Rapport sur le projet de la Big Data & Data Engineering

Panaché de concepts Big Data & Data Engineering

Étudiante : Malak Soulhi
Filière : Génie Informatique
Établissement : ENSA Tanger

Année universitaire : 2024 – 2025

Table des matières

Présentation du projet

Introduction

L'essor massif des technologies numériques a conduit à une explosion sans précédent du volume de données générées quotidiennement. Ces données, issues notamment des réseaux sociaux, des systèmes d'information, des transactions financières et des objets connectés, présentent des caractéristiques spécifiques regroupées sous le concept des *Big Data*. Ces caractéristiques sont généralement décrites par les **5V** : *Volume*, *Vélocité*, *Variété*, *Véracité* et *Valeur*.

Dans ce contexte, l'**Internet des Objets (IoT)** joue un rôle central. Des milliers, voire des millions de capteurs sont aujourd'hui déployés pour collecter en continu des données relatives à l'environnement, à l'énergie, à la santé ou encore aux systèmes industriels. Ces capteurs génèrent des flux de données continus à haute fréquence, rendant les architectures traditionnelles de traitement de données inadaptées en raison de leurs limitations en termes de scalabilité et de latence.

Face à ces défis, les technologies Big Data, et en particulier l'écosystème Hadoop et Apache Spark, offrent des solutions robustes et distribuées permettant de traiter efficacement des volumes massifs de données en temps réel. Apache Kafka s'impose comme une plateforme de streaming distribuée capable d'assurer une ingestion fiable et scalable des données, tandis qu'Apache Spark fournit un moteur de calcul distribué performant pour le traitement en streaming et l'analyse avancée.

Ce projet s'inscrit dans le cadre du module *Big Data / Data Engineering* et vise à concevoir et implémenter une **plateforme Big Data complète**, capable d'ingérer, stocker, traiter et analyser des flux de données IoT en temps réel. L'objectif principal est de mettre en pratique les concepts fondamentaux du Big Data à travers une architecture cohérente, distribuée et réaliste, inspirée des systèmes utilisés dans des environnements industriels.

La solution proposée repose sur une chaîne de traitement complète allant de l'ingestion des données via Apache Kafka, à leur traitement en temps réel à l'aide de Spark Streaming, jusqu'à leur stockage distribué dans HDFS sous un format optimisé pour l'analyse analytique. Les données stockées sont ensuite exploitées à travers Spark SQL afin de pro-

duire des indicateurs pertinents et exploitables, pouvant être visualisés pour faciliter la prise de décision.

Ainsi, ce projet permet non seulement d'acquérir une maîtrise pratique des outils Big Data, mais également de comprendre les enjeux architecturaux et techniques liés à la conception de pipelines de données modernes et distribués.

0.1 Objectifs du Projet

L'objectif principal de ce projet est de concevoir et de mettre en œuvre une plateforme Big Data intégrée permettant le traitement de flux de données IoT en temps réel. Pour atteindre cet objectif global, plusieurs objectifs spécifiques ont été définis, tant sur le plan technique que pédagogique.

0.1.1 Scalabilité du Système

L'un des défis majeurs des systèmes IoT réside dans la capacité à gérer un grand nombre de sources de données simultanées. La plateforme développée doit être capable de supporter l'augmentation du nombre de capteurs sans dégradation significative des performances.

Pour répondre à cette exigence, le projet s'appuie sur des technologies distribuées telles qu'Apache Kafka et Apache Spark. Kafka permet de partitionner les données au sein de topics, assurant ainsi une ingestion parallèle et scalable. De son côté, Spark exploite un modèle de calcul distribué reposant sur des clusters, permettant de traiter efficacement de grands volumes de données en parallèle.

La scalabilité horizontale constitue donc un axe central de ce projet, garantissant que l'architecture proposée peut évoluer en fonction des besoins futurs.

0.1.2 Traitement en Temps Réel et Faible Latence

Un autre objectif fondamental du projet est le traitement des données en **temps réel**. Contrairement aux traitements batch traditionnels, les données IoT doivent être analysées dès leur génération afin de permettre une réaction rapide face à des événements critiques, tels que des anomalies de température ou des dysfonctionnements de capteurs.

Apache Spark Streaming est utilisé pour consommer les données depuis Kafka et effectuer des transformations en continu avec une latence réduite. Le modèle de micro-batching adopté par Spark Streaming permet de trouver un compromis optimal entre performance, fiabilité et simplicité de mise en œuvre.

Cet objectif de faible latence est essentiel pour des cas d'usage tels que la surveillance en temps réel, les alertes automatiques et les systèmes de maintenance prédictive.

0.1.3 Analytique Avancée et Calcul d'Indicateurs

Au-delà du simple traitement des flux, le projet vise à exploiter les données collectées afin d'en extraire une valeur ajoutée. Pour cela, des analyses avancées sont réalisées à partir des données stockées dans le système distribué.

Les indicateurs calculés incluent notamment :

— la température moyenne par capteur,

— le taux d'humidité maximal observé,

— la fréquence d'émission des événements par appareil,

— l'évolution temporelle des mesures.

Ces analyses sont réalisées à l'aide de Spark SQL, permettant d'exprimer des requêtes analytiques complexes sur de grands volumes de données de manière déclarative et efficace. L'utilisation de formats de stockage optimisés tels que Parquet améliore considérablement les performances des requêtes.

0.1.4 Objectifs Pédagogiques

D'un point de vue pédagogique, ce projet vise à consolider les compétences acquises durant le module Big Data / Data Engineering. Il permet notamment de :

— manipuler les RDD, DataFrames et Spark SQL,

— écrire du code Spark en Scala,

— comprendre le fonctionnement des systèmes de streaming distribués,

— concevoir une architecture Big Data cohérente de bout en bout,

— relier les concepts théoriques aux applications pratiques.

Ainsi, ce projet constitue une synthèse complète des notions abordées durant le module et prépare à des problématiques réelles rencontrées dans le domaine de l'ingénierie des données.

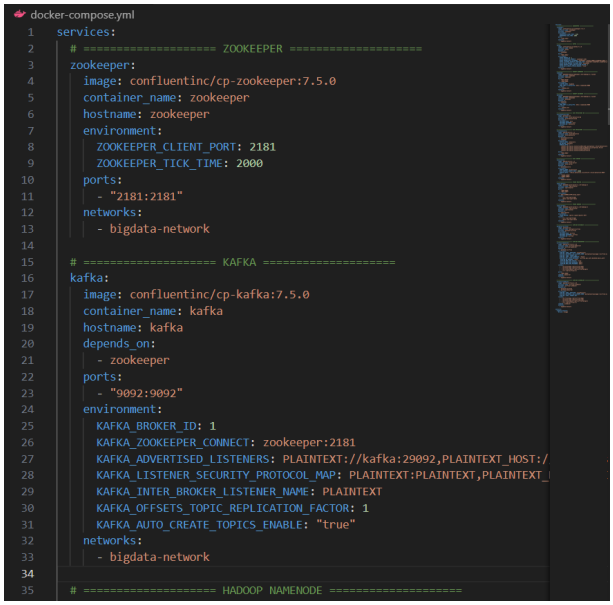
1. Architecture de la plateforme Big Data IoT

Ce chapitre présente la conception et l'implémentation d'une plateforme Big Data dédiée au traitement de flux IoT en temps réel. L'objectif principal est de mettre en place une chaîne complète permettant la génération, la transmission, le traitement et le stockage de données issues de capteurs simulés, en s'appuyant sur des technologies distribuées modernes telles que Apache Kafka, Apache Spark, HDFS et Docker.

1.1 Orchestration avec Docker Compose

Le déploiement de l'infrastructure est assuré par Docker Compose, qui permet de lancer et de coordonner l'ensemble des services distribués de manière cohérente et reproductible.

Le fichier `docker-compose.yml` définit notamment les services suivants :



— **Zookeeper** : service de coordination indispensable au fonctionnement de Kafka.

— **Kafka** : système de messagerie distribué utilisé pour la transmission des flux IoT.

— **HDFS** :

  - Namenode : gestion des métadonnées du système de fichiers distribué.
  - Datanode : stockage effectif des données.
- **Spark** :
  - Spark Master : gestion du cluster Spark.
  - Spark Worker : exécution des tâches distribuées.

L'ensemble des services communiquent à travers un réseau Docker dédié nommé `bigdata-network`, garantissant une communication fiable entre les conteneurs.

1.2 Architecture technique

L'architecture globale du système repose sur un modèle en *Pipeline*, où chaque composant joue un rôle bien défini dans le flux de données :

- **Producteur** : un script Python simulant des capteurs IoT et envoyant des messages au format JSON vers Kafka.
- **Messagerie** : Apache Kafka assure la gestion des flux de données en temps réel à travers des topics.
- **Traitement** : Apache Spark Streaming, écrit en Scala, consomme les messages Kafka, applique des transformations et enrichissements, puis prépare les données pour le stockage.
- **Stockage** : HDFS est utilisé comme système de stockage distribué, avec le format Parquet afin d'optimiser les performances analytiques.
- **Coordination** : Zookeeper est utilisé comme composant de coordination nécessaire au fonctionnement de Kafka.

Cette architecture permet un traitement des données scalable, tolérant aux pannes et adapté aux contraintes du temps réel, répondant ainsi aux exigences des systèmes Big Data modernes.

2. mise en œuvre du pipeline Big Data

2.1 Étapes de réalisation du projet

Cette section décrit de manière chronologique et détaillée les différentes étapes ayant permis la mise en place du pipeline Big Data, depuis l'ingestion des données jusqu'à leur stockage distribué. L'objectif est de présenter une implémentation concrète et fonctionnelle d'un système de traitement de données IoT en temps réel.

2.1.1 Création du topic Kafka

La première étape du projet consiste à préparer la couche d'ingestion des données en configurant Apache Kafka. Kafka joue le rôle de système de messagerie distribué permettant de gérer les flux de données en temps réel produits par les capteurs IoT.

Un *topic* Kafka nommé `iot-sensors` est créé afin de centraliser les messages envoyés par les producteurs. Ce topic est configuré avec plusieurs partitions afin de permettre un traitement parallèle des données par les consommateurs Spark.

```
C:\Users\pc\Documents\BIG DATA - MALAK SOULHI>docker exec -it kafka bash
[appuser@300bd05088bd ~]$ kafka-topics \
>   --create \
>   --topic iot-sensors \
>   --bootstrap-server kafka:9092 \
>   --partitions 3 \
>   --replication-factor 1
```

La création du topic est vérifiée à l'aide de la commande suivante :

```
[appuser@300bd05088bd ~]$ kafka-topics --list --bootstrap-server kafka:9092
iot-sensors
```

```
[appuser@300bd05088bd ~]$ kafka-topics \
>   --describe \
>   --topic iot-sensors \
>   --bootstrap-server kafka:9092
Topic: iot-sensors      TopicId: U9fPFUqSRJOCHMDcqJVV9Q PartitionCount: 3       ReplicationFactor: 1    Configs:
        Topic: iot-sensors      Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: iot-sensors      Partition: 1    Leader: 1       Replicas: 1     Isr: 1
        Topic: iot-sensors      Partition: 2    Leader: 1       Replicas: 1     Isr: 1
```

Cette étape garantit que l'infrastructure Kafka est correctement initialisée et prête à recevoir des flux de données continus.

2.1.2 Simulation des capteurs IoT avec un producteur Kafka

Afin de reproduire un environnement IoT réaliste, un producteur Kafka est développé en langage Python. Ce producteur simule plusieurs capteurs IoT envoyant périodiquement des données environnementales vers le topic Kafka.

Chaque message est structuré au format JSON et contient les informations suivantes :

— l'identifiant du capteur (*deviceId*),

— la température mesurée,

— le taux d'humidité,

— la consommation énergétique,

— le niveau de batterie,

— l'état du capteur,

— un horodatage Unix.

kafka-producer > producer.py > ...

```python
from kafka import KafkaProducer
import json, time, random

producer = KafkaProducer(
    bootstrap_servers='kafka:29092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

while True:
        data = {
            "deviceId": f"sensor{random.randint(1,10)}",
            "temperature": round(random.uniform(20.0, 35.0), 2),
            "humidity": round(random.uniform(30.0, 70.0), 2),
            "energy_kwh": round(random.uniform(0.5, 5.0), 2),
            "battery_level": random.randint(10, 100),
            "status": random.choice(["OK", "WARNING", "CRITICAL"]),
            "timestamp": int(time.time())
        }
        producer.send('iot-sensors', data)
        time.sleep(5)
        print(f"Sent: {data}")
```

Ce producteur génère un flux continu de données, simulant fidèlement le comportement de capteurs IoT dans un contexte industriel.

2.1.3 Déploiement du producteur dans l'environnement Docker

L'ensemble du projet repose sur une architecture conteneurisée basée sur Docker. Afin d'assurer la communication entre le producteur Kafka et les autres composants du système, il est nécessaire d'exécuter le producteur dans le même réseau Docker.

La vérification des réseaux Docker disponibles est effectuée à l'aide de la commande suivante :

```
docker network ls
```

```
PS C:\Users\pc\Documents\iot-spark> docker network ls
NETWORK ID     NAME                          DRIVER    SCOPE
11503b87a35e   bridge                        bridge    local
912fb0fc4d0d   host                          host      local
9ef5adff81f8   iot-spark_bigdata-network     bridge    local
8d2dcbfc7bce   none                          null      local
```

Un conteneur Python temporaire est ensuite lancé en se connectant explicitement au réseau Docker du projet :

```
docker run -it –network iot-spark_bigdata-network -v C :/Users/pc/Documents/iot-spark/kafka-producer :/producer python :3.11 bash
```

À l'intérieur du conteneur, la bibliothèque Kafka est installée :

```
pip install kafka-python cd /producer
```

Le producteur est ensuite lancé :

```
python producer.py
```

Cette étape permet de garantir que les messages sont correctement envoyés vers Kafka dans un environnement distribué.

```
Created topic iot-sensors.
[appuser@bd0dac30709e ~]$ kafka-console-consumer \
>   --bootstrap-server kafka:9092 \
>   --topic iot-sensors \
>   --from-beginning

{"sensor_id": 10, "temperature": 25.74, "humidity": 50.47, "timestamp": 1766572602}
{"sensor_id": 4, "temperature": 22.73, "humidity": 59.5, "timestamp": 1766572603}
{"sensor_id": 7, "temperature": 35.92, "humidity": 40.53, "timestamp": 1766572605}
{"sensor_id": 2, "temperature": 34.57, "humidity": 33.37, "timestamp": 1766572607}
{"sensor_id": 2, "temperature": 15.96, "humidity": 58.35, "timestamp": 1766572608}
{"sensor_id": 7, "temperature": 39.95, "humidity": 30.53, "timestamp": 1766572612}
{"sensor_id": 7, "temperature": 18.88, "humidity": 60.18, "timestamp": 1766572615}
{"sensor_id": 9, "temperature": 28.38, "humidity": 78.78, "timestamp": 1766572621}
{"sensor_id": 7, "temperature": 29.7, "humidity": 43.14, "timestamp": 1766572626}
{"sensor_id": 1, "temperature": 19.03, "humidity": 75.88, "timestamp": 1766572628}
{"sensor_id": 10, "temperature": 30.16, "humidity": 76.02, "timestamp": 1766572629}
{"sensor_id": 7, "temperature": 27.69, "humidity": 38.88, "timestamp": 1766572632}
{"sensor_id": 4, "temperature": 32.77, "humidity": 38.66, "timestamp": 1766572633}
{"sensor_id": 1, "temperature": 24.68, "humidity": 57.46, "timestamp": 1766572636}
{"sensor_id": 1, "temperature": 18.92, "humidity": 70.8, "timestamp": 1766572637}
{"sensor_id": 7, "temperature": 25.03, "humidity": 79.9, "timestamp": 1766572638}
{"sensor_id": 2, "temperature": 37.25, "humidity": 58.0, "timestamp": 1766572639}
{"sensor_id": 9, "temperature": 22.75, "humidity": 72.97, "timestamp": 1766572642}
{"sensor_id": 8, "temperature": 35.29, "humidity": 38.25, "timestamp": 1766572648}
{"sensor_id": 4, "temperature": 25.13, "humidity": 45.29, "timestamp": 1766572650}
{"sensor_id": 3, "temperature": 20.06, "humidity": 66.52, "timestamp": 1766572652}
{"sensor_id": 8, "temperature": 39.28, "humidity": 73.44, "timestamp": 1766572661}
{"sensor_id": 5, "temperature": 15.92, "humidity": 57.46, "timestamp": 1766572664}
{"sensor_id": 4, "temperature": 27.93, "humidity": 67.5, "timestamp": 1766572668}
{"sensor_id": 2, "temperature": 32.15, "humidity": 64.45, "timestamp": 1766572672}
```

2.1.4 Traitement des flux en temps réel avec Apache Spark Streaming

Cette étape constitue le cœur du pipeline Big Data, puisqu'elle assure le traitement en temps réel des données IoT consommées depuis Apache Kafka. Le moteur Apache Spark Streaming est utilisé afin de transformer, enrichir et stocker les flux de données de manière distribuée et scalable.

Organisation du projet Spark

Le projet Spark est structuré selon les conventions standard de l'écosystème Scala et SBT. Le code source est placé dans l'arborescence suivante :

```
spark-work-dir/ src/ main/ scala/ IotStreaming.scala
```

Cette organisation permet une séparation claire entre le code applicatif, les dépendances et les artefacts de build, facilitant ainsi la maintenance et l'évolution du projet.

Implémentation de l'application Spark Streaming

L'application Spark est développée en langage Scala sous la forme d'un objet principal nommé `IotStreaming`. Elle repose sur l'API Structured Streaming de Spark, qui offre un modèle unifié pour le traitement batch et streaming.

Un schéma explicite est défini afin de structurer les messages JSON reçus depuis Kafka :

```
10  val schema = new StructType()
11    .add("deviceId", StringType)
12    .add("temperature", DoubleType)
13    .add("humidity", DoubleType)
14    .add("energy_kwh", DoubleType)
15    .add("battery_level", IntegerType)
16    .add("status", StringType)
17    .add("timestamp", LongType)
```

La création d'un schéma strict permet de garantir la cohérence des données, d'éviter les erreurs de typage et de faciliter les opérations analytiques ultérieures.

Création de la SparkSession

L'application démarre par l'initialisation d'une `SparkSession`, point d'entrée principal pour toutes les opérations Spark :

```
19    val spark = SparkSession.builder
20      .appName("IotStreaming")
21      .getOrCreate()
```

Le niveau de logs est volontairement réduit afin d'améliorer la lisibilité de l'exécution :

```
spark.sparkContext.setLogLevel("WARN")
```

Lecture des flux Kafka

Spark Streaming agit comme consommateur Kafka en lisant les messages publiés dans le topic `iot-sensors`. La connexion est établie via l'adresse réseau Docker du broker Kafka.

```
val df = spark
.readStream
.format("kafka")
.option("kafka.bootstrap.servers", "kafka:29092")
.option("subscribe", "iot-sensors")
.option("startingOffsets", "earliest")
.load()
```

Les données Kafka sont initialement lues sous forme binaire. Une conversion explicite est nécessaire afin de transformer les messages en chaînes JSON exploitables.

Désérialisation et transformation des données

Les messages Kafka sont désérialisés à l'aide de la fonction `from_json`, puis projetés sous forme de colonnes structurées :

```
val jsonDf = df
  .selectExpr("CAST(value AS STRING)")
  .select(from_json(col("value"), schema).as("data"))
  .select("data.*")
```

Une phase d'enrichissement est ensuite appliquée afin d'ajouter de nouvelles métriques utiles à l'analyse :

— conversion de la température de Celsius vers Fahrenheit,

— transformation de l'horodatage Unix en timestamp lisible.

```
val resultDf = jsonDf .withColumn("temperatureF", col("temperature") * 9 / 5 + 32) .withColumn("eventTime", to_timestamp(col("timestamp")/1000))
```

Ces transformations illustrent l'utilisation des fonctions Spark SQL et démontrent la capacité de Spark à enrichir les flux en temps réel.

Écriture des données dans HDFS

Les données traitées sont persistées dans le système de fichiers distribué HDFS au format Parquet, un format colonne optimisé pour l'analytique Big Data. Ce format offre plusieurs avantages :

— une réduction de l'espace de stockage,

— de meilleures performances de lecture,

— une compatibilité native avec Spark SQL.

```
resultDf.writeStream
.format("parquet")
.option("path", "hdfs://namenode:9000/data/iot-output")
.option("checkpointLocation", "hdfs://namenode:9000/checkpoints/iot")
.trigger(Trigger.ProcessingTime("10 seconds"))
.outputMode("append")
.start()
.awaitTermination()
```

```
PS C:\Users\pc\Documents\iot-spark> docker exec -it namenode hdfs dfs -ls /data/iot-output
Found 16 items
drwxr-xr-x   - root supergroup          0 2025-12-26 02:18 /data/iot-output/_spark_metadata
-rw-r--r--   3 root supergroup     149811 2025-12-26 02:18 /data/iot-output/part-00000-18da4a47-275c-4ef7-8b0e-0ad05f98f4d7-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       1863 2025-12-26 02:18 /data/iot-output/part-00000-28a4036b-9899-4ea4-945c-0bf48a6a5490-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       2018 2025-12-26 02:18 /data/iot-output/part-00000-c926c90b-f0a6-4bb2-980a-012d5eb8d68e-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       1928 2025-12-26 02:18 /data/iot-output/part-00000-e2765c04-a6da-44ed-917e-22e698c5b565-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       2051 2025-12-26 02:18 /data/iot-output/part-00000-efbd73ad-c140-4c9d-8e88-d77e15d55dd8-c
000.snappy.parquet
-rw-r--r--   3 root supergroup     155875 2025-12-26 02:18 /data/iot-output/part-00001-60916e0c-9a68-42d6-9d89-5edbde1cf496-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       1863 2025-12-26 02:18 /data/iot-output/part-00001-676237d6-cf9d-43be-85e2-fb88c75dd02b-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       1890 2025-12-26 02:18 /data/iot-output/part-00001-c2c7fdf4-ebc9-4896-a87f-cce1dde25b06-c
000.snappy.parquet
-rw-r--r--   3 root supergroup       1960 2025-12-26 02:18 /data/iot-output/part-00001-c97a204f-6179-4b68-8fba-665bb687dbfc-c
000.snappy.parquet
```

Le mécanisme de *checkpointing* permet de garantir la tolérance aux pannes et la reprise automatique du streaming en cas d'arrêt du job.

Gestion des dépendances avec SBT

Le projet est compilé à l'aide de l'outil SBT. Le fichier `build.sbt` définit les dépendances nécessaires, notamment celles liées à Spark et Kafka.

```
name := "iot-streaming" version := "0.1" scalaVersion := "2.12.15" libraryDependencies ++= Seq(
"org.apache.spark" "org.apache.spark" "org.apache.spark" "org.apache.spark" )
```

La compilation du projet est effectuée via la commande suivante :

```
sbt clean package
```

Cette commande génère un fichier JAR exécutable contenant l'ensemble du code et des dépendances nécessaires à l'exécution du job Spark.

Déploiement du JAR dans le conteneur Spark

Le fichier JAR généré est copié dans le conteneur `spark-master` afin d'être exécuté dans l'environnement distribué :

```
docker cp target/scala-2.12/iot-streaming_2.12 - 0.1.jar spark - master : /spark/work - dir/iot - streaming.jar
```

```
PS C:\Users\pc\Documents\iot-spark\spark\work-dir> docker cp target/scala-2.12/iot-streaming_2.1
2-0.1.jar spark-master:/spark/work-dir/iot-streaming.jar
Successfully copied 0B to spark-master:/spark/work-dir/iot-streaming.jar
```

Cette étape assure que le job Spark est accessible depuis le cluster.

Soumission du job Spark Streaming

Le traitement en temps réel est lancé à l'aide de la commande `spark-submit`, exécutée directement dans le conteneur Spark :

```
docker exec -it spark-master bash -c " /spark/bin/spark-submit –class IotStreaming – master spark ://spark-master :7077 –packages org.apache.spark :spark-sql-kafka-0-10_2.12 : 3.3.0 /spark/work - dir/iot - streaming.jar"
```

Cette commande permet de :

— connecter Spark au cluster,

— consommer les flux Kafka en continu,

— traiter les données en micro-batchs,

— stocker les résultats dans HDFS.

```
+--------+------------------+-----------------+------------+-----------------+--------------------+
|deviceId|       temperature|         humidity|   timestamp|     temperatureF|           eventTime|
+--------+------------------+-----------------+------------+-----------------+--------------------+
| sensor2|28.796500975588778|44.1401715512395|1766706514338|83.83370175605981|2025-12-25 23:48:...|
+--------+------------------+-----------------+------------+-----------------+--------------------+

-------------------------------------------
Batch: 45
-------------------------------------------
+--------+------------------+-----------------+------------+-----------------+--------------------+
|deviceId|       temperature|         humidity|   timestamp|     temperatureF|           eventTime|
+--------+------------------+-----------------+------------+-----------------+--------------------+
| sensor5|23.033971586136907|40.52424689708683|1766706515340|73.46114885504643|2025-12-25 23:48:...|
+--------+------------------+-----------------+------------+-----------------+--------------------+

-------------------------------------------
Batch: 46
-------------------------------------------
+--------+------------------+-----------------+------------+-----------------+--------------------+
|deviceId|       temperature|         humidity|   timestamp|     temperatureF|           eventTime|
+--------+------------------+-----------------+------------+-----------------+--------------------+
| sensor3|21.178337115721757|67.42422447398968|1766706516341|70.12100680829916|2025-12-25 23:48:...|
+--------+------------------+-----------------+------------+-----------------+--------------------+
```

Ainsi, Spark Streaming assure un traitement distribué, fiable et à faible latence des données IoT, répondant pleinement aux exigences du projet Big Data.

2.1.5 Analyse batch et analytique avancée avec Spark SQL et RDD

Après la phase de traitement en temps réel et de stockage des données IoT dans HDFS, une étape d'analyse batch est mise en place. Cette phase permet d'exploiter les données historiques accumulées afin d'extraire des indicateurs clés de performance (KPI) et de réaliser des analyses avancées.

Cette analyse est implémentée à l'aide d'une application Spark distincte, nommée `IotAnalytics`, écrite en langage Scala.

Chargement des données depuis HDFS

Les données stockées au format Parquet dans HDFS sont chargées dans Spark sous forme de DataFrame :

```
val df = spark.read.parquet("hdfs://namenode:9000/data/iot-output")
df.createOrReplaceTempView("iot_data")
```

Le format Parquet permet une lecture efficace des données grâce à son stockage en colonnes, ce qui est particulièrement adapté aux requêtes analytiques massives.

La création d'une vue temporaire (`iot_data`) rend les données directement exploitables via Spark SQL.

Analyse avancée avec Spark SQL

Spark SQL est utilisé afin d'effectuer des agrégations complexes sur l'ensemble des données collectées. Cette approche permet de démontrer l'utilisation du SQL distribué dans un contexte Big Data.

Les indicateurs calculés par appareil incluent :

— la température moyenne,

— le taux d'humidité moyen,

— la consommation énergétique totale,

— le niveau minimal de batterie observé.

```
val analyticsDf = spark.sql("""
  SELECT
    deviceId,
    round(avg(temperature), 2) as avgTemp,
    round(avg(humidity), 2) as avgHum,
    round(sum(energy_kwh), 2) as totalEnergy,
    min(battery_level) as minBattery
  FROM iot_data
  GROUP BY deviceId
""")
analyticsDf.show()
```

Cette étape permet de produire des indicateurs synthétiques exploitables pour la prise de décision et la supervision des capteurs IoT.

Spark SQL offre une interface familière aux analystes tout en bénéficiant de la puissance du calcul distribué, permettant de traiter de grands volumes de données de manière performante.

Traitement bas niveau avec les RDD

Afin de satisfaire les exigences académiques du module, une analyse bas niveau est également réalisée à l'aide des RDD (Resilient Distributed Datasets).

```
val rdd = df.rdd
val countByDevice = rdd
  .map(row => (row.getAs[String]("deviceId"), 1))
  .reduceByKey(_ + _)
```

Cette opération permet de calculer le nombre total de messages reçus par chaque capteur IoT.

Les RDD offrent un contrôle plus fin sur les transformations distribuées et illustrent le fonctionnement fondamental de Spark, notamment les opérations de type *map* et *reduce*.

Calcul des KPI globaux

Des indicateurs globaux sont calculés afin de fournir une vision d'ensemble du système IoT :

— nombre total de capteurs actifs,

— température moyenne globale,

— consommation énergétique totale,

— niveau moyen de batterie.

```
val globalStats = df.select(
  countDistinct("deviceId").as("total_sensors"),
  round(avg("temperature"), 2).as("avg_temp_global"),
  round(sum("energy_kwh"), 2).as("total_energy_consumed"),
  round(avg("battery_level"), 1).as("avg_battery_global")
)
```

Ces KPI sont destinés à être affichés sous forme de valeurs clés dans un tableau de bord décisionnel.

Sauvegarde des résultats analytiques dans HDFS

Les résultats des analyses sont sauvegardés dans HDFS au format CSV afin de faciliter leur exploitation par des outils de visualisation externes.

```
globalStats.coalesce(1).write
  .mode("overwrite")
  .option("header", "true")
  .csv("hdfs://namenode:9000/data/kpi-global")

deviceStats.coalesce(1).write
  .mode("overwrite")
  .option("header", "true")
  .csv("hdfs://namenode:9000/data/kpi-devices")
```

L'utilisation de `coalesce(1)` permet de générer un fichier unique par rapport, facilitant le transfert et l'analyse ultérieure.

Exécution du job Spark Analytics

L'application analytique est soumise au cluster Spark à l'aide de la commande suivante :

```
docker exec -it spark-master /spark/bin/spark-submit –master local[*] –class IotAnalytics –driver-memory 1G /spark/work-dir/iot-streaming.jar
```

```
ark-master, 33983, None)
=== ANALYSE SPARK SQL (DATAFRAME) ===
+--------+-------+------+-----------+----------+
|deviceId|avgTemp|avgHum|totalEnergy|minBattery|
+--------+-------+------+-----------+----------+
| sensor7|  27.36| 47.43|      46.62|        10|
| sensor1|  25.03| 54.85|      61.13|        15|
|sensor10|  27.44| 45.82|      73.33|        17|
| sensor4|  24.99| 55.03|      80.09|        10|
| sensor5|  25.03| 54.99|      55.15|        11|
| sensor8|  26.68| 50.04|      76.71|        13|
| sensor3|  25.03| 54.99|       86.6|        14|
| sensor6|  27.57| 47.42|      61.98|        11|
| sensor2|   25.0| 55.21|      50.73|        13|
| sensor9|  27.68| 49.49|      34.94|        13|
+--------+-------+------+-----------+----------+

=== ANALYSE BAS NIVEAU (RDD) ===
Nombre de messages reçus par appareil (via RDD) :
Appareil: sensor2 -> Messages: 5501
Appareil: sensor9 -> Messages: 12
Appareil: sensor3 -> Messages: 5491
Appareil: sensor4 -> Messages: 5624
Appareil: sensor5 -> Messages: 5478
Appareil: sensor10 -> Messages: 25
Appareil: sensor6 -> Messages: 27
Appareil: sensor7 -> Messages: 18
Appareil: sensor1 -> Messages: 5463
Appareil: sensor8 -> Messages: 26
=== SAUVEGARDE DES RAPPORTS SUR HDFS ===
=== ANALYSE TERMINÉE ET SAUVEGARDÉE ===
```

Cette étape déclenche l'analyse batch complète des données stockées.

Extraction des résultats depuis HDFS

Après l'analyse batch et le calcul des KPI, les fichiers CSV générés dans HDFS (`global.csv` et `devices.csv`) sont extraits vers le poste Windows pour permettre une visualisation interactive.

```
docker exec -it namenode  hdfs dfs -getmerge /data/kpi-global/*.csv global.csv docker cp namenode :/global.csv ./global.csv
docker exec -it namenode  hdfs dfs -getmerge /data/kpi-devices/*.csv devices.csv docker cp namenode :/devices.csv ./devices.csv
```

Étapes pour la visualisation

1. **Création du script Python :** Un fichier `visualisation.py` est créé pour charger les données et générer les graphiques. Le script utilise `pandas` pour la lecture et la

manipulation des fichiers CSV, et `Plotly` pour créer un dashboard interactif. Le contenu du script inclut :

- Chargement des fichiers CSV exportés depuis HDFS.
- Création d'indicateurs globaux (température moyenne, consommation d'énergie, batterie).
- Graphiques par capteur (barres, camemberts, lignes) pour visualiser les KPI détaillés.
- Configuration d'un layout professionnel avec des couleurs cohérentes et titres adaptés.

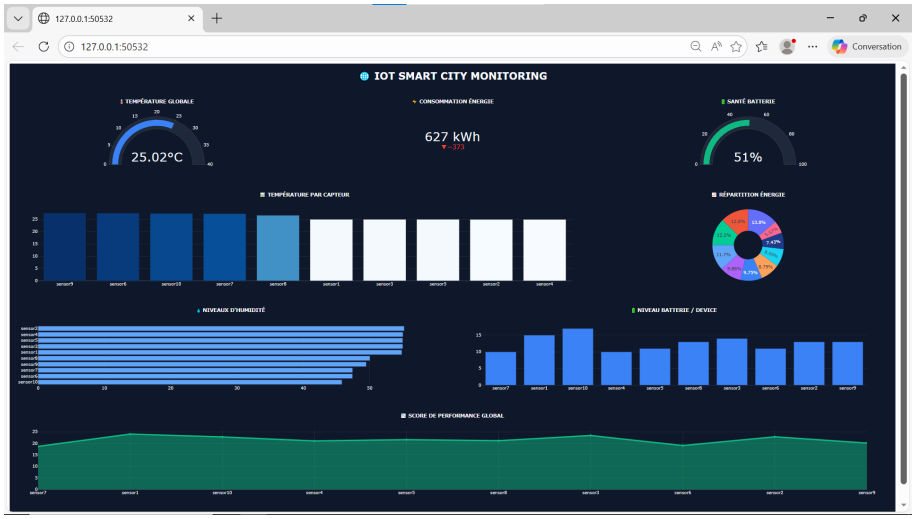
2. **Installation des librairies Python :** Les librairies nécessaires sont installées via pip :

```
pip install pandas matplotlib pyarrow plotly
```

3. **Exécution du script :** Le script est lancé dans l'environnement Python correspondant à ton projet :

```
C :/Users/pc/Documents/iot-spark/.venv/Scripts/python.exe visualisation.py
```

4. **Affichage du dashboard :** Le script génère un dashboard interactif comprenant :
   - Les KPIs globaux sous forme de jauges et indicateurs numériques.
   - Les analyses par capteur avec graphiques en barres et camemberts.
   - Un score global de performance illustré par un graphique en ligne.
5. **Documentation visuelle :**



Le dashboard final affiche 8 graphiques clés permettant un suivi complet des capteurs et de leur environnement :

(a) **Jauge de Température** : État thermique global de l'ensemble des capteurs.

(b) **Total Énergie** : Consommation cumulée en kWh de tous les appareils.

(c) **Santé Batterie** : Moyenne de charge restante, utile pour la maintenance.

(d) **Température par Sensor** : Identification des points chauds et anomalies par capteur.

(e) **Répartition Énergie** : Part de chaque appareil dans la consommation totale.

(f) **Humidité** : Visualisation des niveaux d'humidité par zone ou capteur.

(g) **Batterie par Device** : Liste des niveaux de batterie individuels pour planifier la maintenance.

(h) **Score de Performance** : Calcul algorithmique synthétique de la santé globale du système.

Ainsi, le pipeline complet de collecte, traitement et visualisation est fonctionnel, depuis les capteurs IoT jusqu'au dashboard décisionnel.

3. Journal des Problèmes Rencontrés

Cette section décrit les difficultés techniques rencontrées lors de la mise en place de la plateforme Big Data IoT et les solutions adoptées. Elle reflète l'expérience pratique acquise au cours du projet.

3.1 Problème 1 : HDFS Safe Mode

Symptôme : Erreur d'écriture lors du lancement de Spark.
Cause : Le NameNode était en mode "Safe Mode" car les blocs n'étaient pas encore répliqués après le redémarrage de Docker.
Solution : Utilisation de la commande suivante pour quitter le mode Safe Mode :

```
hdfs dfsadmin -safemode leave
```

3.2 Problème 2 : Dépendances Kafka-Spark

Symptôme : ClassNotFoundException lors du spark-submit.
Cause : Les connecteurs Kafka ne sont pas inclus par défaut dans Spark.
Solution : Ajout manuel du package lors du spark-submit :

```
–packages org.apache.spark :spark-sql-kafka-0-10_2.12 : 3.x.x
```

3.3 Problème 3 : Superposition des graphiques sur le Dashboard

Symptôme : Les titres et les chiffres se chevauchaient sur le dashboard.
Cause : Mauvaise configuration des `specs` dans `Plotly Subplots`.
Solution : Redéfinition de la grille en 4x6 avec des `colspan` et ajustement des `row_heights`.

3.4 Problème 4 : Sérialisation Scala

Symptôme : Erreur "Task not serializable".

Cause : Tentative d'utiliser un objet non sérialisable à l'intérieur d'une transformation RDD.

Solution : Utilisation de variables locales ou d'objets `case class` pour garantir la sérialisation.

4. Conclusion Générale

La réalisation de ce projet a permis de concevoir et de mettre en œuvre une plateforme Big Data complète pour l'ingestion, le traitement et l'analyse de flux de données IoT. Le pipeline développé couvre toutes les étapes : production de données, ingestion via Kafka, traitement en temps réel avec Spark Streaming, stockage distribué sur HDFS, analyse batch avec Spark SQL et RDD, ainsi que la visualisation interactive à l'aide de Plotly.

Les principaux résultats et enseignements sont les suivants :

— **Scalabilité et fiabilité :** Le système supporte un grand nombre de capteurs avec un traitement rapide et tolérant aux pannes.

— **Analytique avancée :** Extraction de KPI pertinents et génération de dashboards interactifs pour le suivi en temps quasi réel.

— **Expérience pratique :** Résolution de problèmes liés à la configuration de HDFS, aux dépendances Spark-Kafka, à la sérialisation Scala et à la visualisation.

— **Intégration complète :** De la génération des données à la prise de décision, toutes les étapes ont été implémentées dans un workflow cohérent.

En conclusion, ce projet illustre de manière concrète la mise en œuvre d'une solution Big Data pour l'IoT, démontrant l'importance de l'architecture distribuée, de l'ingestion en temps réel et de l'analytique avancée dans les systèmes modernes.