Args:
            err: Erreur éventuelle
            msg: Message produit
        """
        if err is not None:
```

```
            logger.error(f"Échec de livraison du message:
{err}")
        else:
            logger.debug(f"Message livré à {msg.topic()}
[{msg.partition()}] @ {msg.offset()}")

    def _make_api_request(self, endpoint, params=None):
        """
        Effectue une requête à l'API REST.

        Args:
            endpoint (str): Point de terminaison de l'API
            params (dict, optional): Paramètres de la requête

        Returns:
            dict: Réponse de l'API
        """
        url = f"{self.config['api_base_url']}/{endpoint}"

        # Ajouter les headers d'authentification si configurés
        headers = {}
        if 'api_key' in self.config:
            headers['Authorization'] = f"Bearer
{self.config['api_key']}"

        try:
            response = requests.get(url, params=params,
headers=headers, timeout=30)
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Erreur lors de la requête API: {e}")
            raise

    def _process_pagination(self, endpoint, params=None):
        """
        Gère la pagination de l'API.

        Args:
            endpoint (str): Point de terminaison de l'API
            params (dict, optional): Paramètres initiaux de la
requête

        Yields:
            dict: Chaque élément de la réponse paginée
        """
        if params is None:
            params = {}

        # Configuration de la pagination
        page = 1
        page_size = self.config.get('page_size', 100)
```

```
        params['page'] = page
        params['limit'] = page_size

        while True:
            logger.info(f"Récupération de la page
{page} (taille {page_size})")
            response = self._make_api_request(endpoint, params)

            # Extraire les données selon la structure de la
réponse
            data_key = self.config.get('response_data_key',
'data')
            items = response.get(data_key, [])

            if not items:
                logger.info("Aucun élément supplémentaire
trouvé")
                break

            # Traiter chaque élément
            for item in items:
                yield item

            # Vérifier s'il y a une page suivante
            total_key = self.config.get('response_total_key',
'total')
            if total_key in response:
                total = response[total_key]
                if page * page_size >= total:
                    logger.info(f"Toutes les pages récupérées
({page} pages, {total} éléments)")
                    break

            # Passer à la page suivante
            page += 1
            params['page'] = page

    def extract(self, endpoint, params=None, continuous=False,
interval=60):
        """
        Extrait les données de l'API et les publie dans Kafka.

        Args:
            endpoint (str): Point de terminaison de l'API
            params (dict, optional): Paramètres de la requête
            continuous (bool): Si True, exécute l'extraction en
continu
            interval (int): Intervalle entre les extractions en
mode continu (secondes)

        Returns:
            int: Nombre d'éléments extraits
```

```
        """
        if params is None:
            params = {}

        count = 0

        try:
            while True:
                start_time = time.time()
                logger.info(f"Début de l'extraction depuis
{endpoint}")

                # Ajouter un timestamp aux paramètres si
configuré
                if self.config.get('add_timestamp', False):
                    params['timestamp'] = int(time.time())

                # Traiter les éléments paginés
                for item in self._process_pagination(endpoint,
params):
                    # Publier l'élément dans Kafka
                    self._publish_to_kafka(item)
                    count += 1

                logger.info(f"{count} éléments extraits et
publiés dans Kafka")

                # En mode non continu, terminer après une
extraction
                if not continuous:
                    break

                # Calculer le temps d'attente
                elapsed = time.time() - start_time
                wait_time = max(0, interval - elapsed)
                logger.info(f"Attente de {wait_time:.1f}
secondes avant la prochaine extraction")
                time.sleep(wait_time)

        except KeyboardInterrupt:
            logger.info("Extraction interrompue par
l'utilisateur")
        except Exception as e:
            logger.error(f"Erreur lors de l'extraction: {e}")
            raise

        return count

    def _publish_to_kafka(self, data):
        """
        Publie les données dans Kafka.
```

```
        Args:
            data (dict): Données à publier
        """
        try:
            # Déterminer la clé du message
            key = None
            if 'key_field' in self.config and
self.config['key_field'] in data:
                key =
str(data[self.config['key_field']]).encode('utf-8')

            # Sérialiser les données
            if self.use_avro:
                value = self.serializer(data)
            else:
                value = json.dumps(data).encode('utf-8')

            # Publier dans Kafka
            self.producer.produce(
                topic=self.config['kafka_topic'],
                key=key,
                value=value,
                on_delivery=self._delivery_report
            )

            # Déclencher l'envoi des messages en attente
            self.producer.poll(0)

        except Exception as e:
            logger.error(f"Erreur lors de la publication dans
Kafka: {e}")

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.producer:
            # S'assurer que tous les messages sont envoyés
            logger.info("Attente de l'envoi de tous les
messages...")
            self.producer.flush()
            logger.info("Producteur Kafka fermé")

def main():
    """Point d'entrée principal du script."""
    parser =
argparse.ArgumentParser(description='Extracteur API REST vers
Kafka')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--endpoint', required=True,
help='Point de terminaison de l\'API')
    parser.add_argument('--params', help='Paramètres de la
requête au format JSON')
```

```
    parser.add_argument('--continuous', action='store_true',
help='Exécution continue')
    parser.add_argument('--interval', type=int, default=60,
help='Intervalle entre les extractions (secondes)')

    args = parser.parse_args()

    try:
        extractor = ApiToKafkaExtractor(args.config)

        # Charger les paramètres depuis JSON si fournis
        params = None
        if args.params:
            params = json.loads(args.params)

        count = extractor.extract(args.endpoint, params,
args.continuous, args.interval)
        extractor.close()

        logger.info(f"Extraction terminée avec succès. {count}
éléments extraits.")
        return 0
    except Exception as e:
        logger.error(f"Erreur lors de l'extraction: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration de l'extracteur API

```
{
  "api_base_url": "https://api.example.com/v1",
  "api_key": "your-api-key",
  "kafka_bootstrap_servers": "kafka:9092",
  "kafka_topic": "products",
  "client_id": "products-api-extractor",
  "acks": "all",
  "retries": 5,
  "retry_backoff_ms": 500,
  "schema_registry_url": "http://schema-registry:8081",
  "avro_schema_path": "/path/to/product_schema.avsc",
  "key_field": "product_id",
  "page_size": 100,
  "response_data_key": "items",
  "response_total_key": "total",
  "add_timestamp": true
}
```

Schéma Avro pour les produits

```
{
  "type": "record",
  "name": "Product",
  "namespace": "com.example.data",
  "fields": [
    {"name": "product_id", "type": "string"},
    {"name": "name", "type": "string"},
    {"name": "description", "type": ["null", "string"],
"default": null},
    {"name": "category", "type": ["null", "string"], "default":
null},
    {"name": "price", "type": "double"},
    {"name": "stock", "type": ["null", "int"], "default": null},
    {"name": "created_at", "type": {"type": "long",
"logicalType": "timestamp-millis"}},
    {"name": "updated_at", "type": {"type": "long",
"logicalType": "timestamp-millis"}}
  ]
}
```

3. Extracteur de base de données vers Kafka et HDFS

Cet extracteur permet de capturer des données depuis une base de données relationnelle, avec support du CDC (Change Data Capture) pour les mises à jour en temps réel.

Structure du code

```
 # src/extractors/db_to_kafka_hdfs_extractor.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Extracteur de base de données vers Kafka et HDFS.

Ce script permet d'extraire des données d'une base de données
relationnelle
et de les publier à la fois dans Kafka (pour le traitement temps
réel) et
dans HDFS (pour le traitement batch), avec support du CDC.
"""

import os
import argparse
import logging
import json
```

```python
import time
from datetime import datetime, timedelta
import pandas as pd
from sqlalchemy import create_engine, text
from confluent_kafka import Producer
from hdfs import InsecureClient
from pyspark.sql import SparkSession

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
)
logger = logging.getLogger(__name__)

class DbToKafkaHdfsExtractor:
    """Extracteur de base de données vers Kafka et HDFS."""

    def __init__(self, config_path):
        """
        Initialise l'extracteur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
        self.config = self._load_config(config_path)
        self.db_engine = self._create_db_engine()
        self.producer = self._create_kafka_producer()
        self.hdfs_client = InsecureClient(
            f'http://{self.config["hdfs_host"]}:
{self.config["hdfs_port"]}'
        )

        # Initialisation de Spark pour l'écriture en Parquet
        self.spark = SparkSession.builder \
            .appName("DbToKafkaHdfsExtractor") \
            .master(self.config.get("spark_master", "local[*]"))
\
            .config("spark.hadoop.fs.defaultFS", f"hdfs://
{self.config['hdfs_host']}:9000") \
            .getOrCreate()

        logger.info(f"Extracteur initialisé avec la
configuration: {self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
```

```
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)

            # Validation des paramètres requis
            required_params = [
                'db_connection_string', 'table_name',
'kafka_bootstrap_servers',
                'kafka_topic', 'hdfs_host', 'hdfs_port',
'hdfs_target_dir'
            ]
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def _create_db_engine(self):
        """
        Crée et configure une connexion à la base de données.

        Returns:
            Engine: Instance du moteur SQLAlchemy
        """
        try:
            engine =
create_engine(self.config['db_connection_string'])
            # Tester la connexion
            with engine.connect() as conn:
                conn.execute(text("SELECT 1"))
            logger.info("Connexion à la base de données établie
avec succès")
            return engine
        except Exception as e:

logger.error(f"Erreur lors de la connexion à la base de données:
{e}")
            raise

    def _create_kafka_producer(self):
```

```
        """
        Crée et configure un producteur Kafka.

        Returns:
            Producer: Instance du producteur Kafka configuré
        """
        producer_config = {
            'bootstrap.servers':
self.config['kafka_bootstrap_servers'],
            'client.id': self.config.get('client_id', 'db-
extractor'),
            'acks': self.config.get('acks', 'all'),
            'retries': self.config.get('retries', 5),
            'retry.backoff.ms':
self.config.get('retry_backoff_ms', 500)
        }

        return Producer(producer_config)

    def _delivery_report(self, err, msg):
        """
        Callback appelé pour chaque message produit pour
confirmer la livraison.

        Args:
            err: Erreur éventuelle
            msg: Message produit
        """
        if err is not None:
            logger.error(f"Échec de livraison du message:
{err}")
        else:
            logger.debug(f"Message livré à {msg.topic()}
[{msg.partition()}] @ {msg.offset()}")

    def extract_batch(self, date=None, full_refresh=False):
        """
        Extrait les données en mode batch et les stocke dans
HDFS.

        Args:
            date (str, optional): Date des données au format
YYYY-MM-DD
                                  Si non spécifiée, utilise la
date du jour
            full_refresh (bool): Si True, extrait toutes les
données sans filtre de date

        Returns:
            str: Chemin HDFS où les données ont été stockées
        """
        # Déterminer la date à utiliser
```

```
        if date:
            try:
                process_date = datetime.strptime(date, '%Y-%m-
%d')
            except ValueError:
                logger.error(f"Format de date invalide: {date}.
Utilisation de la date du jour.")
                process_date = datetime.now()
        else:
            process_date = datetime.now()

        date_str = process_date.strftime('%Y-%m-%d')
        year, month, day = date_str.split('-')

        # Construire le chemin HDFS cible avec partitionnement
par date
        hdfs_target = os.path.join(
            self.config['hdfs_target_dir'],
            f"year={year}/month={month}/day={day}"
        )

        # S'assurer que le répertoire cible existe
        try:
            self.hdfs_client.makedirs(hdfs_target)
            logger.info(f"Répertoire HDFS créé: {hdfs_target}")
        except Exception as e:
            logger.warning(f"Erreur lors de la création du
répertoire HDFS (peut-être déjà existant): {e}")

        # Construire la requête SQL
        table_name = self.config['table_name']
        date_column = self.config.get('date_column',
'updated_at')

        if full_refresh:
            query = f"SELECT * FROM {table_name}"
            logger.info(f"Mode full refresh: extraction de
toutes les données de {table_name}")
        else:
            yesterday = process_date - timedelta(days=1)
            query = f"SELECT * FROM {table_name} WHERE
{date_column} >= '{yesterday.strftime('%Y-%m-%d')}'"
            logger.info(f"Extraction des données mises à jour
depuis {yesterday.strftime('%Y-%m-%d')}")

        try:
            # Exécuter la requête et charger les données
            logger.info(f"Exécution de la requête: {query}")
            df = pd.read_sql(query, self.db_engine)

            if df.empty:
                logger.warning("Aucune donnée extraite")
```

```
                return None

            logger.info(f"{len(df)} lignes extraites")

            # Convertir le DataFrame pandas en DataFrame Spark
            spark_df = self.spark.createDataFrame(df)

            # Chemin complet HDFS
            hdfs_path = os.path.join(hdfs_target,
f"{table_name}.parquet")

            # Écrire en format Parquet dans HDFS
            logger.info(f"Écriture des données dans HDFS:
{hdfs_path}")
            spark_df.write.mode('overwrite').parquet(f"hdfs://
{self.config['hdfs_host']}:9000{hdfs_path}")


logger.info(f"Données écrites avec succès dans HDFS:
{hdfs_path}")
            return hdfs_path

        except Exception as e:
            logger.error(f"Erreur lors de l'extraction batch:
{e}")
            raise

    def extract_cdc(self, continuous=False, interval=60):
        """
        Extrait les changements de données (CDC) et les publie
dans Kafka.

        Args:
            continuous (bool): Si True, exécute l'extraction en
continu
            interval (int): Intervalle entre les extractions en
mode continu (secondes)

        Returns:
            int: Nombre d'éléments extraits
        """
        count = 0
        last_timestamp = None

        try:
            while True:
                start_time = time.time()

                # Construire la requête SQL pour le CDC
                table_name = self.config['table_name']
                date_column = self.config.get('date_column',
'updated_at')
```

```
                if last_timestamp:
                    query = f"SELECT * FROM {table_name} WHERE
{date_column} > '{last_timestamp}'"
                    logger.info(f"Extraction des changements
depuis {last_timestamp}")
                else:
                    # Première exécution, limiter aux N
dernières minutes
                    minutes =
self.config.get('initial_cdc_minutes', 60)
                    query = f"SELECT * FROM {table_name} WHERE
{date_column} > NOW() - INTERVAL '{minutes}' MINUTE"
                    logger.info(f"Extraction initiale des
changements des {minutes} dernières minutes")

                # Exécuter la requête et traiter les résultats
                df = pd.read_sql(query, self.db_engine)

                if not df.empty:
                    logger.info(f"{len(df)} changements
détectés")

                    # Mettre à jour le dernier timestamp traité
                    if date_column in df.columns:
                        last_timestamp = df[date_column].max()

                    # Publier chaque ligne dans Kafka
                    for _, row in df.iterrows():
                        self._publish_to_kafka(row.to_dict())
                        count += 1
                else:
                    logger.info("Aucun changement détecté")

                # En mode non continu, terminer après une
extraction
                if not continuous:
                    break

                # Calculer le temps d'attente
                elapsed = time.time() - start_time
                wait_time = max(0, interval - elapsed)
                logger.info(f"Attente de {wait_time:.1f}
secondes avant la prochaine extraction")
                time.sleep(wait_time)

        except KeyboardInterrupt:
            logger.info("Extraction CDC interrompue par
l'utilisateur")
        except Exception as e:
            logger.error(f"Erreur lors de l'extraction CDC:
{e}")
```

```
            raise

        return count

    def _publish_to_kafka(self, data):
        """
        Publie les données dans Kafka.

        Args:
            data (dict): Données à publier
        """
        try:
            # Déterminer la clé du message
            key = None
            if 'key_field' in self.config and
self.config['key_field'] in data:
                key =
str(data[self.config['key_field']]).encode('utf-8')

            # Sérialiser les données
            value = json.dumps(data).encode('utf-8')

            # Publier dans Kafka
            self.producer.produce(
                topic=self.config['kafka_topic'],
                key=key,
                value=value,
                on_delivery=self._delivery_report
            )

            # Déclencher l'envoi des messages en attente
            self.producer.poll(0)

        except Exception as e:
            logger.error(f"Erreur lors de la publication dans
Kafka: {e}")

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.producer:
            # S'assurer que tous les messages sont envoyés
            logger.info("Attente de l'envoi de tous les
messages...")
            self.producer.flush()
            logger.info("Producteur Kafka fermé")

        if self.spark:
            self.spark.stop()
            logger.info("Session Spark arrêtée")

def main():
    """Point d'entrée principal du script."""
```

```
    parser = argparse.ArgumentParser(description='Extracteur de
base de données vers Kafka et HDFS')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--mode', choices=['batch', 'cdc',
'both'], default='both',
                        help='Mode d\'extraction: batch, cdc ou
both')
    parser.add_argument('--date',
help='Date des données pour le mode batch (YYYY-MM-DD)')
    parser.add_argument('--full-refresh', action='store_true',
help='Extraction complète en mode batch')
    parser.add_argument('--continuous', action='store_true',
help='Exécution continue en mode CDC')
    parser.add_argument('--interval', type=int, default=60,
help='Intervalle entre les extractions CDC (secondes)')

    args = parser.parse_args()

    try:
        extractor = DbToKafkaHdfsExtractor(args.config)

        if args.mode in ['batch', 'both']:
            hdfs_path = extractor.extract_batch(args.date,
args.full_refresh)
            logger.info(f"Extraction batch terminée. Chemin
HDFS: {hdfs_path}")

        if args.mode in ['cdc', 'both']:
            count = extractor.extract_cdc(args.continuous,
args.interval)
            logger.info(f"Extraction CDC terminée. {count}
changements publiés dans Kafka.")

        extractor.close()
        return 0
    except Exception as e:
        logger.error(f"Erreur lors de l'extraction: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration de l'extracteur de base de données

```
{
  "db_connection_string": "postgresql://
username:password@postgres:5432/customers_db",
  "table_name": "customers",
  "date_column": "updated_at",
```

```
  "kafka_bootstrap_servers": "kafka:9092",
  "kafka_topic": "customers",
  "client_id": "customers-db-extractor",
  "key_field": "customer_id",
  "hdfs_host": "namenode",
  "hdfs_port": "9870",
  "hdfs_target_dir": "/data/raw/customers",
  "spark_master": "spark://spark-master:7077",
  "initial_cdc_minutes": 60
}
```

Intégration avec Kafka Connect

Pour certaines sources de données, Kafka Connect offre une solution plus robuste et évolutive que les extracteurs personnalisés. Voici comment configurer Kafka Connect pour l'extraction de données.

Configuration de Kafka Connect dans Docker Compose

Ajoutez le service Kafka Connect à votre fichier `docker/base.yml` :

```
  kafka-connect:
    image: confluentinc/cp-kafka-connect:7.3.0
    container_name: kafka-connect
    depends_on:
      - kafka
      - schema-registry
    ports:
      - "8083:8083"
    environment:
      CONNECT_BOOTSTRAP_SERVERS: kafka:9092
      CONNECT_REST_PORT: 8083
      CONNECT_GROUP_ID: "connect-cluster"
      CONNECT_CONFIG_STORAGE_TOPIC: "connect-configs"
      CONNECT_OFFSET_STORAGE_TOPIC: "connect-offsets"
      CONNECT_STATUS_STORAGE_TOPIC: "connect-status"
      CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_KEY_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_VALUE_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_INTERNAL_KEY_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_INTERNAL_VALUE_CONVERTER:
"org.apache.kafka.connect.json.JsonConverter"
      CONNECT_REST_ADVERTISED_HOST_NAME: "kafka-connect"
```

```
      CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/
confluent-hub-components"
    volumes:
      - ./config/kafka-connect/connectors:/usr/share/confluent-
hub-components
    networks:
      - data_pipeline_network
    command:
      - bash
      - -c
      - |
        echo "Installing connectors..."
        confluent-hub install --no-prompt debezium/debezium-
connector-postgresql:2.1.3
        confluent-hub install --no-prompt confluentinc/kafka-
connect-jdbc:10.7.0
        confluent-hub install --no-prompt confluentinc/kafka-
connect-hdfs3:1.1.16
        echo "Starting Kafka Connect..."
        /etc/confluent/docker/run
```

Configuration d'un connecteur JDBC pour l'extraction de base de données

Créez un fichier de configuration pour le connecteur JDBC :

```
{
  "name": "jdbc-customers-source",
  "config": {
    "connector.class":
"io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "jdbc:postgresql://postgres:5432/
customers_db",
    "connection.user": "username",
    "connection.password": "password",
    "topic.prefix": "jdbc-",
    "table.whitelist": "customers",
    "mode": "timestamp",
    "timestamp.column.name": "updated_at",
    "validate.non.null": "false",
    "transforms": "createKey,extractInt",
    "transforms.createKey.type":
"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields": "customer_id",
    "transforms.extractInt.type":
"org.apache.kafka.connect.transforms.ExtractField$Key",
    "transforms.extractInt.field": "customer_id"
```

```
  }
}
```

Configuration d'un connecteur Debezium pour le CDC

Créez un fichier de configuration pour le connecteur Debezium :

```
{
  "name": "debezium-postgres-source",
  "config": {
    "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "username",
    "database.password": "password",
    "database.dbname": "customers_db",
    "database.server.name": "postgres",
    "table.include.list": "public.customers",
    "plugin.name": "pgoutput",
    "publication.name": "dbz_publication",
    "slot.name": "dbz_slot",
    "snapshot.mode": "initial",
    "transforms": "unwrap",
    "transforms.unwrap.type":
"io.debezium.transforms.ExtractNewRecordState",
    "transforms.unwrap.drop.tombstones": "false",
    "transforms.unwrap.delete.handling.mode": "rewrite"
  }
}
```

Configuration d'un connecteur HDFS pour l'archivage

Créez un fichier de configuration pour le connecteur HDFS :

```
{
  "name": "hdfs-sink",
  "config": {
    "connector.class":
"io.confluent.connect.hdfs3.Hdfs3SinkConnector",
    "topics": "customers",
    "hdfs.url": "hdfs://namenode:9000",
    "flush.size": 1000,
    "rotate.interval.ms": 3600000,
    "path.format": "'year='YYYY/'month='MM/'day='dd",
    "partitioner.class":
"io.confluent.connect.storage.partitioner.TimeBasedPartitioner",
    "timestamp.extractor": "Record",
```

```
    "partition.duration.ms": 86400000,
    "locale": "en-US",
    "timezone": "UTC",
    "format.class":
"io.confluent.connect.hdfs3.parquet.ParquetFormat"
  }
}
```

Déploiement des connecteurs

Créez un script pour déployer les connecteurs :

```
#!/bin/bash
# scripts/deploy_connectors.sh

# Attendre que Kafka Connect soit prêt
echo "Attente de Kafka Connect..."
while ! curl -s kafka-connect:8083/connectors > /dev/null; do
  sleep 1
done
echo "Kafka Connect est prêt!"

# Déployer le connecteur JDBC
echo "Déploiement du connecteur JDBC..."
curl -X POST -H "Content-Type: application/json" --data @config/
kafka-connect/jdbc-source.json http://kafka-connect:8083/
connectors

# Déployer le connecteur Debezium
echo "Déploiement du connecteur Debezium..."
curl -X POST -H "Content-Type: application/json" --data @config/
kafka-connect/debezium-source.json http://kafka-connect:8083/
connectors

# Déployer le connecteur HDFS
echo "Déploiement du connecteur HDFS..."
curl -X POST -H "Content-Type: application/json" --data @config/
kafka-connect/hdfs-sink.json http://kafka-connect:8083/
connectors

echo "Tous les connecteurs ont été déployés!"
```

Monitoring des extracteurs

Pour surveiller les extracteurs, nous pouvons utiliser Prometheus et Grafana.

Configuration de Prometheus pour les métriques Kafka

Créez un fichier `docker/config/prometheus/prometheus.yml` :

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'kafka'
    static_configs:
      - targets: ['kafka:9092']

  - job_name: 'kafka-connect'
    static_configs:
      - targets: ['kafka-connect:8083']
```

Dashboard Grafana pour les extracteurs

Créez un fichier `docker/config/grafana/provisioning/dashboards/extractors.json` avec un dashboard pour surveiller les extracteurs.

Livrables attendus

1. **Code source des extracteurs** : Les trois extracteurs implémentés (CSV vers HDFS, API vers Kafka, DB vers Kafka et HDFS)
2. **Fichiers de configuration** : Les fichiers JSON de configuration pour chaque extracteur
3. **Configurations Kafka Connect** : Les fichiers de configuration pour les connecteurs Kafka Connect
4. **Scripts de déploiement** : Les scripts pour déployer et tester les extracteurs
5. **Documentation** : Un document expliquant l'architecture d'extraction, les choix techniques et les bonnes pratiques

Conclusion

L'implémentation des extracteurs de données avec Kafka et les outils Big Data permet de construire un pipeline capable de traiter à la fois des données batch et temps réel. Les extracteurs développés offrent une grande flexibilité pour s'adapter à différentes sources de données, tout en s'intégrant parfaitement dans notre architecture Lambda.

En utilisant Kafka comme backbone central, nous assurons une séparation claire entre l'ingestion des données et leur traitement, ce qui permet une meilleure scalabilité et résilience du système. L'intégration avec HDFS garantit également la persistance des données pour les traitements batch avec Spark.

Dans la partie suivante, nous verrons comment transformer ces données brutes en utilisant Spark pour le traitement batch et Flink pour le traitement temps réel.

Correction Partie 3 : Implémentation des transformations avec Spark et Flink

La transformation des données est l'étape où la valeur est réellement extraite des données brutes. Dans notre architecture Big Data hybride, cette étape est divisée en deux volets : le traitement batch avec Spark et le traitement temps réel avec Flink.

Principes de transformation dans une architecture Lambda

Dans notre architecture Lambda, les transformations sont effectuées en parallèle par deux systèmes :

1. **Couche batch (Spark)** : Traite l'ensemble des données historiques (stockées dans HDFS) pour générer des vues batch complètes et précises. Ces traitements sont généralement périodiques (quotidiens, hebdomadaires).
2. **Couche temps réel (Flink)** : Traite les données en continu (provenant de Kafka) pour fournir des vues temps réel approximatives mais rapides. Ces traitements sont continus.

La couche de service (Cassandra) est ensuite utilisée pour combiner les résultats de ces deux couches et offrir une vue unifiée aux utilisateurs.

Implémentation des transformations batch avec Spark

Spark est idéal pour les transformations batch à grande échelle grâce à son moteur de traitement distribué et ses APIs riches (DataFrame, SQL, MLlib).

Structure d'un job Spark

Un job Spark typique pour la transformation batch suit les étapes suivantes :

1. **Initialisation** : Création d'une `SparkSession`.
2. **Lecture des données** : Chargement des données brutes depuis HDFS (ou d'autres sources).
3. **Transformations** : Application des logiques métier (nettoyage, enrichissement, agrégation).
4. **Écriture des résultats** : Sauvegarde des données transformées dans HDFS (format Parquet) et/ou Cassandra.

Exemple de job Spark pour transformer les transactions

```
 # src/transformers/batch/transaction_transformer.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Job Spark pour la transformation batch des données de
transactions.

Ce job lit les données brutes depuis HDFS, applique des
transformations
(nettoyage, enrichissement avec données clients et produits), et
écrit
les résultats dans HDFS (Parquet) et Cassandra.
"""

import argparse
import logging
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, udf, year, month,
dayofmonth, lit, when, avg, count, sum as _sum
from pyspark.sql.types import StringType, DecimalType

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s"
)
logger = logging.getLogger(__name__)

class TransactionTransformer:
    """Classe pour la transformation batch des transactions."""

    def __init__(self, spark, config):
        """
```

```
        Initialise le transformateur.

        Args:
            spark (SparkSession): Session Spark
            config (dict): Configuration du job
        """
        self.spark = spark
        self.config = config
        logger.info(f"Transformateur initialisé avec la
configuration: {config}")

    def run(self, date):
        """
        Exécute le processus de transformation pour une date
donnée.

        Args:
            date (str): Date de traitement au format YYYY-MM-DD
        """
        year, month, day = date.split("-")

        # 1. Lire les données brutes
        transactions_df =
self._read_data(f"{self.config["hdfs_base_path"]}/transactions/
year={year}/month={month}/day={day}")
        customers_df =
self._read_data(f"{self.config["hdfs_base_path"]}/customers/
year={year}/month={month}/day={day}")
        products_df =
self._read_data(f"{self.config["hdfs_base_path"]}/products/
year={year}/month={month}/day={day}")

        if transactions_df is None or customers_df is None or
products_df is None:
            logger.warning("Données sources manquantes pour la
date {date}. Arrêt du traitement.")
            return

        # 2. Nettoyer les données
        transactions_cleaned_df =
self._clean_transactions(transactions_df)
        customers_cleaned_df =
self._clean_customers(customers_df)
        products_cleaned_df = self._clean_products(products_df)

        # 3. Enrichir les transactions
        enriched_df = self._enrich_transactions(
            transactions_cleaned_df,
            customers_cleaned_df,
            products_cleaned_df
        )
```

```
        # 4. Calculer les agrégations (KPIs)
        customer_kpis_df =
self._calculate_customer_kpis(enriched_df)

        # 5. Écrire les résultats
        self._write_data(
            enriched_df,
            f"{self.config["hdfs_output_path"]}/
enriched_transactions/year={year}/month={month}/day={day}",
            "pipeline.transactions_enriched",
            self.config["cassandra_keyspace"]
        )
        self._write_data(
            customer_kpis_df,
            f"{self.config["hdfs_output_path"]}/customer_kpis/
year={year}/month={month}/day={day}",
            "pipeline.customer_kpis",
            self.config["cassandra_keyspace"]
        )

        logger.info(f"Transformation terminée pour la date
{date}")

    def _read_data(self, path):
        """Lit les données Parquet depuis HDFS."""
        logger.info(f"Lecture des données depuis: {path}")
        try:
            return self.spark.read.parquet(f"hdfs://
{self.config["hdfs_host"]}:9000{path}")
        except Exception as e:
            logger.error(f"Erreur lors de la lecture de {path}:
{e}")
            return None

    def _clean_transactions(self, df):
        """Nettoie les données de transactions."""
        logger.info("Nettoyage des transactions...")
        # Supprimer les doublons
        df = df.dropDuplicates(["transaction_id"])
        # Filtrer les montants invalides
        df = df.filter(col("amount") > 0)
        # Convertir les types
        df = df.withColumn("amount",
col("amount").cast(DecimalType(10, 2)))
        df = df.withColumn("transaction_date",
col("transaction_date").cast("timestamp"))
        return df

    def _clean_customers(self, df):
        """Nettoie les données clients."""
        logger.info("Nettoyage des clients...")
        df = df.dropDuplicates(["customer_id"])
```

```
        # Standardiser les emails
        df = df.withColumn("email", udf(lambda x:
x.lower().strip() if x else None, StringType())(col("email")))
        return df

    def _clean_products(self, df):
        """Nettoie les données produits."""
        logger.info("Nettoyage des produits...")
        df = df.dropDuplicates(["product_id"])
        # Remplacer les prix négatifs par null
        df = df.withColumn("price", when(col("price") < 0,
None).otherwise(col("price")))
        return df

    def _enrich_transactions(self, transactions_df,
customers_df, products_df):

"""Enrichit les transactions avec les données clients et
produits."""
        logger.info("Enrichissement des transactions...")
        enriched_df = transactions_df \
            .join(customers_df.select("customer_id", "city",
"country"), "customer_id", "left") \
            .join(products_df.select("product_id", "category",
"price"), "product_id", "left") \
            .withColumnRenamed("price", "product_price")
        return enriched_df

    def _calculate_customer_kpis(self, enriched_df):
        """Calcule les KPIs par client."""
        logger.info("Calcul des KPIs clients...")
        kpis_df = enriched_df \
            .groupBy("customer_id",
col("transaction_date").cast("date").alias("date")) \
            .agg(
                _sum("amount").alias("total_amount"),

count("transaction_id").alias("transaction_count"),
                avg("amount").alias("avg_transaction_value")
            )
        return kpis_df

    def _write_data(self, df, hdfs_path, cassandra_table,
cassandra_keyspace):
        """Écrit les données dans HDFS et Cassandra."""
        # Écrire dans HDFS
        logger.info(f"Écriture dans HDFS: {hdfs_path}")
        try:
            df.write.mode("overwrite").parquet(f"hdfs://
{self.config["hdfs_host"]}:9000{hdfs_path}")
            logger.info("Écriture HDFS réussie")
        except Exception as e:
```

```
            logger.error(f"Erreur lors de l\'écriture dans HDFS
{hdfs_path}: {e}")

        # Écrire dans Cassandra
        logger.info(f"Écriture dans Cassandra:
{cassandra_keyspace}.{cassandra_table}")
        try:
            df.write \
                .format("org.apache.spark.sql.cassandra") \
                .options(table=cassandra_table,
keyspace=cassandra_keyspace) \
                .mode("append") \
                .option("spark.cassandra.connection.host",
self.config["cassandra_host"]) \
                .option("spark.cassandra.connection.port",
self.config["cassandra_port"]) \
                .save()
            logger.info("Écriture Cassandra réussie")
        except Exception as e:
            logger.error(f"Erreur lors de l\'écriture dans
Cassandra {cassandra_keyspace}.{cassandra_table}: {e}")

def main():
    """Point d\'entrée principal du script."""
    parser = argparse.ArgumentParser(description=\"Job Spark
pour la transformation batch\")
    parser.add_argument("--config", required=True,
help=\"Chemin vers le fichier de configuration JSON\")
    parser.add_argument("--date", required=True, help=\"Date de
traitement (YYYY-MM-DD)\")

    args = parser.parse_args()

    # Charger la configuration
    with open(args.config, \'r\') as f:
        config = json.load(f)

    # Initialiser SparkSession
    spark = SparkSession.builder \
        .appName(f"TransactionTransformer-{args.date}") \
        .config("spark.jars.packages",
"com.datastax.spark:spark-cassandra-connector_2.12:3.3.0") \
        .config("spark.sql.extensions",
"com.datastax.spark.connector.CassandraSparkExtensions") \
        .getOrCreate()

    try:
        transformer = TransactionTransformer(spark, config)
        transformer.run(args.date)
    except Exception as e:
        logger.error(f"Erreur lors de l\'exécution du job de
transformation: {e}")
```

```
    finally:
        spark.stop()

if __name__ == \'__main__\':
    main()
```

Configuration du job Spark config/csv_extractor_config.json

```
{
  "hdfs_host": "namenode",
  "hdfs_port": "9000",
  "hdfs_base_path": "/data/raw",
  "hdfs_output_path": "/data/processed",
  "cassandra_host": "cassandra",
  "cassandra_port": "9042",
  "cassandra_keyspace": "pipeline"
}
```

Soumission du job Spark

Le job peut être soumis via `spark-submit` :

```
docker exec -it spark-master spark-submit \
  --master spark://spark-master:7077 \
  --packages com.datastax.spark:spark-cassandra-
connector_2.12:3.3.0 \
  /path/to/src/transformers/batch/transaction_transformer.py \
  --config /path/to/config/transformer_config.json \
  --date 2023-01-15
```

Implémentation des transformations temps réel avec Flink

Flink est utilisé pour traiter les flux de données provenant de Kafka en temps réel, en appliquant des transformations et en maintenant un état si nécessaire.

Structure d'une application Flink

Une application Flink typique pour le traitement de flux :

1. **Initialisation** : Création d'un `StreamExecutionEnvironment`.
2. **Source de données** : Connexion à un topic Kafka.
3. **Transformations** : Application des opérations (map, filter, keyBy, window, etc.).

4. **Sink** : Écriture des résultats dans Kafka, Cassandra ou d'autres systèmes.

Exemple d'application Flink pour calculer des KPIs en temps réel

```
 # src/transformers/streaming/realtime_kpi_calculator.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Application Flink pour le calcul de KPIs en temps réel.

Cette application lit les transactions depuis Kafka, calcule des
KPIs
(ex: montant total par catégorie sur une fenêtre glissante) et
écrit
les résultats dans Cassandra.
"""

import argparse
import logging
import json
from pyflink.common import WatermarkStrategy, Time
from pyflink.common.typeinfo import Types
from pyflink.datastream import StreamExecutionEnvironment,
TimeCharacteristic
from pyflink.datastream.connectors import FlinkKafkaConsumer,
FlinkKafkaProducer
from pyflink.datastream.formats.json import
JsonRowDeserializationSchema, JsonRowSerializationSchema
from pyflink.datastream.window import
TumblingProcessingTimeWindows
from pyflink.table import StreamTableEnvironment, DataTypes
from pyflink.table.expressions import col, lit
from pyflink.table.udf import udf

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s"
)
logger = logging.getLogger(__name__)

# UDF pour extraire la catégorie (simplifié)
@udf(result_type=DataTypes.STRING())
def get_category(product_id):
    # En production, utiliser une source externe (ex: lookup
table)
    if product_id and product_id.startswith("P1"): return
"Electronics"
    if product_id and product_id.startswith("P2"): return
```

```
"Clothing"
    return "Other"

def realtime_kpi_calculator(config):
    """Définit et exécute le job Flink."""

    # 1. Initialiser l'environnement Flink
    env = StreamExecutionEnvironment.get_execution_environment()

env.set_stream_time_characteristic(TimeCharacteristic.ProcessingTime)
    env.set_parallelism(config.get("parallelism", 1))

    # Configuration du checkpointing pour la tolérance aux
pannes
    if config.get("enable_checkpointing", True):

env.enable_checkpointing(config.get("checkpoint_interval_ms",
60000))

env.get_checkpoint_config().set_checkpointing_mode(CheckpointingMode.EXACT

env.get_checkpoint_config().set_min_pause_between_checkpoints(config.get("
10000))

env.get_checkpoint_config().set_checkpoint_timeout(config.get("checkpoint_
120000))

env.get_checkpoint_config().set_max_concurrent_checkpoints(1)

env.get_checkpoint_config().enable_externalized_checkpoints(
            ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION
        )
        # Configurer le backend d'état (ex: HDFS)
        # env.get_state_backend(FsStateBackend(f"hdfs://
{config['hdfs_host']}:9000/flink-checkpoints"))

    # Créer l'environnement Table API
    t_env =
StreamTableEnvironment.create(stream_execution_environment=env)

    # 2. Définir la source Kafka
    kafka_props = {
        "bootstrap.servers": config["kafka_bootstrap_servers"],
        "group.id": config.get("kafka_group_id", "flink-kpi-
calculator")
    }

    # Définir le schéma des données entrantes
    source_type_info = Types.ROW_NAMED(
        ["transaction_id", "customer_id", "product_id",
"transaction_date", "amount", "quantity"],
        [Types.STRING(), Types.STRING(), Types.STRING(),
```

```
Types.SQL_TIMESTAMP(), Types.BIG_DEC(), Types.INT()]
    )

    kafka_consumer = FlinkKafkaConsumer(
        topics=config["kafka_input_topic"],

deserialization_schema=JsonRowDeserializationSchema.builder().type_info(sc
        properties=kafka_props
    )
    kafka_consumer.set_start_from_latest() # Ou
set_start_from_earliest()

    # Créer le DataStream
    transaction_stream =
env.add_source(kafka_consumer).name("KafkaTransactionSource")

    # Convertir le DataStream en Table
    transaction_table = t_env.from_data_stream(
        transaction_stream,
        col("transaction_id"),
        col("customer_id"),
        col("product_id"),
        col("transaction_date"),
        col("amount"),
        col("quantity"),
        col("proctime").proctime() # Ajouter une colonne de
temps de traitement
    )

    # 3. Appliquer les transformations
    # Ajouter la catégorie du produit
    t_env.create_temporary_function("get_category",
get_category)
    kpi_table =
transaction_table.add_columns(col("product_id").get_category().alias("cate

    # Définir la fenêtre temporelle (ex: 1 minute glissante)
    window =
Tumbling.over(lit(1).minutes).on(col("proctime")).alias("w")

    # Calculer les KPIs par catégorie dans la fenêtre
    kpi_results = kpi_table \
        .window(window) \
        .group_by(col("w"), col("category")) \
        .select(
            col("w").start.alias("window_start"),
            col("w").end.alias("window_end"),
            col("category"),
            col("amount").sum.alias("total_amount"),

col("transaction_id").count.alias("transaction_count")
        )
```

```
    # 4. Définir le Sink Cassandra
    # Note: L'écriture directe vers Cassandra depuis PyFlink
Table API est complexe.
    # Une approche courante est de convertir la Table en
DataStream et d'utiliser un Sink personnalisé
    # ou d'écrire dans un topic Kafka intermédiaire qui sera lu
par un connecteur Cassandra Sink.

    # Exemple: Écriture dans un topic Kafka de sortie
    output_type_info = Types.ROW_NAMED(
        ["window_start", "window_end", "category",
"total_amount", "transaction_count"],
        [Types.SQL_TIMESTAMP(), Types.SQL_TIMESTAMP(),
Types.STRING(), Types.BIG_DEC(), Types.LONG()]
    )

    kafka_producer = FlinkKafkaProducer(
        topic=config["kafka_output_topic"],

serialization_schema=JsonRowSerializationSchema.builder().with_type_info(o
        producer_config=kafka_props
    )

    # Convertir la Table en DataStream et écrire dans Kafka
    result_stream = t_env.to_data_stream(kpi_results)
    result_stream.add_sink(kafka_producer).name("KafkaKPISink")

    # 5. Exécuter le job
    logger.info("Démarrage du job Flink...")
    env.execute(config.get("job_name", "RealtimeKPICalculator"))

def main():
    """Point d'entrée principal du script."""
    parser = argparse.ArgumentParser(description="Application
Flink pour KPIs temps réel")
    parser.add_argument("--config", required=True, help="Chemin
vers le fichier de configuration JSON")

    args = parser.parse_args()

    # Charger la configuration
    with open(args.config, 'r') as f:
        config = json.load(f)

    try:
        realtime_kpi_calculator(config)
    except Exception as e:
        logger.error(f"Erreur lors de l'exécution du job Flink:
{e}")
```

```
if __name__ == '__main__':
    main()
```

Configuration de l'application Flink

```
{
  "kafka_bootstrap_servers": "kafka:9092",
  "kafka_input_topic": "transactions",
  "kafka_output_topic": "realtime_kpis",
  "kafka_group_id": "flink-kpi-calculator",
  "cassandra_host": "cassandra",
  "cassandra_port": "9042",
  "cassandra_keyspace": "pipeline",
  "cassandra_table": "realtime_category_kpis",
  "parallelism": 2,
  "enable_checkpointing": true,
  "checkpoint_interval_ms": 60000,
  "job_name": "RealtimeKPICalculator"
}
```

Soumission du job Flink

Le job peut être soumis au cluster Flink via la ligne de commande : config/flink_kpi_config.json

```
docker exec -it jobmanager flink run \
  -py /path/to/src/transformers/streaming/
realtime_kpi_calculator.py \
  --config /path/to/config/flink_kpi_config.json
```

Gestion de l'état dans Flink

Flink excelle dans la gestion de l'état pour les traitements de flux. L'état peut être utilisé pour :

- **Agrégations** : Maintenir des sommes, comptages, moyennes sur des fenêtres.
- **Jointures de flux** : Joindre des événements de différents flux basés sur une clé commune.
- **Détection de patterns** : Identifier des séquences d'événements spécifiques.

Flink assure la cohérence de l'état même en cas de panne grâce à son mécanisme de **checkpointing**. Les checkpoints sauvegardent périodiquement l'état de l'application dans un stockage durable (comme HDFS), permettant une récupération rapide et sans perte de données.

Réconciliation Batch et Temps Réel

Dans une architecture Lambda, il est crucial de pouvoir réconcilier les vues batch (exactes mais retardées) et temps réel (rapides mais potentiellement approximatives).

Plusieurs stratégies peuvent être employées :

1. **Remplacement périodique** : Les résultats batch écrasent les résultats temps réel dans la couche de service (Cassandra) à la fin de chaque cycle batch.
2. **Fusion dans la couche de service** : La couche de service (ou l'API qui l'expose) est responsable de fusionner les données des deux couches, en privilégiant les données batch lorsqu'elles sont disponibles pour une période donnée.
3. **Utilisation de Delta Lake** : Delta Lake sur HDFS peut être utilisé pour gérer les mises à jour incrémentielles et fournir une vue unifiée, simplifiant la réconciliation.

Bonnes pratiques pour les transformations Big Data

1. **Optimisation des performances Spark** :
2. Utiliser le partitionnement et le bucketing.
3. Choisir les bons formats de stockage (Parquet, ORC).
4. Optimiser les jointures (broadcast joins).
5. Gérer correctement la mémoire (caching, sérialisation Kryo).
6. **Optimisation des performances Flink** :
7. Choisir le bon backend d'état.
8. Configurer correctement le checkpointing.
9. Gérer la pression sur les sources et les sinks.
10. Utiliser les opérateurs asynchrones pour les I/O externes.
11. **Qualité des données** :
12. Implémenter des validations de données dans les jobs Spark et Flink.
13. Gérer les données invalides ou manquantes (quarantaine, imputation).
14. **Monitoring** :
15. Utiliser les UIs de Spark et Flink pour suivre l'exécution des jobs.
16. Exporter les métriques vers Prometheus/Grafana.
17. Mettre en place des alertes sur les erreurs et les dégradations de performance.

Conclusion

L'implémentation des transformations avec Spark et Flink permet de tirer pleinement parti de notre architecture Big Data hybride. Spark gère efficacement les traitements batch complexes sur de grands volumes de données historiques, tandis que Flink assure

le traitement en temps réel des flux de données avec une faible latence et des garanties de cohérence.

En combinant ces deux outils et en mettant en place des stratégies de réconciliation appropriées, nous pouvons construire une vue complète et à jour des données, répondant ainsi aux besoins variés des utilisateurs finaux.

La prochaine étape consistera à implémenter les chargeurs de données pour stocker les résultats finaux dans nos systèmes de stockage distribués (HDFS et Cassandra).

Correction Partie 4 : Implémentation des chargeurs pour Hadoop et Cassandra

Le chargement des données est l'étape finale du pipeline ETL, où les données transformées sont stockées dans des systèmes de stockage adaptés à leur utilisation. Dans notre architecture Big Data, nous utiliserons deux systèmes de stockage complémentaires :

1. **Hadoop HDFS** : Pour le stockage des données batch à long terme
2. **Apache Cassandra** : Pour le stockage des données optimisées pour les requêtes à faible latence

Cette partie détaille l'implémentation des chargeurs pour ces deux systèmes, en tenant compte des spécificités de chacun et des besoins de notre architecture Lambda.

Stratégie de chargement dans une architecture Lambda

Dans notre architecture Lambda, les données sont chargées dans différents systèmes selon leur nature et leur utilisation :

1. **Données brutes** : Stockées dans HDFS au format Parquet pour un accès efficace par les jobs Spark
2. **Données transformées batch** : Stockées dans HDFS (format Parquet) et dans Cassandra pour les requêtes rapides
3. **Données temps réel** : Stockées principalement dans Cassandra pour un accès à faible latence

Implémentation des chargeurs pour HDFS

HDFS est le système de fichiers distribué de Hadoop, optimisé pour le stockage de grands volumes de données et les traitements batch.

Chargeur de données Parquet vers HDFS

```
 # src/loaders/hdfs_parquet_loader.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Chargeur de données au format Parquet vers HDFS.

Ce script permet de charger des données au format Parquet dans
HDFS,
avec support du partitionnement et de la compression.
"""

import os
import argparse
import logging
import json
from datetime import datetime
from pyspark.sql import SparkSession

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s"
)
logger = logging.getLogger(__name__)

class HdfsParquetLoader:
    """Chargeur de données au format Parquet vers HDFS."""

    def __init__(self, config_path):
        """
        Initialise le chargeur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
        self.config = self._load_config(config_path)

        # Initialisation de Spark
        self.spark = SparkSession.builder \
```

```
            .appName("HdfsParquetLoader") \
            .master(self.config.get("spark_master", "local[*]"))
\
            .config("spark.hadoop.fs.defaultFS", f"hdfs://
{self.config['hdfs_host']}:9000") \
            .getOrCreate()


logger.info(f"Chargeur initialisé avec la configuration:
{self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)

            # Validation des paramètres requis
            required_params = ['hdfs_host', 'hdfs_target_dir']
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def load(self, input_path, date=None, partition_cols=None):
        """
        Charge les données au format Parquet dans HDFS.

        Args:
            input_path (str): Chemin vers les données à charger
            date (str, optional): Date des données au format
YYYY-MM-DD
                                  Si non spécifiée, utilise la
date du jour
            partition_cols (list, optional): Colonnes à utiliser
pour le partitionnement
```

```
        Returns:
            str: Chemin HDFS où les données ont été chargées
        """
        # Déterminer la date à utiliser
        if date:
            try:
                process_date = datetime.strptime(date, '%Y-%m-
%d')
            except ValueError:
                logger.error(f"Format de date invalide: {date}.
Utilisation de la date du jour.")
                process_date = datetime.now()
        else:
            process_date = datetime.now()

        date_str = process_date.strftime('%Y-%m-%d')
        year, month, day = date_str.split('-')

        # Construire le chemin HDFS cible avec partitionnement
par date
        if self.config.get("use_date_partitioning", True):
            hdfs_target = os.path.join(
                self.config['hdfs_target_dir'],
                f"year={year}/month={month}/day={day}"
            )
        else:
            hdfs_target = self.config['hdfs_target_dir']

        # Charger les données
        try:
            logger.info(f"Chargement des données depuis
{input_path}")
            df = self.spark.read.parquet(input_path)

            # Appliquer les transformations configurées
            if 'transformations' in self.config:
                df = self._apply_transformations(df)

            # Configurer les options d'écriture
            write_options = {}

            # Compression
            compression = self.config.get('compression',
'snappy')
            write_options['compression'] = compression

            # Mode d'écriture
            write_mode = self.config.get('write_mode',
'overwrite')

            # Écrire les données
```

```
            logger.info(f"Écriture des données dans HDFS:
{hdfs_target}")
            writer =
df.write.mode(write_mode).options(**write_options)

            # Partitionnement
            if partition_cols:
                logger.info(f"Partitionnement par colonnes:
{partition_cols}")
                writer.partitionBy(*partition_cols)

            # Écrire au format Parquet
            writer.parquet(f"hdfs://{self.config['hdfs_host']}:
9000{hdfs_target}")

            logger.info(f"Données chargées avec succès dans
HDFS: {hdfs_target}")

            # Collecter des métriques
            row_count = df.count()
            logger.info(f"Nombre de lignes chargées:
{row_count}")

            return hdfs_target

        except Exception as e:
            logger.error(f"Erreur lors du chargement des
données: {e}")
            raise

    def _apply_transformations(self, df):
        """
        Applique les transformations configurées au DataFrame.

        Args:
            df (DataFrame): DataFrame Spark à transformer

        Returns:
            DataFrame: DataFrame transformé
        """
        transformations = self.config['transformations']

        # Appliquer les filtres
        if 'filters' in transformations:
            for filter_expr in transformations['filters']:
                df = df.filter(filter_expr)
                logger.info(f"Transformation: Application du
filtre {filter_expr}")

        # Sélectionner les colonnes
        if 'select_columns' in transformations:
            df = df.select(*transformations['select_columns'])
```

```
logger.info(f"Transformation: Sélection des colonnes
{transformations['select_columns']}")

        # Renommer les colonnes
        if 'rename_columns' in transformations:
            for old_name, new_name in
transformations['rename_columns'].items():
                df = df.withColumnRenamed(old_name, new_name)
                logger.info(f"Transformation: Renommage de la
colonne {old_name} en {new_name}")

        return df

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.spark:
            self.spark.stop()
            logger.info("Session Spark arrêtée")

def main():
    """Point d'entrée principal du script."""
    parser = argparse.ArgumentParser(description='Chargeur de
données Parquet vers HDFS')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--input', required=True, help='Chemin
vers les données à charger')
    parser.add_argument('--date', help='Date des données (YYYY-
MM-DD)')
    parser.add_argument('--partition-cols', help='Colonnes de
partitionnement (séparées par des virgules)')

    args = parser.parse_args()

    try:
        loader = HdfsParquetLoader(args.config)

        # Traiter les colonnes de partitionnement
        partition_cols = None
        if args.partition_cols:
            partition_cols = [col.strip() for col in
args.partition_cols.split(',')]

        hdfs_path = loader.load(args.input, args.date,
partition_cols)
        loader.close()

        logger.info(f"Chargement terminé avec succès. Données
chargées dans: {hdfs_path}")
        return 0
    except Exception as e:
```

```
        logger.error(f"Erreur lors du chargement: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration du chargeur HDFS config\csv_extractor_config.json

```
{
  "hdfs_host": "namenode",
  "hdfs_target_dir": "/data/processed/transactions",
  "spark_master": "spark://spark-master:7077",
  "use_date_partitioning": true,
  "compression": "snappy",
  "write_mode": "overwrite",
  "transformations": {
    "filters": ["amount > 0"],
    "select_columns": ["transaction_id", "customer_id",
"product_id", "transaction_date", "amount", "quantity"],
    "rename_columns": {
      "transaction_date": "date"
    }
  }
}
```

Implémentation des chargeurs pour Cassandra

Cassandra est une base de données NoSQL distribuée, optimisée pour les écritures à haut débit et les lectures à faible latence. Elle est particulièrement adaptée pour stocker les résultats des transformations qui doivent être accessibles rapidement.

Chargeur de données vers Cassandra

```
 # src/loaders/cassandra_loader.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Chargeur de données vers Cassandra.

Ce script permet de charger des données dans Cassandra, avec
support
des différents modes d'écriture et des TTL (Time To Live).
"""

import os
```

```
import argparse
import logging
import json
import pandas as pd
from datetime import datetime
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
from cassandra.query import BatchStatement, ConsistencyLevel
from pyspark.sql import SparkSession

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %
(message)s"
)
logger = logging.getLogger(__name__)

class CassandraLoader:
    """Chargeur de données vers Cassandra."""

    def __init__(self, config_path):
        """
        Initialise le chargeur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
        self.config = self._load_config(config_path)
        self.cluster = self._connect_to_cassandra()
        self.session = self.cluster.connect()

        # Créer le keyspace si nécessaire
        self._create_keyspace_if_not_exists()

        # Utiliser le keyspace
        self.session.set_keyspace(self.config['keyspace'])

        # Créer la table si nécessaire
        if self.config.get('create_table_if_not_exists', True):
            self._create_table_if_not_exists()


logger.info(f"Chargeur initialisé avec la configuration:
{self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
```

```
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)

            # Validation des paramètres requis
            required_params = ['hosts', 'keyspace', 'table']
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def _connect_to_cassandra(self):
        """
        Établit une connexion au cluster Cassandra.

        Returns:
            Cluster: Instance du cluster Cassandra
        """
        try:
            # Configurer l'authentification si nécessaire
            auth_provider = None
            if 'username' in self.config and 'password' in
self.config:
                auth_provider = PlainTextAuthProvider(
                    username=self.config['username'],
                    password=self.config['password']
                )

            # Se connecter au cluster
            cluster = Cluster(
                contact_points=self.config['hosts'],
                port=self.config.get('port', 9042),
                auth_provider=auth_provider
            )

            logger.info(f"Connexion établie au cluster
Cassandra: {self.config['hosts']}")
            return cluster
        except Exception as e:
```

```
            logger.error(f"Erreur lors de la connexion à
Cassandra: {e}")
            raise

    def _create_keyspace_if_not_exists(self):
        """Crée le keyspace s'il n'existe pas déjà."""
        if self.config.get('create_keyspace_if_not_exists',
True):
            replication_strategy =
self.config.get('replication_strategy', 'SimpleStrategy')
            replication_factor =
self.config.get('replication_factor', 1)

            query = f"""
            CREATE KEYSPACE IF NOT EXISTS
{self.config['keyspace']}
            WITH REPLICATION = {{
                'class': '{replication_strategy}',
                'replication_factor': {replication_factor}
            }}
            """

            try:
                self.session.execute(query)
                logger.info(f"Keyspace
{self.config['keyspace']} créé ou déjà existant")
            except Exception as e:
                logger.error(f"Erreur lors de la création du
keyspace: {e}")
                raise

    def _create_table_if_not_exists(self):
        """Crée la table si elle n'existe pas déjà."""
        if 'table_schema' not in self.config:
            logger.warning("Schéma de table non spécifié,
impossible de créer la table")
            return

        table_schema = self.config['table_schema']
        columns = []
        primary_key = []

        # Construire la définition des colonnes
        for col_name, col_def in
table_schema['columns'].items():
            columns.append(f"{col_name} {col_def['type']}")
            if col_def.get('primary_key', False):
                primary_key.append(col_name)

        # Construire la clause PRIMARY KEY
        if primary_key:
            pk_clause = f"PRIMARY KEY ({',
```

```
'.join(primary_key)})"
            columns.append(pk_clause)

        # Construire la requête CREATE TABLE
        query = f"""
        CREATE TABLE IF NOT EXISTS {self.config['keyspace']}.
{self.config['table']} (
            {', '.join(columns)}
        )
        """

        # Ajouter les options de clustering si spécifiées
        if 'clustering_order' in table_schema:
            clustering_cols = []
            for col, order in
table_schema['clustering_order'].items():
                clustering_cols.append(f"{col} {order}")

            query += f" WITH CLUSTERING ORDER BY ({',
'.join(clustering_cols)})"

        try:
            self.session.execute(query)
            logger.info(f"Table {self.config['keyspace']}.
{self.config['table']} créée ou déjà existante")
        except Exception as e:
            logger.error(f"Erreur lors de la création de la
table: {e}")
            raise

    def load_from_parquet(self, input_path, batch_size=1000):
        """
        Charge des données depuis un fichier Parquet vers
Cassandra.

        Args:
            input_path (str): Chemin vers le fichier Parquet
            batch_size (int): Taille des lots pour les
insertions par batch

        Returns:
            int: Nombre de lignes chargées
        """
        try:
            # Initialiser Spark pour lire le fichier Parquet
            spark = SparkSession.builder \
                .appName("CassandraLoader") \
                .master("local[*]") \
                .getOrCreate()

            # Lire le fichier Parquet
            logger.info(f"Lecture du fichier Parquet:
```

```
{input_path}")
            df = spark.read.parquet(input_path)

            # Convertir en DataFrame pandas pour traitement par
lots
            pdf = df.toPandas()

            # Charger les données
            count = self._load_dataframe(pdf, batch_size)

            # Arrêter Spark
            spark.stop()

            return count
        except Exception as e:
            logger.error(f"Erreur lors du chargement depuis
Parquet: {e}")
            raise

    def load_from_csv(self, input_path, batch_size=1000,
delimiter=',', header=True):
        """
        Charge des données depuis un fichier CSV vers Cassandra.

        Args:
            input_path (str): Chemin vers le fichier CSV
            batch_size (int): Taille des lots pour les
insertions par batch
            delimiter (str): Délimiteur utilisé dans le fichier
CSV
            header (bool): Si True, la première ligne contient
les noms des colonnes

        Returns:
            int: Nombre de lignes chargées
        """
        try:
            # Lire le fichier CSV
            logger.info(f"Lecture du fichier CSV: {input_path}")
            df = pd.read_csv(input_path, delimiter=delimiter,
header=0 if header else None)

            # Si pas d'en-tête, générer des noms de colonnes
            if not header:
                df.columns = [f"col{i}" for i in
range(len(df.columns))]

            # Charger les données
            return self._load_dataframe(df, batch_size)
        except Exception as e:

logger.error(f"Erreur lors du chargement depuis CSV: {e}")
```

```
            raise

    def load_from_json(self, input_path, batch_size=1000):
        """
        Charge des données depuis un fichier JSON vers
Cassandra.

        Args:
            input_path (str): Chemin vers le fichier JSON
            batch_size (int): Taille des lots pour les
insertions par batch

        Returns:
            int: Nombre de lignes chargées
        """
        try:
            # Lire le fichier JSON
            logger.info(f"Lecture du fichier JSON:
{input_path}")
            df = pd.read_json(input_path)

            # Charger les données
            return self._load_dataframe(df, batch_size)
        except Exception as e:
            logger.error(f"Erreur lors du chargement depuis
JSON: {e}")
            raise

    def _load_dataframe(self, df, batch_size=1000):
        """
        Charge un DataFrame pandas dans Cassandra.

        Args:
            df (DataFrame): DataFrame pandas à charger
            batch_size (int): Taille des lots pour les
insertions par batch

        Returns:
            int: Nombre de lignes chargées
        """
        # Préparer la requête d'insertion
        columns = df.columns.tolist()
        placeholders = ', '.join(['%s'] * len(columns))

        query = f"""
        INSERT INTO {self.config['keyspace']}.
{self.config['table']}
        ({', '.join(columns)})
        VALUES ({placeholders})
        """

        # Ajouter TTL si spécifié
```

```
        if 'ttl' in self.config:
            query += f" USING TTL {self.config['ttl']}"

        # Préparer la requête
        prepared = self.session.prepare(query)

        # Définir le niveau de cohérence
        consistency = self.config.get('consistency_level',
'LOCAL_QUORUM')
        prepared.consistency_level = getattr(ConsistencyLevel,
consistency)

        # Charger par lots
        total_rows = len(df)
        loaded_rows = 0

        for i in range(0, total_rows, batch_size):
            batch = BatchStatement()
            end = min(i + batch_size, total_rows)

            for _, row in df.iloc[i:end].iterrows():
                # Convertir les valeurs NaN en None
                values = [None if pd.isna(val) else val for val
in row.values]
                batch.add(prepared, values)

            # Exécuter le batch
            self.session.execute(batch)
            loaded_rows += (end - i)
            logger.info(f"Chargé {loaded_rows}/{total_rows}
lignes")

        logger.info(f"Chargement terminé: {loaded_rows} lignes
chargées dans {self.config['keyspace']}.{self.config['table']}")
        return loaded_rows

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.cluster:
            self.cluster.shutdown()
            logger.info("Connexion Cassandra fermée")

def main():
    """Point d'entrée principal du script."""
    parser = argparse.ArgumentParser(description='Chargeur de
données vers Cassandra')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--input', required=True, help='Chemin
vers les données à charger')
    parser.add_argument('--format', choices=['parquet', 'csv',
'json'], default='parquet',
```

```
                          help='Format des données d\'entrée')
    parser.add_argument('--batch-size', type=int, default=1000,
help='Taille des lots pour les insertions')
    parser.add_argument('--delimiter', default=',',
help='Délimiteur pour les fichiers CSV')
    parser.add_argument('--no-header', action='store_true',
help='Le fichier CSV n\'a pas d\'en-tête')

    args = parser.parse_args()

    try:
        loader = CassandraLoader(args.config)

        if args.format == 'parquet':
            count = loader.load_from_parquet(args.input,
args.batch_size)
        elif args.format == 'csv':
            count = loader.load_from_csv(args.input,
args.batch_size, args.delimiter, not args.no_header)
        elif args.format == 'json':
            count = loader.load_from_json(args.input,
args.batch_size)

        loader.close()

        logger.info(f"Chargement terminé avec succès. {count}
lignes chargées.")
        return 0
    except Exception as e:
        logger.error(f"Erreur lors du chargement: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration du chargeur Cassandra config\transactions_cassandra_loader_config.json

```
{
  "hosts": ["cassandra"],
  "port": 9042,
  "keyspace": "pipeline",
  "table": "transactions_enriched",
  "create_keyspace_if_not_exists": true,
  "create_table_if_not_exists": true,
  "replication_strategy": "SimpleStrategy",
  "replication_factor": 1,
  "consistency_level": "LOCAL_QUORUM",
  "ttl": 2592000,
  "table_schema": {
    "columns": {
```

```
      "transaction_id": {"type": "uuid", "primary_key": true},
      "customer_id": {"type": "text"},
      "product_id": {"type": "text"},
      "transaction_date": {"type": "timestamp"},
      "amount": {"type": "decimal"},
      "quantity": {"type": "int"},
      "city": {"type": "text"},
      "country": {"type": "text"},
      "category": {"type": "text"},
      "product_price": {"type": "decimal"}
    },
    "clustering_order": {
      "transaction_date": "DESC"
    }
  }
}
```

Intégration avec Spark pour le chargement en masse

Pour les chargements de données volumineux, Spark offre des connecteurs optimisés pour HDFS et Cassandra.

Chargeur Spark pour HDFS et Cassandra

```
 # src/loaders/spark_bulk_loader.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Chargeur en masse avec Spark pour HDFS et Cassandra.

Ce script permet de charger des données en masse depuis diverses
sources
vers HDFS et/ou Cassandra en utilisant Spark.
"""

import os
import argparse
import logging
import json
from datetime import datetime
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit

# Configuration du logging
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %
```

```
(message)s"
)
logger = logging.getLogger(__name__)

class SparkBulkLoader:
    """Chargeur en masse avec Spark pour HDFS et Cassandra."""

    def __init__(self, config_path):
        """
        Initialise le chargeur avec la configuration spécifiée.

        Args:
            config_path (str): Chemin vers le fichier de
configuration JSON
        """
        self.config = self._load_config(config_path)

        # Initialisation de Spark avec les dépendances Cassandra
        self.spark = SparkSession.builder \
            .appName("SparkBulkLoader") \
            .master(self.config.get("spark_master", "local[*]"))
\
            .config("spark.jars.packages",
"com.datastax.spark:spark-cassandra-connector_2.12:3.3.0") \
            .config("spark.sql.extensions",
"com.datastax.spark.connector.CassandraSparkExtensions") \
            .config("spark.cassandra.connection.host",
self.config.get("cassandra_host", "cassandra")) \
            .config("spark.cassandra.connection.port",
self.config.get("cassandra_port", "9042")) \
            .getOrCreate()


logger.info(f"Chargeur initialisé avec la configuration:
{self.config}")

    def _load_config(self, config_path):
        """
        Charge la configuration depuis un fichier JSON.

        Args:
            config_path (str): Chemin vers le fichier de
configuration

        Returns:
            dict: Configuration chargée
        """
        try:
            with open(config_path, 'r') as f:
                config = json.load(f)

            # Validation des paramètres requis
```

```
            required_params = ['input_format', 'input_path']
            for param in required_params:
                if param not in config:
                    raise
ValueError(f"Paramètre requis manquant dans la configuration:
{param}")

            return config
        except Exception as e:
            logger.error(f"Erreur lors du chargement de la
configuration: {e}")
            raise

    def load(self, date=None):
        """
        Charge les données selon la configuration.

        Args:
            date (str, optional): Date des données au format
YYYY-MM-DD
                                  Si non spécifiée, utilise la
date du jour

        Returns:
            dict: Résultats du chargement
        """
        # Déterminer la date à utiliser
        if date:
            try:
                process_date = datetime.strptime(date, '%Y-%m-
%d')
            except ValueError:
                logger.error(f"Format de date invalide: {date}.
Utilisation de la date du jour.")
                process_date = datetime.now()
        else:
            process_date = datetime.now()

        date_str = process_date.strftime('%Y-%m-%d')

        # Lire les données d'entrée
        input_format = self.config['input_format']
        input_path = self.config['input_path']

        logger.info(f"Lecture des données au format
{input_format} depuis {input_path}")

        if input_format == 'parquet':
            df = self.spark.read.parquet(input_path)
        elif input_format == 'csv':
            df = self.spark.read.csv(
                input_path,
```

```
                header=self.config.get('csv_header', True),
                inferSchema=self.config.get('csv_infer_schema',
True),
                sep=self.config.get('csv_delimiter', ',')
            )
        elif input_format == 'json':
            df = self.spark.read.json(input_path)
        elif input_format == 'jdbc':
            df = self.spark.read.format('jdbc') \
                .option('url', self.config['jdbc_url']) \
                .option('dbtable', self.config['jdbc_table']) \
                .option('user', self.config['jdbc_user']) \
                .option('password',
self.config['jdbc_password']) \
                .load()
        else:
            raise ValueError(f"Format d'entrée non supporté:
{input_format}")

        # Appliquer les transformations configurées
        if 'transformations' in self.config:
            df = self._apply_transformations(df)

        # Ajouter la date de traitement si demandé
        if self.config.get('add_processing_date', False):
            df = df.withColumn('processing_date', lit(date_str))

        # Résultats du chargement
        results = {}

        # Charger dans HDFS si configuré
        if 'hdfs_output' in self.config:
            hdfs_path = self._load_to_hdfs(df, date_str)
            results['hdfs_path'] = hdfs_path

        # Charger dans Cassandra si configuré
        if 'cassandra_output' in self.config:
            cassandra_count = self._load_to_cassandra(df)
            results['cassandra_count'] = cassandra_count

        return results

    def _apply_transformations(self, df):
        """
        Applique les transformations configurées au DataFrame.

        Args:
            df (DataFrame): DataFrame Spark à transformer

        Returns:
            DataFrame: DataFrame transformé
        """
```

```
        transformations = self.config['transformations']

        # Appliquer les filtres
        if 'filters' in transformations:
            for filter_expr in transformations['filters']:
                df = df.filter(filter_expr)
                logger.info(f"Transformation: Application du
filtre {filter_expr}")

        # Sélectionner les colonnes
        if 'select_columns' in transformations:
            df = df.select(*transformations['select_columns'])

logger.info(f"Transformation: Sélection des colonnes
{transformations['select_columns']}")

        # Renommer les colonnes
        if 'rename_columns' in transformations:
            for old_name, new_name in
transformations['rename_columns'].items():
                df = df.withColumnRenamed(old_name, new_name)
                logger.info(f"Transformation: Renommage de la
colonne {old_name} en {new_name}")

        return df

    def _load_to_hdfs(self, df, date_str):
        """
        Charge les données dans HDFS.

        Args:
            df (DataFrame): DataFrame Spark à charger
            date_str (str): Date au format YYYY-MM-DD

        Returns:
            str: Chemin HDFS où les données ont été chargées
        """
        hdfs_config = self.config['hdfs_output']
        hdfs_base_path = hdfs_config['path']

        # Construire le chemin avec partitionnement par date si
configuré
        if hdfs_config.get('use_date_partitioning', True):
            year, month, day = date_str.split('-')
            hdfs_path = os.path.join(hdfs_base_path,
f"year={year}/month={month}/day={day}")
        else:
            hdfs_path = hdfs_base_path

        # Configurer les options d'écriture
        write_options = {}
```

```
        # Compression
        compression = hdfs_config.get('compression', 'snappy')
        write_options['compression'] = compression

        # Mode d'écriture
        write_mode = hdfs_config.get('write_mode', 'overwrite')

        # Écrire les données
        logger.info(f"Écriture des données dans HDFS:
{hdfs_path}")
        writer =
df.write.mode(write_mode).options(**write_options)

        # Partitionnement
        if 'partition_columns' in hdfs_config:
            partition_columns = hdfs_config['partition_columns']
            logger.info(f"Partitionnement par colonnes:
{partition_columns}")
            writer.partitionBy(*partition_columns)

        # Format de sortie
        output_format = hdfs_config.get('format', 'parquet')
        if output_format == 'parquet':
            writer.parquet(hdfs_path)
        elif output_format == 'orc':
            writer.orc(hdfs_path)
        elif output_format == 'csv':
            writer.csv(hdfs_path,
header=hdfs_config.get('csv_header', True))
        elif output_format == 'json':
            writer.json(hdfs_path)
        else:
            raise ValueError(f"Format de sortie HDFS non
supporté: {output_format}")

        logger.info(f"Données chargées avec succès dans HDFS:
{hdfs_path}")
        return hdfs_path

    def _load_to_cassandra(self, df):
        """
        Charge les données dans Cassandra.

        Args:
            df (DataFrame): DataFrame Spark à charger

        Returns:
            int: Nombre de lignes chargées
        """
        cassandra_config = self.config['cassandra_output']
        keyspace = cassandra_config['keyspace']
        table = cassandra_config['table']
```

```
        # Mode d'écriture
        write_mode = cassandra_config.get('write_mode',
'append')

        # TTL (Time To Live)
        ttl = cassandra_config.get('ttl')
        ttl_option = f"ttl {ttl}" if ttl else ""

        logger.info(f"Écriture des données dans Cassandra:
{keyspace}.{table}")

        # Écrire dans Cassandra
        try:
            df.write \
                .format("org.apache.spark.sql.cassandra") \
                .options(table=table, keyspace=keyspace) \
                .mode(write_mode) \
                .option("confirm.truncate", "true") \
                .option("ttl", ttl_option) \
                .save()

            # Compter les lignes chargées
            count = df.count()
            logger.info(f"Données chargées avec succès dans
Cassandra: {count} lignes")
            return count
        except Exception as e:
            logger.error(f"Erreur lors du chargement dans
Cassandra: {e}")
            raise

    def close(self):
        """Ferme les connexions et libère les ressources."""
        if self.spark:
            self.spark.stop()
            logger.info("Session Spark arrêtée")

def main():
    """Point d'entrée principal du script."""
    parser = argparse.ArgumentParser(description='Chargeur en
masse avec Spark')
    parser.add_argument('--config', required=True, help='Chemin
vers le fichier de configuration')
    parser.add_argument('--date', help='Date des données (YYYY-
MM-DD)')

    args = parser.parse_args()

    try:
        loader = SparkBulkLoader(args.config)
        results = loader.load(args.date)
```

```
        loader.close()


logger.info(f"Chargement terminé avec succès. Résultats:
{results}")
        return 0
    except Exception as e:
        logger.error(f"Erreur lors du chargement: {e}")
        return 1

if __name__ == '__main__':
    exit(main())
```

Configuration du chargeur Spark config\transaction_transformer_config.json

```
{
  "spark_master": "spark://spark-master:7077",
  "input_format": "parquet",
  "input_path": "/path/to/input/data",
  "add_processing_date": true,
  "transformations": {
    "filters": ["amount > 0"],
    "select_columns": ["transaction_id", "customer_id",
"product_id", "transaction_date", "amount", "quantity",
"category"],
    "rename_columns": {
      "transaction_date": "date"
    }
  },
  "hdfs_output": {
    "path": "/data/processed/transactions",
    "format": "parquet",
    "compression": "snappy",
    "write_mode": "overwrite",
    "use_date_partitioning": true,
    "partition_columns": ["category"]
  },
  "cassandra_output": {
    "keyspace": "pipeline",
    "table": "transactions_enriched",
    "write_mode": "append",
    "ttl": 2592000
  },
  "cassandra_host": "cassandra",
  "cassandra_port": "9042"
}
```

Stratégies de chargement pour les données temps réel

Pour les données temps réel, plusieurs stratégies de chargement peuvent être utilisées :

1. Chargement direct depuis Flink vers Cassandra

Flink peut écrire directement dans Cassandra en utilisant le connecteur Cassandra :

```
// Exemple en Java pour Flink
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

// Définir la source Kafka
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "kafka:9092");
properties.setProperty("group.id", "flink-cassandra-loader");

FlinkKafkaConsumer<String> kafkaSource = new
FlinkKafkaConsumer<>(
    "transactions",
    new SimpleStringSchema(),
    properties
);

// Lire les données de Kafka
DataStream<String> stream = env.addSource(kafkaSource);

// Transformer les données JSON en objets Transaction
DataStream<Transaction> transactions = stream.map(new
MapFunction<String, Transaction>() {
    @Override
    public Transaction map(String value) throws Exception {
        // Convertir JSON en objet Transaction
        ObjectMapper mapper = new ObjectMapper();
        return mapper.readValue(value, Transaction.class);
    }
});

// Écrire dans Cassandra
CassandraSink.addSink(transactions)
    .setHost("cassandra")
    .setQuery("INSERT INTO
pipeline.transactions_realtime(transaction_id, customer_id,
product_id, transaction_date, amount, quantity) VALUES
(?, ?, ?, ?, ?, ?)")
    .build();

env.execute("Flink to Cassandra Loader");
```

2. Utilisation de Kafka Connect pour Cassandra

Kafka Connect offre un connecteur Cassandra qui peut être utilisé pour charger automatiquement les données des topics Kafka vers Cassandra :

```
{
  "name": "cassandra-sink",
  "config": {
    "connector.class":
"com.datastax.oss.kafka.sink.CassandraSinkConnector",
    "tasks.max": "1",
    "topics": "transactions",
    "contactPoints": "cassandra",
    "loadBalancing.localDc": "datacenter1",
    "port": "9042",
    "auth.username": "",
    "auth.password": "",

"topic.transactions.pipeline.transactions_realtime.mapping":
"transaction_id=value.transaction_id,
customer_id=value.customer_id, product_id=value.product_id,
transaction_date=value.transaction_date, amount=value.amount,
quantity=value.quantity",

"topic.transactions.pipeline.transactions_realtime.consistencyLevel":
"LOCAL_QUORUM"
  }
}
```

Bonnes pratiques pour les chargeurs de données

1. **Performance** :
2. Utiliser le format Parquet pour HDFS (compression, schéma, performances de lecture).
3. Optimiser les écritures Cassandra (batch, niveau de cohérence approprié).
4. Partitionner les données HDFS par date ou autres dimensions pertinentes.

5. Configurer correctement le modèle de données Cassandra pour éviter les anti-patterns.

6. **Fiabilité** :

7. Implémenter des mécanismes de reprise après échec.
8. Journaliser les opérations de chargement pour audit et débogage.

9. Valider les données avant chargement.

10. **Monitoring** :
11. Collecter des métriques sur les opérations de chargement (temps, volume, erreurs).
12. Configurer des alertes en cas d'échec ou de dégradation des performances.
13. **Gestion des erreurs** :
14. Implémenter une stratégie de gestion des données invalides (fichiers de rejet, tables d'erreurs).
15. Prévoir des mécanismes de reprise partielle après échec.

Conclusion

L'implémentation des chargeurs pour Hadoop HDFS et Cassandra permet de stocker efficacement les données transformées, qu'elles proviennent de traitements batch ou temps réel. HDFS offre un stockage économique et performant pour les grands volumes de données historiques, tandis que Cassandra assure un accès rapide aux données récentes et aux résultats d'agrégation.

En combinant ces deux systèmes de stockage et en utilisant les bonnes pratiques de chargement, nous pouvons construire une couche de service robuste qui répond aux besoins variés des utilisateurs finaux, tout en maintenant des performances optimales et une haute disponibilité.

La prochaine étape consistera à orchestrer l'ensemble du pipeline avec Apache Airflow, en automatisant l'exécution des différentes étapes (extraction, transformation, chargement) selon des plannings définis.

Correction Partie 5 : Orchestration avec Airflow dans un environnement conteneurisé

L'orchestration est une étape cruciale dans la mise en place d'un pipeline de données Big Data. Elle permet d'automatiser l'exécution des différentes étapes du pipeline, de gérer les dépendances entre les tâches, et de surveiller l'ensemble du processus. Dans cette partie, nous allons implémenter l'orchestration de notre pipeline avec Apache Airflow dans un environnement conteneurisé avec Docker Compose.

Apache Airflow dans un environnement Docker Compose

Apache Airflow est un outil d'orchestration open-source qui permet de programmer, planifier et surveiller des workflows complexes. Dans notre architecture Big Data, Airflow sera responsable de l'orchestration des jobs Spark pour le traitement batch, du déploiement des applications Flink pour le traitement temps réel, et de la coordination des différentes étapes d'extraction, transformation et chargement.

Configuration d'Airflow dans Docker Compose

Nous avons déjà configuré Airflow dans notre fichier Docker Compose. Rappelons les principaux éléments :

```
   airflow-webserver:
     image: apache/airflow:2.5.1
     container_name: airflow-webserver
     depends_on:
       - postgres
     environment:
       - AIRFLOW__CORE__EXECUTOR=LocalExecutor
       - AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgresql+psycopg2://
airflow:airflow@postgres/airflow
       -
AIRFLOW__CORE__FERNET_KEY=46BKJoQYlPPOexq0OhDZnIlNepKFf87WFwLbfzqDDho=
       - AIRFLOW__CORE__LOAD_EXAMPLES=False
       -
AIRFLOW__API__AUTH_BACKENDS=airflow.api.auth.backend.basic_auth
     volumes:
       - ./dags:/opt/airflow/dags
       - ./logs:/opt/airflow/logs
       - ./plugins:/opt/airflow/plugins
       - ./config/airflow/airflow.cfg:/opt/airflow/airflow.cfg
     ports:
       - "8082:8080"
     command: webserver
     healthcheck:
       test: ["CMD", "curl", "--fail", "http://localhost:8080/
health"]
       interval: 30s
       timeout: 10s
       retries: 5
     networks:
       - data_pipeline_network

   airflow-scheduler:
     image: apache/airflow:2.5.1
```

```
    container_name: airflow-scheduler
    depends_on:
      - postgres
    environment:
      - AIRFLOW__CORE__EXECUTOR=LocalExecutor
      - AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgresql+psycopg2://
airflow:airflow@postgres/airflow
      -
AIRFLOW__CORE__FERNET_KEY=46BKJoQYlPPOexq0OhDZnIlNepKFf87WFwLbfzqDDho=
      - AIRFLOW__CORE__LOAD_EXAMPLES=False
    volumes:
      - ./dags:/opt/airflow/dags
      - ./logs:/opt/airflow/logs
      - ./plugins:/opt/airflow/plugins
      - ./config/airflow/airflow.cfg:/opt/airflow/airflow.cfg
    command: scheduler
    networks:
      - data_pipeline_network
```

Structure des DAGs Airflow

Dans Airflow, les workflows sont définis sous forme de DAGs (Directed Acyclic Graphs). Nous allons créer plusieurs DAGs pour orchestrer notre pipeline :

1. **DAG d'extraction** : Pour extraire les données des différentes sources
2. **DAG de transformation batch** : Pour exécuter les jobs Spark de transformation
3. **DAG de chargement** : Pour charger les données transformées dans Cassandra
4. **DAG de déploiement des applications temps réel** : Pour déployer et gérer les applications Flink

Implémentation des DAGs Airflow

DAG d'extraction des données

```
# dags/extraction_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.providers.apache.hdfs.hooks.hdfs import HDFSHook
from airflow.providers.apache.kafka.hooks.kafka import KafkaHook
import os
import json

# Définition des arguments par défaut
default_args = {
```

```
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Création du DAG
dag = DAG(
    'data_extraction',
    default_args=default_args,
    description='Extraction des données depuis différentes
sources',
    schedule_interval='0 1 * * *',  # Tous les jours à 1h du
matin
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['extraction', 'pipeline'],
)

# Définition des chemins des extracteurs et des configurations
extractors_base_path = '/opt/airflow/dags/extractors'
config_base_path = '/opt/airflow/dags/config'

# Tâche 1: Extraction CSV vers HDFS
extract_csv_to_hdfs = BashOperator(
    task_id='extract_csv_to_hdfs',
    bash_command=f'python {extractors_base_path}/
csv_to_hdfs_extractor.py '
                 f'--config {config_base_path}/
csv_extractor_config.json '
                 f'--input /data/raw/csv '
                 f'--date {{ ds }}',
    dag=dag,
)

# Tâche 2: Extraction API vers Kafka
extract_api_to_kafka = BashOperator(
    task_id='extract_api_to_kafka',
    bash_command=f'python {extractors_base_path}/
api_to_kafka_extractor.py '
                 f'--config {config_base_path}/
api_extractor_config.json '
                 f'--endpoint products '
                 f'--params \'{{"date": "{{ ds }}"}}\'',
    dag=dag,
)

# Tâche 3: Extraction DB vers Kafka et HDFS
extract_db_to_kafka_hdfs = BashOperator(
    task_id='extract_db_to_kafka_hdfs',
```

```
    bash_command=f'python {extractors_base_path}/
db_to_kafka_hdfs_extractor.py '
                 f'--config {config_base_path}/
db_extractor_config.json '
                 f'--mode both '
                 f'--date {{ ds }}',
    dag=dag,
)

# Fonction pour vérifier les données extraites dans HDFS
def check_hdfs_extraction(**kwargs):
    hdfs_hook = HDFSHook(hdfs_conn_id='hdfs_default')
    execution_date = kwargs['ds']
    year, month, day = execution_date.split('-')

    # Vérifier les chemins HDFS
    paths_to_check = [
        f'/data/raw/transactions/year={year}/month={month}/
day={day}',
        f'/data/raw/customers/year={year}/month={month}/
day={day}'
    ]

    for path in paths_to_check:
        if not hdfs_hook.check_for_path(path):
            raise ValueError(f"Chemin HDFS non trouvé: {path}")

    return True

# Tâche 4: Vérification des données extraites dans HDFS
check_hdfs_data = PythonOperator(
    task_id='check_hdfs_data',
    python_callable=check_hdfs_extraction,
    provide_context=True,
    dag=dag,
)

# Fonction pour vérifier les données extraites dans Kafka
def check_kafka_extraction(**kwargs):
    kafka_hook = KafkaHook(kafka_conn_id='kafka_default')

    # Vérifier les topics Kafka
    topics_to_check = ['transactions', 'products', 'customers']

    for topic in topics_to_check:
        consumer = kafka_hook.get_consumer(topic=topic,
group_id='airflow-checker')
        # Vérifier s'il y a des messages dans le topic
        messages = consumer.poll(timeout_ms=5000, max_records=1)
        if not messages:
            raise ValueError(f"Aucun message trouvé dans le
topic Kafka: {topic}")
```

```
    return True

# Tâche 5: Vérification des données extraites dans Kafka
check_kafka_data = PythonOperator(
    task_id='check_kafka_data',
    python_callable=check_kafka_extraction,
    provide_context=True,
    dag=dag,
)

# Définition des dépendances entre les tâches
extract_csv_to_hdfs >> check_hdfs_data
extract_api_to_kafka >> check_kafka_data
extract_db_to_kafka_hdfs >> [check_hdfs_data, check_kafka_data]
```

DAG de transformation batch avec Spark

```
# dags/batch_transformation_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.apache.spark.operators.spark_submit
import SparkSubmitOperator
from airflow.operators.python import PythonOperator
from airflow.providers.apache.hdfs.hooks.hdfs import HDFSHook
import os

# Définition des arguments par défaut
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Création du DAG
dag = DAG(
    'batch_transformation',
    default_args=default_args,
    description='Transformation batch des données avec Spark',
    schedule_interval='0 3 * * *',  # Tous les jours à 3h du
matin
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['transformation', 'batch', 'spark', 'pipeline'],
)

# Définition des chemins des transformateurs et des
```

```
configurations
transformers_base_path = '/opt/airflow/dags/transformers/batch'
config_base_path = '/opt/airflow/dags/config'

# Tâche 1: Transformation des transactions
transform_transactions = SparkSubmitOperator(
    task_id='transform_transactions',
    application=f'{transformers_base_path}/
transaction_transformer.py',
    conn_id='spark_default',
    application_args=[
        '--config', f'{config_base_path}/
transaction_transformer_config.json',
        '--date', '{{ ds }}'
    ],
    conf={
        'spark.driver.memory': '2g',
        'spark.executor.memory': '2g',
        'spark.executor.cores': '2',
        'spark.jars.packages': 'com.datastax.spark:spark-
cassandra-connector_2.12:3.3.0'
    },
    dag=dag,
)

# Tâche 2: Transformation des clients
transform_customers = SparkSubmitOperator(
    task_id='transform_customers',
    application=f'{transformers_base_path}/
customer_transformer.py',
    conn_id='spark_default',
    application_args=[
        '--config', f'{config_base_path}/
customer_transformer_config.json',
        '--date', '{{ ds }}'
    ],
    conf={
        'spark.driver.memory': '2g',
        'spark.executor.memory': '2g',
        'spark.executor.cores': '2',
        'spark.jars.packages': 'com.datastax.spark:spark-
cassandra-connector_2.12:3.3.0'
    },
    dag=dag,
)

# Tâche 3: Transformation des produits
transform_products = SparkSubmitOperator(
    task_id='transform_products',
    application=f'{transformers_base_path}/
product_transformer.py',
    conn_id='spark_default',
```

```
    application_args=[
        '--config', f'{config_base_path}/
product_transformer_config.json',
        '--date', '{{ ds }}'
    ],
    conf={
        'spark.driver.memory': '2g',
        'spark.executor.memory': '2g',
        'spark.executor.cores': '2',
        'spark.jars.packages': 'com.datastax.spark:spark-
cassandra-connector_2.12:3.3.0'
    },
    dag=dag,
)

# Tâche 4: Calcul des KPIs (dépend des transformations
précédentes)
calculate_kpis = SparkSubmitOperator(
    task_id='calculate_kpis',
    application=f'{transformers_base_path}/kpi_calculator.py',
    conn_id='spark_default',
    application_args=[
        '--config', f'{config_base_path}/
kpi_calculator_config.json',
        '--date', '{{ ds }}'
    ],
    conf={
        'spark.driver.memory': '2g',
        'spark.executor.memory': '2g',
        'spark.executor.cores': '2',
        'spark.jars.packages': 'com.datastax.spark:spark-
cassandra-connector_2.12:3.3.0'
    },
    dag=dag,
)

# Fonction pour vérifier les données transformées dans HDFS
def check_transformed_data(**kwargs):
    hdfs_hook = HDFSHook(hdfs_conn_id='hdfs_default')
    execution_date = kwargs['ds']
    year, month, day = execution_date.split('-')

    # Vérifier les chemins HDFS
    paths_to_check = [
        f'/data/processed/transactions/year={year}/
month={month}/day={day}',
        f'/data/processed/customers/year={year}/month={month}/
day={day}',
        f'/data/processed/products/year={year}/month={month}/
day={day}',
        f'/data/processed/kpis/year={year}/month={month}/
day={day}'
```

```
    ]

    for path in paths_to_check:
        if not hdfs_hook.check_for_path(path):
            raise ValueError(f"Chemin HDFS non trouvé: {path}")

    return True

# Tâche 5: Vérification des données transformées
check_transformed = PythonOperator(
    task_id='check_transformed_data',
    python_callable=check_transformed_data,
    provide_context=True,
    dag=dag,
)

# Définition des dépendances entre les tâches
[transform_transactions, transform_customers,
transform_products] >> calculate_kpis >> check_transformed
```

DAG de chargement des données

```
# dags/data_loading_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.providers.apache.cassandra.hooks.cassandra import
CassandraHook
import os

# Définition des arguments par défaut
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Création du DAG
dag = DAG(
    'data_loading',
    default_args=default_args,
    description='Chargement des données transformées',
    schedule_interval='0 5 * * *',  # Tous les jours à 5h du
matin
    start_date=datetime(2023, 1, 1),
    catchup=False,
```

```
    tags=['loading', 'pipeline'],
)

# Définition des chemins des chargeurs et des configurations
loaders_base_path = '/opt/airflow/dags/loaders'
config_base_path = '/opt/airflow/dags/config'

# Tâche 1: Chargement des transactions dans Cassandra
load_transactions = BashOperator(
    task_id='load_transactions_to_cassandra',
    bash_command=f'python {loaders_base_path}/
cassandra_loader.py '
                 f'--config {config_base_path}/
transactions_cassandra_loader_config.json '
                 f'--input /data/processed/transactions/year={{
execution_date.year }}/month={{ execution_date.month }}/day={{
execution_date.day }} '
                 f'--format parquet',
    dag=dag,
)

# Tâche 2: Chargement des clients dans Cassandra
load_customers = BashOperator(
    task_id='load_customers_to_cassandra',
    bash_command=f'python {loaders_base_path}/
cassandra_loader.py '
                 f'--config {config_base_path}/
customers_cassandra_loader_config.json '
                 f'--input /data/processed/customers/year={{
execution_date.year }}/month={{ execution_date.month }}/day={{
execution_date.day }} '
                 f'--format parquet',
    dag=dag,
)

# Tâche 3: Chargement des produits dans Cassandra
load_products = BashOperator(
    task_id='load_products_to_cassandra',
    bash_command=f'python {loaders_base_path}/
cassandra_loader.py '
                 f'--config {config_base_path}/
products_cassandra_loader_config.json '
                 f'--input /data/processed/products/year={{
execution_date.year }}/month={{ execution_date.month }}/day={{
execution_date.day }} '
                 f'--format parquet',
    dag=dag,
)

# Tâche 4: Chargement des KPIs dans Cassandra
load_kpis = BashOperator(
    task_id='load_kpis_to_cassandra',
```

```
    bash_command=f'python {loaders_base_path}/
cassandra_loader.py '
                 f'--config {config_base_path}/
kpis_cassandra_loader_config.json '
                 f'--input /data/processed/kpis/year={{
execution_date.year }}/month={{ execution_date.month }}/day={{
execution_date.day }} '
                 f'--format parquet',
    dag=dag,
)

# Fonction pour vérifier les données chargées dans Cassandra
def check_cassandra_data(**kwargs):
    cassandra_hook =
CassandraHook(cassandra_conn_id='cassandra_default')
    session = cassandra_hook.get_conn()

    # Vérifier les tables Cassandra
    tables_to_check = [
        'transactions_enriched',
        'customers_enriched',
        'products_enriched',
        'customer_kpis'
    ]

    execution_date = kwargs['ds']

    for table in tables_to_check:
        query = f"SELECT COUNT(*) FROM pipeline.{table} WHERE
processing_date = '{execution_date}'"
        result = session.execute(query).one()
        count = result[0]

        if count == 0:
            raise ValueError(f"Aucune donnée trouvée dans la
table Cassandra: pipeline.{table} pour la date
{execution_date}")

    return True

# Tâche 5: Vérification des données chargées dans Cassandra
check_cassandra = PythonOperator(
    task_id='check_cassandra_data',
    python_callable=check_cassandra_data,
    provide_context=True,
    dag=dag,
)

# Définition des dépendances entre les tâches
[load_transactions, load_customers, load_products, load_kpis] >>
check_cassandra
```

DAG de déploiement des applications temps réel

```
# dags/realtime_deployment_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.sensors.external_task import ExternalTaskSensor
import requests
import json
import time

# Définition des arguments par défaut
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

# Création du DAG
dag = DAG(
    'realtime_deployment',
    default_args=default_args,
    description='Déploiement des applications temps réel avec
Flink',
    schedule_interval='0 2 * * *',  # Tous les jours à 2h du
matin
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['deployment', 'realtime', 'flink', 'pipeline'],
)

# Attendre que l'extraction des données soit terminée
wait_for_extraction = ExternalTaskSensor(
    task_id='wait_for_extraction',
    external_dag_id='data_extraction',
    external_task_id=None,  # Attendre que tout le DAG soit
terminé
    timeout=3600,
    mode='reschedule',
    dag=dag,
)

# Tâche 1: Déployer l'application Flink de calcul des KPIs en
temps réel
deploy_realtime_kpi_calculator = BashOperator(
    task_id='deploy_realtime_kpi_calculator',
    bash_command='flink run -d -m jobmanager:8081 '
```

```
                '/opt/airflow/dags/transformers/streaming/
realtime_kpi_calculator.py '
                '--config /opt/airflow/dags/config/
flink_kpi_config.json',
    dag=dag,
)

# Tâche 2: Déployer l'application Flink de détection de fraude
en temps réel
deploy_fraud_detector = BashOperator(
    task_id='deploy_fraud_detector',
    bash_command='flink run -d -m jobmanager:8081 '
                '/opt/airflow/dags/transformers/streaming/
fraud_detector.py '
                '--config /opt/airflow/dags/config/
flink_fraud_config.json',
    dag=dag,
)

# Fonction pour vérifier l'état des jobs Flink
def check_flink_jobs(**kwargs):
    # URL de l'API REST de Flink
    flink_api_url = 'http://jobmanager:8081/jobs/overview'

    # Attendre un peu pour que les jobs démarrent
    time.sleep(10)

    # Récupérer la liste des jobs
    response = requests.get(flink_api_url)
    jobs = response.json()['jobs']

    # Vérifier que les jobs sont en cours d'exécution
    running_jobs = [job for job in jobs if job['status'] ==
'RUNNING']

    if len(running_jobs) < 2:
        raise ValueError(f"Certains jobs Flink ne sont pas en
cours d'exécution. Jobs en cours: {len(running_jobs)}")

    return True

# Tâche 3: Vérifier l'état des jobs Flink
check_flink = PythonOperator(
    task_id='check_flink_jobs',
    python_callable=check_flink_jobs,
    provide_context=True,
    dag=dag,
)

# Définition des dépendances entre les tâches
wait_for_extraction >> [deploy_realtime_kpi_calculator,
deploy_fraud_detector] >> check_flink
```

DAG principal pour orchestrer l'ensemble du pipeline

```
# dags/master_pipeline_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.operators.python import PythonOperator
from airflow.sensors.external_task import ExternalTaskSensor
import requests
import json

# Définition des arguments par défaut
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'email': ['data-team@example.com'],
}

# Création du DAG
dag = DAG(
    'master_pipeline',
    default_args=default_args,
    description='Orchestration complète du pipeline de données',
    schedule_interval='0 0 * * *',  # Tous les jours à minuit
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['master', 'pipeline'],
)

# Tâche de début
start = DummyOperator(
    task_id='start_pipeline',
    dag=dag,
)

# Attendre que l'extraction des données soit terminée
wait_for_extraction = ExternalTaskSensor(
    task_id='wait_for_extraction',
    external_dag_id='data_extraction',
    external_task_id=None,
    timeout=7200,
    mode='reschedule',
    dag=dag,
)

# Attendre que la transformation batch soit terminée
wait_for_batch_transformation = ExternalTaskSensor(
```

```
    task_id='wait_for_batch_transformation',
    external_dag_id='batch_transformation',
    external_task_id=None,
    timeout=7200,
    mode='reschedule',
    dag=dag,
)

# Attendre que le chargement des données soit terminé
wait_for_loading = ExternalTaskSensor(
    task_id='wait_for_loading',
    external_dag_id='data_loading',
    external_task_id=None,
    timeout=7200,
    mode='reschedule',
    dag=dag,
)

# Attendre que le déploiement des applications temps réel soit
terminé
wait_for_realtime = ExternalTaskSensor(
    task_id='wait_for_realtime_deployment',
    external_dag_id='realtime_deployment',
    external_task_id=None,
    timeout=7200,
    mode='reschedule',
    dag=dag,
)

# Fonction pour envoyer une notification de fin de pipeline
def send_pipeline_completion_notification(**kwargs):
    execution_date = kwargs['ds']

    # Ici, vous pourriez envoyer un email, un message Slack,
etc.
    print(f"Pipeline terminé avec succès pour la date
{execution_date}")

    # Simuler l'envoi d'une notification
    notification = {
        'status': 'success',
        'execution_date': execution_date,
        'message': f"Pipeline de données terminé avec succès
pour la date {execution_date}"
    }

    return notification

# Tâche de notification de fin de pipeline
notify_completion = PythonOperator(
    task_id='notify_completion',
    python_callable=send_pipeline_completion_notification,
```

```
    provide_context=True,
    dag=dag,
)

# Tâche de fin
end = DummyOperator(
    task_id='end_pipeline',
    dag=dag,
)

# Définition des dépendances entre les tâches
start >> wait_for_extraction >> wait_for_batch_transformation >>
wait_for_loading
wait_for_extraction >> wait_for_realtime
[wait_for_loading, wait_for_realtime] >> notify_completion >>
end
```

Plugins Airflow personnalisés

Pour étendre les fonctionnalités d'Airflow et faciliter l'intégration avec notre stack Big Data, nous pouvons créer des plugins personnalisés.

Plugin pour les opérateurs Flink

```
# plugins/flink_plugin.py
from airflow.plugins_manager import AirflowPlugin
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults
import requests
import json
import time

class FlinkSubmitOperator(BaseOperator):
    """
    Opérateur pour soumettre un job Flink.
    """

    @apply_defaults
    def __init__(
        self,
        job_jar_path,
        flink_master_url='http://jobmanager:8081',
        entry_class=None,
        program_args=None,
        parallelism=1,
        allow_non_restored_state=False,
        savepoint_path=None,
        *args, **kwargs
```

```
    ):
        super(FlinkSubmitOperator, self).__init__(*args,
**kwargs)
        self.job_jar_path = job_jar_path
        self.flink_master_url = flink_master_url
        self.entry_class = entry_class
        self.program_args = program_args
        self.parallelism = parallelism
        self.allow_non_restored_state = allow_non_restored_state
        self.savepoint_path = savepoint_path

    def execute(self, context):
        """
        Exécute la soumission du job Flink.
        """
        # Construire l'URL pour l'API REST de Flink
        submit_url = f"{self.flink_master_url}/jars/upload"

        # Télécharger le JAR
        with open(self.job_jar_path, 'rb') as jar_file:
            files = {'jarfile': jar_file}
            upload_response = requests.post(submit_url,
files=files)

        if upload_response.status_code != 200:
            raise Exception(f"Échec du téléchargement du JAR:
{upload_response.text}")

        # Récupérer l'ID du JAR
        jar_id = upload_response.json()['filename'].split('/')
[-1]

        # Construire les paramètres pour l'exécution du job
        run_params = {}

        if self.entry_class:
            run_params['entry-class'] = self.entry_class

        if self.program_args:
            run_params['program-args'] = self.program_args

        if self.parallelism:
            run_params['parallelism'] = self.parallelism

        if self.allow_non_restored_state:
            run_params['allowNonRestoredState'] = 'true'

        if self.savepoint_path:
            run_params['savepointPath'] = self.savepoint_path

        # Exécuter le job
        run_url = f"{self.flink_master_url}/jars/{jar_id}/run"
```

```
        run_response = requests.post(run_url, params=run_params)

        if run_response.status_code != 200:
            raise Exception(f"Échec de l'exécution du job:
{run_response.text}")

        job_id = run_response.json()['jobid']
        self.log.info(f"Job Flink soumis avec succès. Job ID:
{job_id}")

        return job_id

class FlinkJobSensor(BaseOperator):
    """
    Capteur pour surveiller l'état d'un job Flink.
    """

    @apply_defaults
    def __init__(
        self,
        job_id,
        flink_master_url='http://jobmanager:8081',
        target_status='FINISHED',
        poke_interval=60,
        timeout=3600,
        *args, **kwargs
    ):
        super(FlinkJobSensor, self).__init__(*args, **kwargs)
        self.job_id = job_id
        self.flink_master_url = flink_master_url
        self.target_status = target_status
        self.poke_interval = poke_interval
        self.timeout = timeout

    def execute(self, context):
        """
        Surveille l'état du job Flink.
        """
        start_time = time.time()

        while True:
            # Vérifier si le timeout est atteint
            if time.time() - start_time > self.timeout:
                raise Exception(f"Timeout atteint en attendant
que le job Flink {self.job_id} atteigne l'état
{self.target_status}")

            # Récupérer l'état du job
            job_url = f"{self.flink_master_url}/jobs/
{self.job_id}"
            response = requests.get(job_url)
```

```
            if response.status_code != 200:
                self.log.warning(f"Impossible de récupérer
l'état du job: {response.text}")
                time.sleep(self.poke_interval)
                continue

            job_status = response.json()['state']
            self.log.info(f"État actuel du job Flink
{self.job_id}: {job_status}")

            # Vérifier si l'état cible est atteint
            if job_status == self.target_status:
                self.log.info(f"Job Flink {self.job_id} a
atteint l'état {self.target_status}")
                return True

            # Vérifier si le job a échoué
            if job_status == 'FAILED' or job_status ==
'CANCELED':
                raise Exception(f"Job Flink {self.job_id} a
échoué ou a été annulé. État actuel: {job_status}")

            # Attendre avant la prochaine vérification
            time.sleep(self.poke_interval)

class FlinkPlugin(AirflowPlugin):
    name = 'flink_plugin'
    operators = [FlinkSubmitOperator, FlinkJobSensor]
```

Plugin pour les hooks Cassandra avancés

```
# plugins/cassandra_plugin.py
from airflow.plugins_manager import AirflowPlugin
from airflow.hooks.base import BaseHook
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
from cassandra.query import BatchStatement, SimpleStatement
import pandas as pd

class CassandraAdvancedHook(BaseHook):
    """
    Hook pour interagir avec Cassandra avec des fonctionnalités
avancées.
    """

    def __init__(self, cassandra_conn_id='cassandra_default'):
        super().__init__()
        self.cassandra_conn_id = cassandra_conn_id
        self.cluster = None
        self.session = None
```

```
    def get_conn(self):
        """
        Établit une connexion à Cassandra.
        """
        if self.session is not None and not
self.session.is_shutdown:
            return self.session

        conn = self.get_connection(self.cassandra_conn_id)

        # Configurer l'authentification si nécessaire
        auth_provider = None
        if conn.login and conn.password:
            auth_provider = PlainTextAuthProvider(
                username=conn.login,
                password=conn.password
            )

        # Se connecter au cluster
        self.cluster = Cluster(
            contact_points=[conn.host],
            port=conn.port or 9042,
            auth_provider=auth_provider
        )

        self.session = self.cluster.connect()

        return self.session

    def close_conn(self):
        """
        Ferme la connexion à Cassandra.
        """
        if self.cluster:
            self.cluster.shutdown()
            self.cluster = None
            self.session = None

    def execute(self, cql, parameters=None):
        """
        Exécute une requête CQL.
        """
        session = self.get_conn()

        if parameters:
            prepared = session.prepare(cql)
            return session.execute(prepared, parameters)
        else:
            return session.execute(cql)

    def execute_batch(self, statements):
```

```
        """
        Exécute un lot de requêtes CQL.
        """
        session = self.get_conn()
        batch = BatchStatement()

        for statement in statements:
            if isinstance(statement, tuple) and len(statement)
== 2:
                cql, params = statement
                prepared = session.prepare(cql)
                batch.add(prepared, params)
            else:
                batch.add(SimpleStatement(statement))

        return session.execute(batch)

    def load_dataframe(self, df, keyspace, table,
batch_size=1000):
        """
        Charge un DataFrame pandas dans une table Cassandra.
        """
        session = self.get_conn()

        # Préparer la requête d'insertion
        columns = df.columns.tolist()
        placeholders = ', '.join(['%s'] * len(columns))

        query = f"""
        INSERT INTO {keyspace}.{table}
        ({', '.join(columns)})
        VALUES ({placeholders})
        """

        # Préparer la requête
        prepared = session.prepare(query)

        # Charger par lots
        total_rows = len(df)
        loaded_rows = 0

        for i in range(0, total_rows, batch_size):
            batch = BatchStatement()
            end = min(i + batch_size, total_rows)

            for _, row in df.iloc[i:end].iterrows():
                # Convertir les valeurs NaN en None
                values = [None if pd.isna(val) else val for val
in row.values]
                batch.add(prepared, values)

            # Exécuter le batch
```

```
            session.execute(batch)
            loaded_rows += (end - i)
            self.log.info(f"Chargé {loaded_rows}/{total_rows}
lignes")

        return loaded_rows

    def get_dataframe(self, cql, parameters=None):
        """
        Exécute une requête CQL et retourne les résultats sous
forme de DataFrame pandas.
        """
        session = self.get_conn()

        if parameters:
            prepared = session.prepare(cql)
            rows = session.execute(prepared, parameters)
        else:
            rows = session.execute(cql)

        # Convertir les résultats en DataFrame
        columns = rows.column_names
        data = [[row[col] for col in columns] for row in rows]

        return pd.DataFrame(data, columns=columns)

class CassandraPlugin(AirflowPlugin):
    name = 'cassandra_plugin'
    hooks = [CassandraAdvancedHook]
```

Monitoring et alertes

Pour surveiller l'exécution de nos DAGs et être alerté en cas de problème, nous pouvons configurer des alertes dans Airflow.

Configuration des alertes par email

Modifiez le fichier `airflow.cfg` pour configurer l'envoi d'emails :

```
[email]
email_backend = airflow.utils.email.send_email_smtp
smtp_host = smtp.gmail.com
smtp_starttls = True
smtp_ssl = False
smtp_user = your-email@gmail.com
smtp_password = your-password
```

```
smtp_port = 587
smtp_mail_from = your-email@gmail.com
```

DAG de surveillance

```
# dags/monitoring_dag.py
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.http.operators.http import
SimpleHttpOperator
from airflow.providers.http.sensors.http import HttpSensor
import json
import requests

# Définition des arguments par défaut
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    'email': ['data-team@example.com'],
}

# Création du DAG
dag = DAG(
    'pipeline_monitoring',
    default_args=default_args,
    description='Surveillance du pipeline de données',
    schedule_interval='*/30 * * * *',  # Toutes les 30 minutes
    start_date=datetime(2023, 1, 1),
    catchup=False,
    tags=['monitoring', 'pipeline'],
)

# Fonction pour vérifier l'état des services
def check_services(**kwargs):
    services = [
        {'name': 'Kafka', 'url': 'http://kafka:9092', 'port':
9092},
        {'name': 'Spark Master', 'url': 'http://spark-master:
8080', 'port': 8080},
        {'name': 'Flink JobManager', 'url': 'http://jobmanager:
8081', 'port': 8081},
        {'name': 'Cassandra', 'url': 'http://cassandra:9042',
'port': 9042},
        {'name': 'HDFS NameNode', 'url': 'http://namenode:9870',
'port': 9870}
```

```
    ]

    results = {}

    for service in services:
        try:
            # Vérifier si le port est ouvert
            import socket
            sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            sock.settimeout(3)
            result =
sock.connect_ex((service['url'].split('://')[1].split(':')[0],
service['port']))
            sock.close()

            status = 'UP' if result == 0 else 'DOWN'
            results[service['name']] = status

            if status == 'DOWN':
                # Envoyer une alerte
                send_alert(f"Service {service['name']} is
DOWN!")
        except Exception as e:
            results[service['name']] = f"ERROR: {str(e)}"
            send_alert(f"Error checking service
{service['name']}: {str(e)}")

    return results

# Fonction pour envoyer une alerte
def send_alert(message):
    # Ici, vous pourriez envoyer un email, un message Slack,
etc.
    print(f"ALERT: {message}")

# Tâche pour vérifier l'état des services
check_services_task = PythonOperator(
    task_id='check_services',
    python_callable=check_services,
    provide_context=True,
    dag=dag,
)

# Fonction pour vérifier l'état des DAGs
def check_dags_status(**kwargs):
    # URL de l'API Airflow
    airflow_api_url = 'http://localhost:8082/api/v1/dags'

    try:
        # Récupérer la liste des DAGs
        response = requests.get(
```

```
            airflow_api_url,
            auth=('admin', 'admin')  # Remplacer par vos
identifiants
        )

        dags = response.json()['dags']

        # Vérifier l'état des DAGs critiques
        critical_dags = [
            'data_extraction',
            'batch_transformation',
            'data_loading',
            'realtime_deployment',
            'master_pipeline'
        ]

        results = {}

        for dag_id in critical_dags:
            # Récupérer les dernières exécutions du DAG
            dag_runs_url = f"{airflow_api_url}/{dag_id}/dagRuns"
            runs_response = requests.get(
                dag_runs_url,
                auth=('admin', 'admin')  # Remplacer par vos
identifiants
            )

            dag_runs = runs_response.json()['dag_runs']

            if dag_runs:
                latest_run = dag_runs[0]
                results[dag_id] = latest_run['state']

                # Alerter si le DAG a échoué
                if latest_run['state'] == 'failed':
                    send_alert(f"DAG {dag_id} has failed!
Execution date: {latest_run['execution_date']}")
            else:
                results[dag_id] = 'NO_RUNS'

        return results
    except Exception as e:
        send_alert(f"Error checking DAGs status: {str(e)}")
        return {'error': str(e)}

# Tâche pour vérifier l'état des DAGs
check_dags_status_task = PythonOperator(
    task_id='check_dags_status',
    python_callable=check_dags_status,
    provide_context=True,
    dag=dag,
)
```

```
# Définition des dépendances entre les tâches
check_services_task >> check_dags_status_task
```

Intégration avec Prometheus et Grafana

Pour une surveillance plus avancée, nous pouvons intégrer Airflow avec Prometheus et Grafana.

Configuration de Prometheus pour Airflow

Créez un fichier `docker/config/prometheus/prometheus.yml` :

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'airflow'
    static_configs:
      - targets: ['airflow-webserver:8080']

  - job_name: 'kafka'
    static_configs:
      - targets: ['kafka:9092']

  - job_name: 'spark'
    static_configs:
      - targets: ['spark-master:8080']

  - job_name: 'flink'
    static_configs:
      - targets: ['jobmanager:8081']

  - job_name: 'cassandra'
    static_configs:
      - targets: ['cassandra:9042']

  - job_name: 'hdfs'
    static_configs:
      - targets: ['namenode:9870']
```

Dashboard Grafana pour le pipeline

Créez un fichier `docker/config/grafana/provisioning/dashboards/pipeline_dashboard.json` avec un dashboard pour surveiller le pipeline.

Bonnes pratiques pour l'orchestration avec Airflow

1. **Structure des DAGs** :
2. Diviser les DAGs par domaine fonctionnel.
3. Utiliser un DAG maître pour orchestrer l'ensemble du pipeline.
4. Limiter le nombre de tâches par DAG pour faciliter la maintenance.
5. **Gestion des dépendances** :
6. Utiliser les sensors pour attendre la disponibilité des données.
7. Définir clairement les dépendances entre les tâches.
8. Éviter les dépendances circulaires.
9. **Gestion des erreurs** :
10. Configurer des retries appropriés pour les tâches.
11. Implémenter des mécanismes de notification en cas d'échec.
12. Prévoir des procédures de reprise après échec.
13. **Performance** :
14. Utiliser le bon exécuteur (LocalExecutor, CeleryExecutor, KubernetesExecutor).
15. Optimiser les ressources allouées aux tâches.
16. Éviter les goulots d'étranglement dans le pipeline.
17. **Sécurité** :
18. Stocker les informations sensibles dans des variables ou connections Airflow.
19. Limiter les privilèges des tâches.
20. Auditer régulièrement les accès et les exécutions.