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.