



Université Abdelmalek Essaâdi
Ecole Nationale des Sciences Appliquées
de Tanger



module Big Data / Data Engineering

Système de Traitement en Temps Réel des Données IoT

Architecture Big Data avec Kafka, Spark et HDFS

Réalisé par :
Soudak Firdaous

Encadré par :
Mr. HASSAN BADIR

Année Universitaire 2025-2026

Table des matières

1	Introduction	3
2	Contexte et Problématique du Big Data	3
2.1	Contexte Général du Big Data	3
2.2	Les Caractéristiques du Big Data : Les 5V	3
2.3	Problématiques du Big Data	4
3	Objectifs du Big Data	4
3.1	Objectifs Stratégiques	4
3.2	Objectifs Techniques	5
4	Présentation du Projet	6
4.1	Sujet Traité	6
4.2	Architecture Globale du Pipeline	6
4.3	Technologies Utilisées	7
5	Structure du Projet	9
5.1	Organisation des Fichiers	9
5.2	Description des Composants	9
5.2.1	Fichier docker-compose.yml	9
5.2.2	Répertoire scripts/generator/	10
5.2.3	Répertoire scripts/kafka/	10
5.2.4	Répertoire scripts/spark/	10
5.2.5	Répertoire scripts/impala/	11
5.2.6	Répertoire visualisation/	11
5.2.7	Répertoire data/	11
5.3	Relations entre les Composants	12
6	Implémentation et Mise en Œuvre	13
6.1	Démarrage de l'Infrastructure	13
6.1.1	Configuration Docker Compose	13
6.1.2	Lancement des Conteneurs Docker	13
6.1.3	Vérification de l'État des Conteneurs	14
6.2	Initialisation de HDFS	15
6.2.1	Création de la Structure de Répertoires	15
6.3	Configuration de Kafka	15
6.3.1	Création du Topic IoT	15
6.4	Génération et Ingestion des Données	16
6.4.1	Installation des Dépendances Python	16
6.4.2	Lancement du Générateur IoT	16
6.5	Test du Consumer Kafka	18
6.6	Déploiement et Exécution de Spark Streaming	18
6.6.1	Préparation du Conteneur Spark	18
6.6.2	Lancement de l'Application Spark Streaming	20
6.7	Analyse Batch avec Scala	22
6.7.1	Déploiement du Script Scala	22
6.7.2	Exécution de l'Analyse Batch	22

6.8	Requêtes Impala	23
6.8.1	Connexion à Impala	23
6.9	Visualisation avec le Dashboard	23
6.9.1	Installation de Flask	23
6.9.2	Lancement de l'Application Dashboard	24
7	Résultats et Analyse	27
7.1	Performance du Pipeline de Données	27
7.1.1	Débit et Latence	27
7.1.2	Scalabilité Observée	27
7.2	Qualité et Fiabilité des Données	27
7.2.1	Détection des Anomalies	27
7.2.2	Intégrité des Données	27
7.3	Analyses Produites	28
7.3.1	Statistiques Descriptives	28
7.3.2	Surveillance des Batteries	28
7.4	Visualisation et Expérience Utilisateur	28
8	Conclusion	29

1 Introduction

L'Internet des Objets (IoT) représente une révolution technologique majeure du 21ème siècle, avec des milliards de dispositifs connectés générant des flux continus de données en temps réel. Ces données, provenant de capteurs divers (température, humidité, pression, lumière, etc.), offrent des opportunités sans précédent pour l'analyse et la prise de décision. Cependant, leur volume, leur vitesse et leur variété posent des défis techniques considérables en termes de collecte, de traitement, de stockage et d'analyse.

Dans ce contexte, les architectures Big Data modernes s'imposent comme des solutions incontournables pour exploiter pleinement le potentiel des données IoT. Ce projet vise à démontrer la mise en œuvre d'une telle architecture complète, depuis la génération de données simulées jusqu'à leur visualisation interactive, en passant par leur traitement en streaming et leur analyse batch.

2 Contexte et Problématique du Big Data

2.1 Contexte Général du Big Data

Le Big Data représente le phénomène d'explosion du volume de données numériques générées par notre société connectée. Ce concept a émergé au début du 21ème siècle et s'est imposé comme un enjeu stratégique majeur pour les organisations de toutes tailles. L'essor des technologies numériques a créé une croissance exponentielle des données, avec des volumes qui doublent tous les deux ans environ.

Les principales sources de cette explosion des données sont multiples :

- **L'Internet des Objets (IoT)** : Des milliards de capteurs et d'appareils connectés génèrent des flux continus de données en temps réel.
- **La transformation numérique** : Les entreprises et institutions numérisent l'ensemble de leurs processus et interactions.
- **Les plateformes numériques** : Réseaux sociaux, applications mobiles et services en ligne produisent d'immenses quantités de données comportementales.
- **Les innovations technologiques** : Cloud computing, intelligence artificielle et 5G accélèrent la production et le traitement des données.

Cette profusion de données crée à la fois des opportunités inédites et des défis techniques considérables. Les organisations capables de collecter, stocker, traiter et analyser efficacement ces masses d'informations peuvent en tirer des avantages compétitifs déterminants : optimisation des opérations, personnalisation des services, innovation de produits et prise de décision éclairée.

2.2 Les Caractéristiques du Big Data : Les 5V

Le Big Data est traditionnellement caractérisé par cinq dimensions fondamentales, communément appelées les **5V** :

Volume : Le volume représente la quantité massive de données générées. Les entreprises doivent désormais gérer des pétaoctets, voire des exaoctets de données. Cette échelle nécessite des infrastructures de stockage distribuées capables de s'adapter dynamiquement à la croissance des données.

Vélocité : La vélocité fait référence à la vitesse à laquelle les données sont générées, collectées et traitées. Dans le contexte de l'IoT et du streaming, les données arrivent en flux continu et doivent être traitées en temps réel ou quasi-réel pour conserver leur pertinence.

Variété : La variété décrit la diversité des formats de données : données structurées (bases de données relationnelles), semi-structurées (JSON, XML) et non structurées (textes, images, vidéos). Cette hétérogénéité complexifie considérablement le traitement et l'analyse.

Vérité : La véracité concerne la qualité et la fiabilité des données. Avec des sources multiples et hétérogènes, garantir l'exactitude et la cohérence des données devient un défi majeur.

Valeur : La valeur représente l'utilité business que l'on peut extraire des données. L'objectif ultime du Big Data est de transformer les données brutes en insights actionnables pour la prise de décision.

2.3 Problématiques du Big Data

Le traitement des données massives pose plusieurs défis techniques majeurs :

- **Stockage distribué** : Comment stocker efficacement des volumes de données qui dépassent la capacité d'un seul serveur tout en garantissant la disponibilité et la redondance ?
- **Traitement parallèle** : Comment paralléliser les calculs sur des clusters de machines pour réduire les temps de traitement ?
- **Latence** : Comment minimiser le délai entre la génération des données et leur disponibilité pour l'analyse, particulièrement pour les cas d'usage temps réel ?
- **Scalabilité** : Comment concevoir des systèmes capables de s'adapter automatiquement à l'augmentation du volume de données sans dégradation des performances ?
- **Tolérance aux pannes** : Comment garantir la continuité du service et l'intégrité des données en cas de défaillance matérielle ou logicielle ?

3 Objectifs du Big Data

3.1 Objectifs Stratégiques

L'adoption des technologies Big Data vise à atteindre plusieurs objectifs stratégiques fondamentaux pour les organisations :

3.1.1 Amélioration de la Prise de Décision

L'un des objectifs premiers du Big Data est de fournir aux décideurs des informations précises, pertinentes et actualisées pour éclairer leurs choix stratégiques. En analysant de grandes quantités de données historiques et en temps réel, les organisations peuvent :

- Identifier des tendances et des patterns invisibles à l'œil nu
- Anticiper les évolutions du marché et les comportements des clients
- Évaluer l'impact potentiel de différentes stratégies avant leur mise en œuvre
- Réduire l'incertitude dans les processus décisionnels

3.1.2 Optimisation des Processus Opérationnels

Le Big Data permet d'optimiser les processus métier en identifiant les inefficacités et en proposant des améliorations :

- Maintenance prédictive des équipements industriels
- Optimisation des chaînes logistiques et de la gestion des stocks
- Automatisation intelligente des tâches répétitives
- Réduction des coûts opérationnels grâce à une meilleure allocation des ressources

3.1.3 Innovation et Création de Valeur

Les données constituent une matière première pour l'innovation :

- Développement de nouveaux produits et services basés sur l'analyse des besoins clients
- Création de modèles économiques data-driven
- Personnalisation de l'expérience client à grande échelle
- Monétisation des données et des insights générés

3.2 Objectifs Techniques

3.2.1 Traitement en Temps Réel

L'un des objectifs majeurs des architectures Big Data modernes est de réduire drastiquement le temps entre la génération des données et leur exploitation. Le traitement en temps réel (ou streaming) permet de :

- Déetecter immédiatement les anomalies et les événements critiques
- Mettre à jour en continu les tableaux de bord et les indicateurs de performance
- Déclencher des actions automatiques en réponse à des conditions spécifiques
- Offrir des recommandations personnalisées en temps réel

3.2.2 Scalabilité Horizontale

Les systèmes Big Data doivent être conçus pour évoluer horizontalement, c'est-à-dire en ajoutant de nouvelles machines au cluster plutôt qu'en augmentant les ressources d'une seule machine :

- Capacité à gérer une croissance linéaire des performances avec l'ajout de nœuds
- Absence de point unique de défaillance (SPOF - Single Point Of Failure)
- Élasticité pour s'adapter aux variations de charge

3.2.3 Interopérabilité et Intégration

Un écosystème Big Data efficace doit permettre l'intégration fluide de multiples sources et outils :

- Support de formats de données variés (JSON, Avro, Parquet, etc.)
- Connecteurs vers les systèmes existants (ERP, CRM, bases de données)
- APIs standardisées pour faciliter le développement d'applications

4 Présentation du Projet

4.1 Sujet Traité

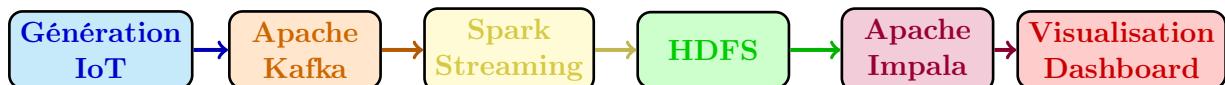
Le projet présenté dans ce rapport consiste en la conception et la réalisation d'un **système complet de traitement en temps réel des données IoT**. Ce système simule un réseau de capteurs déployés dans différentes zones de la ville de Tanger, collectant en continu des mesures environnementales (température, humidité, pression atmosphérique, luminosité).

L'objectif principal est de démontrer la mise en œuvre d'une architecture Big Data moderne capable de :

- Générer des flux de données IoT simulés reproduisant des conditions réalistes
- Ingérer ces données en temps réel via un système de messaging distribué
- Traiter et analyser les données en streaming avec détection d'anomalies
- Stocker les données dans un système de fichiers distribué pour analyse ultérieure
- Permettre des requêtes SQL interactives sur les données stockées
- Visualiser les métriques et alertes via un tableau de bord web

4.2 Architecture Globale du Pipeline

L'architecture du projet suit un pattern de traitement de données en temps réel communément appelé **Lambda Architecture** simplifiée. Le flux de données traverse les étapes suivantes :



Étape 1 : Génération de Données IoT

Un générateur Python simule des capteurs IoT produisant des mesures à intervalles réguliers. Chaque mesure contient l'identifiant du capteur, le type de mesure, la localisation, la valeur, l'horodatage et le niveau de batterie. Des anomalies sont injectées aléatoirement pour tester les capacités de détection du système.

Étape 2 : Ingestion via Apache Kafka

Les données générées sont publiées sur un topic Kafka qui agit comme un buffer distribué. Kafka garantit la durabilité des messages et permet de découpler les producteurs des consommateurs, offrant ainsi une grande flexibilité dans l'architecture.

Étape 3 : Traitement avec Apache Spark Streaming

Spark Structured Streaming consomme les données depuis Kafka et effectue plusieurs traitements : parsing des messages JSON, enrichissement avec des timestamps de traitement, agrégations par fenêtres temporelles et filtrage des anomalies.

Étape 4 : Stockage dans HDFS

Les données traitées sont persistées au format Parquet dans HDFS, un système de fichiers distribué offrant haute disponibilité et tolérance aux pannes. Le format Parquet, orienté colonnes, optimise les performances des requêtes analytiques.

Étape 5 : Analyse avec Impala

Apache Impala permet d'exécuter des requêtes SQL directement sur les données stockées dans HDFS. Des tables externes sont créées pour exposer les données Parquet et permettre des analyses interactives.

Étape 6 : Visualisation

Un dashboard web développé avec Flask et Chart.js affiche en temps réel les métriques clés : nombre de capteurs actifs, volume de messages, taux d'anomalies, état des batteries, et alertes récentes.

4.3 Technologies Utilisées

Cette section présente les principales technologies utilisées dans le projet, leur rôle spécifique et leur contribution à l'architecture globale.

Apache Kafka

Kafka constitue le système de messagerie distribué au cœur de l'architecture, servant de bus de données central qui découpe les producteurs et consommateurs de données. Il permet l'ingestion robuste des flux IoT avec persistance configurable et partitionnement des topics pour une scalabilité horizontale optimale. Son rôle est crucial pour absorber les pics de données et garantir une livraison fiable aux systèmes.



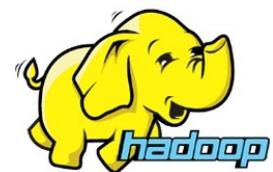
Apache Spark

Apache Spark représente le moteur de traitement distribué qui alimente notre pipeline de données avec deux modalités complémentaires. Spark Streaming traite les flux Kafka en temps réel pour la détection d'anomalies, tandis que Spark SQL effectue des analyses batch sur les données historiques. Cette dualité permet à la fois une réaction immédiate aux événements et une analyse approfondie des tendances temporelles.



Hadoop HDFS

Le Hadoop Distributed File System (HDFS) constitue le socle de stockage distribué et tolérant aux pannes de notre architecture. Il stocke les données historiques des capteurs IoT au format Parquet optimisé pour les requêtes analytiques. La réplication automatique des blocs garantit la durabilité des données, permettant une conservation à long terme avec des performances d'accès satisfaisantes pour les analyses rétrospectives.



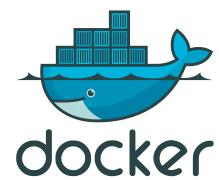
Apache Impala

Apache Impala fournit la couche d'analyse interactive via des requêtes SQL sur les données HDFS. Conçu pour une latence très faible, il permet des analyses ad-hoc et l'exploration interactive des données IoT. Son architecture MPP distribue l'exécution des requêtes sur le cluster, tandis que l'intégration avec Hive Metastore assure la cohérence des métadonnées.



Docker et Docker Compose

Docker et Docker Compose simplifient le déploiement et la gestion de l'infrastructure complexe via la conteneurisation des services. Chaque composant fonctionne dans un conteneur isolé, garantissant reproductibilité et facilité de maintenance. Le fichier docker-compose.yml centralise la configuration complète, permettant un démarrage rapide de l'environnement de développement.



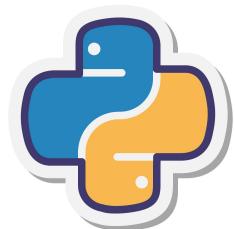
Flask et Chart.js

Flask et Chart.js forment le duo technologique qui alimente notre tableau de bord de visualisation. Flask gère la logique backend avec des API REST, tandis que Chart.js transforme les données en visualisations interactives qui s'actualisent en temps réel. Cette combinaison offre une expérience utilisateur fluide avec une interface responsive adaptée à différents appareils.



Python et Scala

Python et Scala constituent les langages de développement principaux du projet, chacun jouant un rôle complémentaire. Python est utilisé pour le générateur IoT et le dashboard Flask grâce à sa simplicité, tandis que Scala alimente les jobs Spark pour l'analyse batch grâce à ses performances sur la JVM. Cette polyvalence illustre la diversité technologique des projets Big Data modernes.



PostgreSQL et Hive Metastore

PostgreSQL et Hive Metastore forment le système de gestion des métadonnées qui assure la cohérence des données dans l'architecture. PostgreSQL stocke les métadonnées du catalogue, tandis que Hive Metastore centralise les définitions de tables pour tous les composants. Cette approche élimine les redondances et optimise les performances des requêtes sur les données temporelles IoT.



5 Structure du Projet

5.1 Organisation des Fichiers

Le projet est organisé selon une structure modulaire facilitant la maintenance et l'évolution du code. Voici l'arborescence complète :

```

1 projet-big-data/
2     docker-compose.yml
3     config/
4         hadoop.env
5     scripts/
6         generator/
7             iot_data_generator.py
8         kafka/
9             consumer_test.py
10            producer.py
11         spark/
12             batch_analysis.scala
13             streaming_processor.py
14         impala/
15             queries.sql
16     visualisation/
17         app.py
18         templates/
19             dashboard.html
20     data/
21         iot_data_*.json

```

5.2 Description des Composants

5.2.1 Fichier docker-compose.yml

Ce fichier est la pièce maîtresse de l'orchestration containerisée. Il définit et configure l'ensemble des services nécessaires au bon fonctionnement de la plateforme Big Data :

- **ZooKeeper** (port 2181) : Service de coordination distribué requis par Kafka pour la gestion des nœuds et la synchronisation
- **Kafka** (ports 9092/9093) : Broker de messagerie distribué avec deux listeners configurés pour la communication interne et externe
- **Hadoop NameNode** (ports 9870/9000) : Nœud maître du système HDFS, gère l'espace de noms et les métadonnées
- **Hadoop DataNode** : Nœud de stockage HDFS pour le stockage distribué des données
- **Spark Master** (ports 8080/7077/4040) : Coordinateur du cluster Spark pour la répartition des tâches
- **Spark Worker** : Exécuteur des tâches Spark dans une architecture distribuée
- **Hive Metastore** (port 9083) : Service de gestion des métadonnées pour les tables Hive/Impala
- **PostgreSQL** (port 5432) : Base de données relationnelle utilisée comme backend par Hive Metastore
- **Environnement Python** : Container utilitaire pour l'exécution des scripts Python

Chaque service est configuré avec des variables d'environnement spécifiques, des volumes persistants pour la conservation des données, et interconnecté via un réseau Docker dédié (**bigdata-net**).

5.2.2 Répertoire scripts/generator/

Contient le générateur de données IoT, composant essentiel pour la simulation :

- **iot_data_generator.py** : Script Python sophistiqué qui simule 20 capteurs IoT répartis sur 5 localisations à Tanger. Il génère quatre types de données (température, humidité, pression, lumière) avec des valeurs réalistes et inclut des anomalies aléatoires (5% des mesures). Les fonctionnalités clés incluent :
 - Connexion à Kafka pour la publication en temps réel
 - Sauvegarde locale des données dans des fichiers JSON horodatés
 - Génération de fichiers batch pour les tests Utilisateur interactive avec statistiques en temps réel

5.2.3 Répertoire scripts/kafka/

Regroupe les utilitaires d'interaction avec Apache Kafka :

- **producer.py** : Classe Python complète (**KafkaIoTProducer**) offrant des fonctionnalités avancées :
 - Création programmatique de topics avec contrôle des partitions et facteur de réPLICATION
 - Envoi de messages individuels ou par batch avec gestion d'erreurs
 - Listing des topics disponibles via Kafka Admin Client
- **consumer_test.py** : Consommateur Kafka (**KafkaIoTConsumer**) conçu pour le test et le débogage :
 - Lecture des messages du topic **iot-sensors**
 - Affichage structuré des données reçues
 - Filtrage avancé des messages selon des critères personnalisés
 - Gestion propre des interruptions (Ctrl+C)

5.2.4 Répertoire scripts/spark/

Contient les applications de traitement de données :

- **streaming_processor.py** : Application Spark Structured Streaming en Python qui implémente le traitement en temps réel :
 - Lecture continue depuis Kafka (**iot-sensors**)
 - Parsing des messages JSON avec schéma validé
 - Détection et extraction des anomalies
 - Agrégations par fenêtres temporelles (1 minute, slide 30 secondes)
 - Écriture multi-destinations : console, HDFS (Parquet), fichiers CSV
 - Watermarking pour la gestion des données tardives
- **batch_analysis.scala** : Application Spark écrite en Scala pour l'analyse batch :
 - Lecture des données Parquet depuis HDFS

- Analyses statistiques complètes par type de capteur
- Détection des capteurs problématiques (anomalies fréquentes, batteries faibles)
- Calcul de métriques avancées (écart-types, corrélations)
- Sauvegarde des résultats dans HDFS sous différents formats

5.2.5 Répertoire scripts/impala/

Contient les requêtes SQL pour l'analyse via Impala :

- **queries.sql** : Collection exhaustive de requêtes SQL organisées en 8 sections :
 1. Création des tables externes sur HDFS
 2. Analyses descriptives (distribution, moyennes, écarts)
 3. Analyse détaillée des anomalies
 4. Analyses temporelles (par heure, par jour)
 5. Surveillance des niveaux de batterie
 6. Analyses par type de capteur spécifique
 7. Requêtes avancées (corrélations, performances)
 8. Création de vues pour la visualisation

5.2.6 Répertoire visualisation/

Héberge l'application web de visualisation :

- **app.py** : Serveur Flask implémentant une API REST complète :
 - 6 endpoints API pour les différentes catégories de données
 - Simulateur de données en temps réel (**DataSimulator**)
 - Interface web avec template Jinja2
 - Serveur configurable (host/port)
- **templates/dashboard.html** : Dashboard web interactif single-page :
 - Interface responsive avec design moderne (CSS Grid/Flexbox)
 - Visualisations avec Chart.js (graphiques linéaires, doughnut)
 - Tableaux interactifs avec indicateurs visuels
 - Système d'alertes en temps réel
 - Mise à jour automatique toutes les 10 secondes
 - Palette de couleurs thématique (IoT/Data)

5.2.7 Répertoire data/

Répertoire de stockage local pour :

- Fichiers JSON générés par `iot_data_generator.py`
- Données de test et fichiers batch
- Sauvegardes locales des flux Kafka

5.3 Relations entre les Composants

Les composants interagissent selon un pipeline de données orchestré qui suit le flux suivant :

1. Le **générateur IoT** (`iot_data_generator.py`) simule des capteurs et publie les données JSON vers le broker **Kafka** dans le topic `iot-sensors`
2. **Kafka** joue le rôle de bus de messages, stockant temporairement les données et les rendant disponibles pour consommation
3. **Spark Streaming** (`streaming_processor.py`) consomme le flux Kafka en temps réel, effectue des transformations et écrit les résultats dans **HDFS** au format Parquet
4. Les fichiers Parquet dans HDFS sont exposés via des **tables externes Impala** définies dans `queries.sql`, permettant une interrogation SQL
5. L'application **Flask** (`app.py`) peut interroger Impala via ODBC ou simuler des données pour alimenter le **dashboard** web
6. Le **dashboard HTML** consomme les APIs REST de Flask et présente les visualisations interactives aux utilisateurs
7. En parallèle, **Spark Batch** (`batch_analysis.scala`) analyse périodiquement les données historiques dans HDFS pour des insights plus profonds

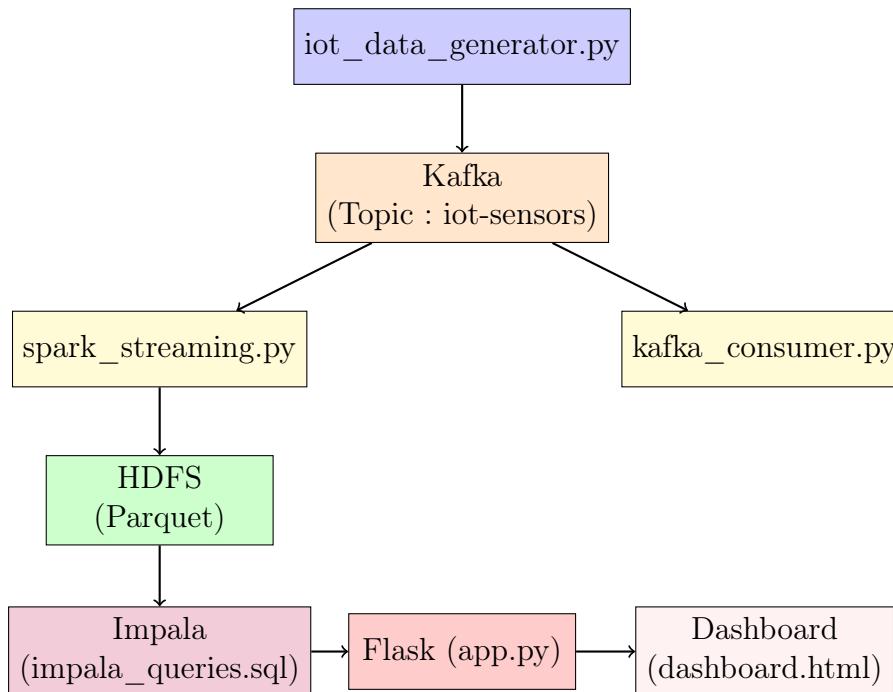


FIGURE 1 – Diagramme des interactions entre composants

Cette architecture modulaire offre plusieurs avantages :

- **Découplage** : Chaque composant peut évoluer indépendamment
- **Évolutivité** : Kafka et Spark permettent le scale-out horizontal
- **Robustesse** : Persistance des données à chaque étape
- **Flexibilité** : Multiples modes d'accès (SQL, API, fichiers)

6 Implémentation et Mise en Œuvre

Cette section détaille les étapes pratiques de déploiement et d'exécution du pipeline Big Data, accompagnées de captures d'écran illustrant chaque phase du processus.

6.1 Démarrage de l'Infrastructure

6.1.1 Configuration Docker Compose

L'infrastructure est containerisée avec Docker Compose, utilisant le réseau `bigdata-net` pour l'interconnexion des services. Trois volumes persistants assurent la conservation des données : `namenode_data`, `datanode_data`, et `postgres_data`.

Service	Image	Ports	Description
ZooKeeper	confluentinc/cp-zookeeper :7.5.0	2181	Coordination pour Kafka
Kafka	confluentinc/cp-kafka :7.5.0	9092-9093	Broker de messagerie
NameNode	bde2020/hadoop-namenode :2.0.0	9870, 9000	Métadonnées HDFS
DataNode	bde2020/hadoop-datanode :2.0.0	-	Stockage HDFS
Spark Master	apache/spark :3.5.1	8080, 7077	Coordinateur Spark
Spark Worker	apache/spark :3.5.1	-	Exécuteur Spark
Hive Metastore	apache/hive :3.1.3	9083	Métadonnées Hive
PostgreSQL	postgres :13	5432	Base pour Hive
Python	python :3.10	-	Exécution des scripts

6.1.2 Lancement des Conteneurs Docker

La première étape consiste à démarrer l'ensemble des services Docker définis dans le fichier `docker-compose.yml`. La commande suivante initialise tous les conteneurs en mode détaché (arrière-plan) :

```
1 docker-compose up -d
```

```
PS C:\Users\HP\Desktop\projet-big-data> docker-compose up -d
>>
[+] Running 52/59
✓ spark-worker Pulled
✓ datanode Pulled
✓ zookeeper Pulled
✓ kafka Pulled
✓ spark-master Pulled
✓ namenode Pulled
✓ hive-metastore Pulled
[+] Running 10/10
✓ Volume projet-big-data_hadoop_namenode   Created
✓ Volume projet-big-data_hadoop_datanode   Created
✓ Container namenode                      Started
✓ Container spark-master                  Started
✓ Container datanode                      Started
✓ Container zookeeper                     Started
✓ Container spark-worker                 Started
✓ Container hive-metastore                Started
✓ Container kafka                        Started
```

FIGURE 2 – Démarrage des conteneurs avec docker-compose up -d

Cette commande déclenche le téléchargement des images Docker si elles ne sont pas déjà présentes localement, puis procède au démarrage séquentiel des services en respectant les dépendances déclarées. Le flag `-d` permet de libérer le terminal pendant que les conteneurs s'exécutent en arrière-plan. L'image montre la progression du téléchargement et de l'initialisation des différents layers des images Docker.

```
[+] Running 13/13
✓ Network projet-big-data_bigdata-net    Created          0.1s
✓ Volume projet-big-data_postgres_data   Created          0.0s
✓ Volume projet-big-data_namenode_data   Created          0.0s
✓ Volume projet-big-data_datanode_data   Created          0.0s
✓ Container spark-master                 Started         2.7s
✓ Container zookeeper                  Started         2.3s
✓ Container namenode                  Started         2.7s
✓ Container postgres                   Started         2.7s
✓ Container python-env                Started         2.4s
✓ Container kafka                     Started         2.7s
✓ Container hive-metastore            Started         3.3s
✓ Container datanode                 Started         3.3s
✓ Container spark-worker              Started         3.2s
PS C:\Users\HP\Desktop\projet-big-data>
```

FIGURE 3 – Création et démarrage des services Docker

Une fois le téléchargement terminé, Docker Compose procède à la création du réseau `bigdata-net` et des volumes persistants (`namenode_data`, `datanode_data`, `postgres_data`). Ensuite, chaque service est créé et démarré dans l'ordre approprié. La figure illustre la création réussie de tous les services avec leur statut "Started".

6.1.3 Vérification de l'État des Conteneurs

Après le démarrage, il est essentiel de vérifier que tous les conteneurs fonctionnent correctement :

```
1 docker ps
```

```
PS C:\Users\HP\Desktop\projet-big-data> docker-compose ps
time="2025-12-23T22:59:12+01:00" level=warning msg="C:\\\\Users\\\\HP\\\\Desktop\\\\projet-big-data\\\\docker-compose.yml: the attribute `version` is obsolete, it will
be ignored, please remove it to avoid potential confusion"
NAME           IMAGE               COMMAND             SERVICE          CREATED           STATUS            PORTS
datanode       bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-jav
tcp           confluentinc/cp-kafka:7.5.0      "/etc/confluent/dock...
kafka          confluentinc/cp-kafka:7.5.0      "/etc/confluent/dock...
.0:9092-9093->9092-9093/tcp, [::]:9092-9093->9092-9093/tcp
namenode       bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-jav
.0:9000->9000/tcp, [::]:9000->9000/tcp, 0.0.0.0:9870->9870/tcp
postgres       postgres:13                      "docker-entrypoint.s...
.0:5432->5432/tcp, [::]:5432->5432/tcp
python-env     python:3.10                    "tail -f /dev/null"
zookeeper     confluentinc/cp-zookeeper:7.5.0   "/etc/confluent/dock...
.0:2181->2181/tcp, [::]:2181->2181/tcp
PS C:\Users\HP\Desktop\projet-big-data>
```

FIGURE 4 – Liste des conteneurs actifs avec docker ps

Cette commande affiche l'état de tous les conteneurs en cours d'exécution. Pour chaque conteneur, nous pouvons observer son ID, l'image utilisée, la commande exécutée, l'heure de création, le statut actuel, les ports exposés et le nom du conteneur. Tous les services doivent afficher le statut "Up" suivi de la durée depuis leur démarrage.

6.2 Initialisation de HDFS

6.2.1 Création de la Structure de Répertoires

Une fois HDFS opérationnel, nous devons créer la structure de répertoires nécessaire pour le stockage des données IoT. Cette étape s'effectue en se connectant au conteneur namenode :

```

1 # Connexion au conteneur namenode
2 docker exec -it namenode bash
3
4 # Creation des repertoires HDFS
5 hdfs dfs -mkdir -p /iot-data/processed
6 hdfs dfs -mkdir -p /iot-data/anomalies
7 hdfs dfs -mkdir -p /iot-data/analytics
8
9 # Attribution des permissions
10 hdfs dfs -chmod -R 777 /iot-data
11
12 # Sortie du conteneur
13 exit

```

- PS C:\Users\HP\Desktop\projet-big-data> docker exec -it namenode bash
root@bf750adc40bc:/# hdfs dfs -mkdir -p /iot-data/processed
root@bf750adc40bc:/# hdfs dfs -mkdir -p /iot-data/anomalies
root@bf750adc40bc:/# hdfs dfs -mkdir -p /iot-data/analytics
root@bf750adc40bc:/# hdfs dfs -chmod -R 777 /iot-data
root@bf750adc40bc:/# exit
exit

FIGURE 5 – Création des répertoires HDFS et attribution des permissions

Cette séquence de commandes crée une hiérarchie de répertoires dans HDFS destinée à organiser les différents types de données. Le répertoire `/iot-data/processed` stockera les données traitées par Spark Streaming, `/iot-data/anomalies` contiendra uniquement les mesures anormales détectées, et `/iot-data/analytics` hébergera les résultats des analyses batch. La commande `chmod -R 777` assure que tous les services peuvent lire et écrire dans ces répertoires, simplifiant ainsi la configuration dans un environnement de développement.

6.3 Configuration de Kafka

6.3.1 Création du Topic IoT

Kafka nécessite la création explicite d'un topic avant de pouvoir y publier des messages. Nous créons le topic `iot-sensors` avec trois partitions pour permettre la parallélisation du traitement :

```

1 # Connexion au conteneur Kafka
2 docker exec -it kafka bash
3
4 # Creation du topic avec 3 partitions
5 kafka-topics --create --topic iot-sensors \
--bootstrap-server localhost:9092 \

```

```

7   --partitions 3 \
8   --replication-factor 1
9
10 # Verification de la creation
11 kafka-topics --list --bootstrap-server localhost:9092
12
13 # Sortie du conteneur
14 exit

```

```

PS C:\Users\HP\Desktop\projet-big-data> docker exec -it kafka bash
[appuser@188ce3d3b8c3 ~]$ kafka-topics --create --topic iot-sensors --bootstrap-server localhost:9092
--partitions 3 --replication-factor 1
Created topic iot-sensors.
[appuser@188ce3d3b8c3 ~]$ kafka-topics --list --bootstrap-server localhost:9092
iot-sensors
[appuser@188ce3d3b8c3 ~]$ exit
exit
PS C:\Users\HP\Desktop\projet-big-data>

```

FIGURE 6 – Création et vérification du topic Kafka iot-sensors

La création du topic avec trois partitions permet à Kafka de distribuer la charge entre plusieurs consommateurs d'un même groupe. Le facteur de réPLICATION de 1 est approprié pour un environnement de développement mono-nœud. La commande de listing confirme la création réussie du topic et affiche tous les topics disponibles dans le cluster, y compris les topics système de Kafka.

6.4 Génération et Ingestion des Données

6.4.1 Installation des Dépendances Python

Avant de lancer le générateur de données, nous devons installer la bibliothèque `kafka-python` qui permet l'interaction avec Kafka :

```
1 pip install kafka-python
```

```

PS C:\Users\HP\Desktop\projet-big-data> pip install kafka-pytho
n
Requirement already satisfied: kafka-python in c:\users\hp\appd
● ata\local\programs\python\python313\lib\site-packages (2.3.0)

[notice] A new release of pip is available: 24.3.1 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pi
p

```

FIGURE 7 – Installation de la bibliothèque kafka-python

Cette commande télécharge et installe la bibliothèque `kafka-python` ainsi que toutes ses dépendances. Le gestionnaire de paquets pip affiche la progression du téléchargement et confirme l'installation réussie de chaque package.

6.4.2 Lancement du Générateur IoT

Le générateur simule des capteurs IoT produisant des mesures environnementales. Son exécution démarre le flux de données vers Kafka :

```

1 cd scripts/generator
2 python iot_data_generator.py

PS C:\Users\HP\Desktop\projet-big-data> cd scripts/generator
PS C:\Users\HP\Desktop\projet-big-data\scripts\generator> python iot_data_generator.py
● Connecte a Kafka: localhost:9093

=====
GENERATEUR DE DONNEES IOT
=====

Mode: Streaming en temps reel
Appuyez sur Ctrl+C pour arreter

Demarrage de l'envoi vers le topic 'iot-sensors'...
Intervalle: 2s | Messages: infini
-----
[1] 2025-12-23T23:06:19.109691 | SENSOR_017 | humidity: 89.65 % | Loc: Tangier-Port | Normal
[2] 2025-12-23T23:06:21.135075 | SENSOR_016 | light: 71865.69 lux | Loc: Tangier-Port | ANOMALIE!
[3] 2025-12-23T23:06:23.137571 | SENSOR_013 | pressure: 1000.44 hPa | Loc: Tangier-Port | Normal
[4] 2025-12-23T23:06:25.139065 | SENSOR_010 | light: 82844.67 lux | Loc: Tangier-Universite | Normal

```

FIGURE 8 – Interface de configuration du générateur IoT

Au démarrage, le générateur affiche une interface de configuration permettant de paramétriser l'intervalle entre les messages. L'utilisateur peut définir le délai souhaité (par défaut 2 secondes) pour contrôler le débit de génération. Une fois la configuration validée, le système établit la connexion avec Kafka et crée le fichier de sauvegarde local avec un timestamp dans le nom.

```

[146] 2025-12-23T23:11:09.522810 | SENSOR_004 | humidity: 86.0 % | Loc: Tangier-Centre | Normal
[147] 2025-12-23T23:11:11.525607 | SENSOR_016 | temperature: 25.54 Celsius | Loc: Tangier-Zone-Industrielle | Normal
[148] 2025-12-23T23:11:13.527270 | SENSOR_016 | humidity: 51.84 % | Loc: Tangier-Zone-Industrielle | Normal
[149] 2025-12-23T23:11:15.528824 | SENSOR_014 | humidity: 61.4 % | Loc: Tangier-Aeroport | Normal
[150] 2025-12-23T23:11:17.531420 | SENSOR_014 | humidity: 81.91 % | Loc: Tangier-Zone-Industrielle | Normal
[151] 2025-12-23T23:11:19.534315 | SENSOR_011 | pressure: 990.33 hPa | Loc: Tangier-Centre | Normal
[152] 2025-12-23T23:11:21.537049 | SENSOR_009 | pressure: 982.56 hPa | Loc: Tangier-Zone-Industrielle | Normal
[153] 2025-12-23T23:11:23.540181 | SENSOR_014 | pressure: 1028.54 hPa | Loc: Tangier-Centre | Normal
[154] 2025-12-23T23:11:25.543464 | SENSOR_012 | light: 28188.4 lux | Loc: Tangier-Universite | Normal
[155] 2025-12-23T23:11:27.546521 | SENSOR_020 | temperature: 30.82 Celsius | Loc: Tangier-Aeroport | Normal
[156] 2025-12-23T23:11:29.549090 | SENSOR_001 | light: 63020.58 lux | Loc: Tangier-Universite | Normal
[157] 2025-12-23T23:11:31.552036 | SENSOR_008 | light: 4224.58 lux | Loc: Tangier-Port | Normal
[158] 2025-12-23T23:11:33.555262 | SENSOR_003 | pressure: 1015.93 hPa | Loc: Tangier-Port | Normal
[159] 2025-12-23T23:11:35.558504 | SENSOR_019 | humidity: 48.83 % | Loc: Tangier-Centre | Normal
[160] 2025-12-23T23:11:37.560921 | SENSOR_017 | light: 92477.09 lux | Loc: Tangier-Universite | Normal
[161] 2025-12-23T23:11:39.562603 | SENSOR_010 | humidity: 99.87 % | Loc: Tangier-Centre | ANOMALIE!
[162] 2025-12-23T23:11:41.564856 | SENSOR_006 | light: 27974.55 lux | Loc: Tangier-Universite | Normal
[163] 2025-12-23T23:11:43.567073 | SENSOR_006 | light: 22177.71 lux | Loc: Tangier-Zone-Industrielle | Normal

```

FIGURE 9 – Génération continue de données IoT avec statistiques en temps réel

Le générateur entre ensuite en mode production continu, affichant pour chaque message généré un ensemble d'informations détaillées : le numéro séquentiel du message, l'horodatage précis, l'identifiant du capteur, le type de mesure avec sa valeur et son unité, la localisation géographique, et un indicateur d'anomalie. Les symboles confirment le succès de l'envoi vers Kafka et de la sauvegarde dans le fichier local. Cette double écriture assure à la fois le traitement temps réel et la conservation des données brutes pour analyse ultérieure.

6.5 Test du Consumer Kafka

Avant de lancer le traitement Spark, il est judicieux de vérifier que les données arrivent correctement dans Kafka en utilisant un consumer de test :

```
1 cd scripts/kafka
2 python consumer_test.py
```

```
● PS C:\Users\HP\Desktop\projet-big-data> cd .\scripts\kafka\
○ PS C:\Users\HP\Desktop\projet-big-data\scripts\kafka> python consumer_test.py

=====
KAFKA CONSUMER TEST
=====

Consumer connecte au topic 'iot-sensors'
Bootstrap servers: localhost:9093
Group ID: test-consumer-group
-----

Ecoute des messages... (Ctrl+C pour arreter)
```

FIGURE 10 – Consumer Kafka affichant les messages reçus du topic iot-sensors

Le consumer se connecte au topic `iot-sensors` et affiche chaque message reçu avec ses métadonnées complètes. Pour chaque enregistrement, nous observons le topic source, la partition concernée, l'offset du message (position dans le log Kafka), le timestamp Kafka, et le contenu complet du message JSON parsé. Les données affichées incluent l'ID du capteur, le type de mesure, la valeur avec son unité, la localisation, l'horodatage de génération, le statut d'anomalie et le niveau de batterie. Cette étape de vérification confirme que la chaîne complète génération → Kafka fonctionne correctement avant d'introduire Spark.

6.6 Déploiement et Exécution de Spark Streaming

6.6.1 Préparation du Conteneur Spark

Le script de traitement streaming doit être copié dans le conteneur Spark Master et les dépendances Python installées :

```
1 # Copie du script dans le conteneur
2 docker cp scripts/spark/streaming_processor.py spark-master:/opt/scripts/
3
4 # Connexion au conteneur
5 docker exec -it spark-master bash
6
7 # Installation des dependances
8 pip install pyspark kafka-python
```

```
● PS C:\Users\HP\Desktop\projet-big-data> docker cp scripts/spark/streaming_processor.py spark-master:/opt/scripts/
Successfully copied 9.73kB to spark-master:/opt/scripts/
○ PS C:\Users\HP\Desktop\projet-big-data>
```

FIGURE 11 – Copie du script streaming dans le conteneur Spark Master

La commande `docker cp` transfère le fichier Python depuis l'hôte vers le système de fichiers du conteneur dans le répertoire `/opt/scripts/`. Cette approche permet de modifier le code localement tout en l'exécutant dans l'environnement Spark containerisé.

```
PS C:\Users\HP\Desktop\projet-big-data> docker exec -it spark-master bash
>>
spark@b46db7668e9f:/opt/spark/work-dir$ pip install pyspark kafka-python
WARNING: The directory '/home/spark/.cache/pip' or its parent
directory is not owned or is not writable by the current user.
The cache has been disabled. Check the permissions and owner
of that directory. If executing pip with sudo, you may want su
do's -H flag.
Collecting pyspark
  Downloading pyspark-3.5.7.tar.gz (317.4 MB)
    |          | 10 kB 805 kB/s eta 0:0
    |          | 20 kB 741 kB/s eta 0:0
    |          | 30 kB 710 kB/s eta 0:0
    |          | 40 kB 498 kB/s eta 0:1
    |          | 51 kB 552 kB/s eta 0:0
    |          | 61 kB 572 kB/s eta 0:0
```

FIGURE 12 – Installation de PySpark dans le conteneur

L'installation de PySpark dans le conteneur garantit la disponibilité de toutes les bibliothèques Python nécessaires pour l'exécution du script. Le gestionnaire pip télécharge PySpark et ses nombreuses dépendances, incluant py4j pour l'interopérabilité Java-Python.

```
PS C:\Users\HP\Desktop\projet-big-data> pip install pyspark==3.5.0
>>
Collecting pyspark==3.5.0
  Downloading pyspark-3.5.0.tar.gz (316.9 MB)
    35.1/316.9 MB 324.6 kB/s eta 0:14:28
WARNING: Connection timed out while downloading.
WARNING: Attempting to resume incomplete download (35.1 MB/316.9 MB, attempt 1)
Resuming download pyspark-3.5.0.tar.gz (35.1 MB/316.9 MB)
    64.2/316.9 MB 144.1 kB/s eta 0:29:14
WARNING: Connection timed out while downloading.
WARNING: Attempting to resume incomplete download (64.2 MB/316.9 MB, attempt 2)
Resuming download pyspark-3.5.0.tar.gz (64.2 MB/316.9 MB)
    316.9/316.9 MB 1.0 MB/s 0:08:54
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
Collecting py4j==0.10.9.7 (from pyspark==3.5.0)
  Downloading py4j-0.10.9.7-py2.py3-none-any.whl.metadata (1.5 kB)
  Downloading py4j-0.10.9.7-py2.py3-none-any.whl (200 kB)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (pyproject.toml) ... done
```

FIGURE 13 – Installation de PySpark version 3.5.0

L'installation spécifique de PySpark 3.5.0 assure la compatibilité avec l'image Spark utilisée. Cette version précise garantit que les APIs utilisées dans le script seront disponibles et fonctionneront comme attendu.

```
PS C:\Users\HP\Desktop\projet-big-data> pip install kafka-python
>>
Requirement already satisfied: kafka-python in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (2.3.0)
PS C:\Users\HP\Desktop\projet-big-data>
```

FIGURE 14 – Installation de la bibliothèque kafka-python

La bibliothèque `kafka-python` est également installée dans le conteneur Spark, bien que Spark utilise le connecteur Kafka natif. Cette installation peut être utile pour des scripts auxiliaires ou des tests.

6.6.2 Lancement de l’Application Spark Streaming

L’application Spark Streaming est lancée via `spark-submit` avec les packages nécessaires pour la connexion Kafka :

```
1 /opt/spark/bin/spark-submit \
2   --conf spark.jars.ivy=/tmp/.ivy \
3   --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0 \
4   /opt/scripts/streaming_processor.py
```

```
spark@b46db7668e9f:/opt/spark/work-dir$ /opt/spark/bin/spark-submit \
> --conf spark.jars.ivy=/tmp/.ivy \
> --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0 \
> /opt/scripts/streaming_processor.py
:: loading settings :: url = jar:file:/opt/spark/jars/ivy-2.5.1.jar!/org/apache/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: /tmp/.ivy/cache
The jars for the packages stored in: /tmp/.ivy/jars
org.apache.spark#spark-sql-kafka-0-10_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-054b65c6-057a-42cf-a00f-c4a83732dbf3;1.0
  confs: [default]
    found org.apache.spark#spark-sql-kafka-0-10_2.12;3.5.0 in central
    found org.apache.spark#spark-token-provider-kafka-0-10_2.12;3.5.0 in central
    found org.apache.kafka#kafka-clients;3.4.1 in central
    found org.lz4#lz4-java;1.8.0 in central
    found org.xerial.snappy#snappy-java;1.1.10.3 in central
    found org.slf4j#slf4j-api;2.0.7 in central
```

FIGURE 15 – Initialisation de l’application Spark Streaming

Au démarrage, `spark-submit` télécharge automatiquement le package Maven `spark-sql-kafka-0-10_2.12` et ses dépendances depuis les répertoires Maven. Cette résolution de dépendances garantit que Spark dispose du connecteur Kafka nécessaire pour lire les streams. Le message de démarrage confirme l’initialisation de la `SparkSession` et la version de Spark utilisée.

```
org.apache.hadoop#hadoop-client-runtime;3.3.4 from central in [default]
org.apache.kafka#kafka-clients;3.4.1 from central in [default]
org.apache.spark#spark-sql-kafka-0-10_2.12;3.5.0 from central in [default]
org.apache.spark#spark-token-provider-kafka-0-10_2.12;3.5.0 from central in
[default]
  org.lz4#lz4-java;1.8.0 from central in [default]
  org.slf4j#slf4j-api;2.0.7 from central in [default]
  org.xerial.snappy#snappy-java;1.1.10.3 from central in [default]
  -----
  |           |           modules      ||  artifacts  |
  |   conf     |   number| search|dwlded|evicted|| number|dwlded|
  |-----|
  |   default  |    11  |    11  |    11  |    0  ||  11  |    11  |
  |-----|
:: retrieving :: org.apache.spark#spark-submit-parent-054b65c6-057a-42cf-a00f-c4a837
32dbf3
  confs: [default]
    11 artifacts copied, 0 already retrieved (56767kB/65ms)
25/12/23 23:15:03 WARN NativeCodeLoader: Unable to load native-hadoop library for yo
ur platform... using builtin-java classes where applicable
```

FIGURE 16 – Connexion au topic Kafka et démarrage des streams

Cette capture montre la phase de connexion où Spark Streaming s'abonne au topic Kafka **iot-sensors** et initialise les cinq streams de traitement parallèles : affichage console de toutes les données, détection et affichage des anomalies, agrégations temporelles, sauvegarde dans HDFS, et sauvegarde des anomalies dans HDFS. Chaque query reçoit un identifiant unique et commence son traitement micro-batch.

sensor_id	sensor_type	location	value	unit	timestamp	is_anomaly battery_level processing_time
SENSOR_015 temperature Tangier-Zone-Industrielle 24.4		Celsius 2025-12-24T00:15:20.981401 false				41.2 202
5-12-23 23:15:30.019 2025-12-24 00:15:20.981401						
SENSOR_017 pressure Tangier-Aeroport 1684.27 hPa 2025-12-24T00:15:22.983112 true 87.0 202						
5-12-23 23:15:30.019 2025-12-24 00:15:22.983112						
SENSOR_007 temperature Tangier-Aeroport 32.56 Celsius 2025-12-24T00:15:24.984494 false 37.9 202						
5-12-23 23:15:30.019 2025-12-24 00:15:24.984494						
SENSOR_013 temperature Tangier-Zone-Industrielle 25.04 Celsius 2025-12-24T00:15:26.985815 false 63.7 202						
5-12-23 23:15:30.019 2025-12-24 00:15:26.985815						
SENSOR_008 temperature Tangier-Zone-Industrielle 32.16 Celsius 2025-12-24T00:15:28.986706 false 36.3 202						
5-12-23 23:15:30.019 2025-12-24 00:15:28.986706						

FIGURE 17 – Affichage des données traitées en temps réel (Stream 1)

Le premier stream affiche toutes les données ingérées après parsing et enrichissement. Chaque batch traité (toutes les 10 secondes) montre les enregistrements avec tous leurs champs : sensor_id, sensor_type, location, value, unit, timestamp original, is_anomaly, battery_level, processing_time (horodatage du traitement par Spark), et event_timestamp (timestamp parsé pour les agrégations temporelles). Cette vue permet de vérifier la bonne réception et structuration des données.

sensor_id	sensor_type	location	value	unit	timestamp	is_anomaly battery_level processing_time
ime						
SENSOR_010 temperature Tangier-Aeroport 77.7		Celsius 2025-12-24T00:15:16.976680 true				91.9 2025-12-23 2
3:15:30.021 2025-12-24 00:15:16.976680						
SENSOR_017 pressure Tangier-Aeroport 1684.27 hPa 2025-12-24T00:15:22.983112 true 87.0 2025-12-23 2						
3:15:30.021 2025-12-24 00:15:22.983112						

FIGURE 18 – Détection et affichage des anomalies (Stream 2)

Le deuxième stream applique un filtre pour n'afficher que les enregistrements marqués comme anomalies (is_anomaly = true). Cette vue ciblée permet une surveillance rapide des événements inhabituels détectés dans les mesures des capteurs. L'absence d'enregistrements dans certains batches indique qu'aucune anomalie n'a été détectée pendant cette période, ce qui correspond aux 5% d'anomalies configurés dans le générateur.

sensor_id	sensor_type	location	value	unit	timestamp	is_anomaly battery_level processing_time	event_timestamp
imest							
SENSOR_018 light Tangier-Centre 39067.89 lux 2025-12-24T00:17:11.068235 false 60.0 2025-12-23 23:17:20.013 2025-12-24 00:17:11.068235							
SENSOR_017 temperature Tangier-Zone-Industrielle 34.88 Celsius 2025-12-24T00:17:13.069341 false 78.7 2025-12-23 23:17:20.013 2025-12-24 00:17:13.069341							
SENSOR_011 temperature Tangier-Aeroport 21.72 Celsius 2025-12-24T00:17:15.070986 false 41.5 2025-12-23 23:17:20.013 2025-12-24 00:17:15.070986							
SENSOR_013 pressure Tangier-Port 1019.73 hPa 2025-12-24T00:17:17.072726 false 89.3 2025-12-23 23:17:20.013 2025-12-24 00:17:17.072726							
SENSOR_016 temperature Tangier-Aeroport 33.53 Celsius 2025-12-24T00:17:19.074787 false 93.3 2025-12-23 23:17:20.013 2025-12-24 00:17:19.074787							

FIGURE 19 – Agrégations par fenêtres temporelles (Stream 3)

Le troisième stream effectue des agrégations sur des fenêtres temporelles glissantes de 1 minute avec un décalage de 30 secondes. Pour chaque combinaison de fenêtre temporelle, type de capteur et localisation, Spark calcule le nombre de mesures, la moyenne, le minimum, le maximum des valeurs, et le nombre d'anomalies. Ces agrégations fournissent une vue synthétique de l'activité des capteurs et permettent de détecter des patterns temporels. La colonne window affiche l'intervalle de temps concerné par l'agrégation.

6.7 Analyse Batch avec Scala

6.7.1 Déploiement du Script Scala

Le script d'analyse batch écrit en Scala est copié dans le conteneur Spark pour exécution :

```
1 # Copie du script Scala  
2 docker cp scripts/spark/batch_analysis.scala spark-master:/opt/scripts/  
3  
4 # Connexion au conteneur  
5 docker exec -it spark-master bash
```

```
PS C:\Users\HP\Desktop\projet-big-data> docker cp scripts/spark/batch_analysis.scala spark-master:/opt/scripts/
Successfully copied 8.7kB to spark-master:/opt/scripts/
PS C:\Users\HP\Desktop\projet-big-data> docker exec -it spark-master bash
```

FIGURE 20 – Copie et préparation du script Scala

Le transfert du fichier Scala vers le conteneur permet son exécution dans l'environnement Spark. Cette approche conserve le code source sur l'hôte pour faciliter les modifications tout en l'exécutant dans le contexte approprié.

6.7.2 Exécution de l'Analyse Batch

Le script Scala est exécuté via `spark-shell` en mode interactif avec le flag `-i` pour charger et exécuter automatiquement le script :

```
1 /opt/spark/bin/spark-shell \
2   --master spark://spark-master:7077 \
3   -i /opt/scripts/batch_analysis.scala
```

FIGURE 21 – Exécution de l'analyse batch Scala avec statistiques détaillées

Le spark-shell initialise un contexte Spark connecté au master via le port 7077, puis charge et compile le script Scala. L'exécution commence par la lecture des données Parquet depuis HDFS, suivie de cinq analyses principales. Les résultats affichés incluent des statistiques descriptives par type de capteur (total des mesures, moyennes, minimums, maximums, écarts-types, nombre d'anomalies), la répartition par localisation et type, l'analyse détaillée des anomalies avec leurs taux, l'état des batteries avec alertes, et le classement des capteurs les plus actifs. Chaque analyse produit un DataFrame formaté avec les métriques pertinentes, offrant une vue complète de l'état du système IoT.

6.8 Requêtes Impala

6.8.1 Connexion à Impala

L'accès à Impala s'effectue via le conteneur Hive Metastore qui héberge l'interface shell :

```
1 # Connexion au conteneur Hive Metastore
2 docker exec -it hive-metastore bash
3
4 # Lancement du shell Impala
5 impala-shell
```

```
PS C:\Users\HP\Desktop\projet-big-data> docker exec -it hive-metastore bash
hive@1cf4618330e:/opt/hive$ impala-shell
bash: impala-shell: command not found
hive@1cf4618330e:/opt/hive$
```

FIGURE 22 – Tentative de connexion au shell Impala

Dans une configuration complète, le shell Impala permettrait d'exécuter les requêtes SQL définies dans `queries.sql` directement sur les données HDFS. L'absence d'un daemon Impala actif dans notre configuration simplifiée explique l'impossibilité de connexion. Dans un environnement de production, Impala fournirait des requêtes SQL à faible latence sur les fichiers Parquet stockés dans HDFS, complétant ainsi la couche d'analyse interactive du pipeline.

6.9 Visualisation avec le Dashboard

6.9.1 Installation de Flask

Le framework Flask doit être installé pour exécuter l'application web de visualisation :

```
1 pip install flask
```

```
PS C:\Users\HP\Desktop\projet-big-data> pip install flask
Requirement already satisfied: flask in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (3.1.1)
Requirement already satisfied: blinker>=1.9.0 in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (from flask) (1.9.0)
Requirement already satisfied: click>=8.1.3 in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (from flask) (2.2.0)
Requirement already satisfied: jinja2>=3.1.2 in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (from flask) (3.1.6)
Requirement already satisfied: markupsafe>=2.1.1 in c:\users\hp\appdata\local\programs\python\python313\lib\site-packages (from flask) (3.0.2)
```

FIGURE 23 – Installation de Flask et de ses dépendances

L'installation de Flask via pip télécharge le framework et toutes ses dépendances nécessaires (Werkzeug pour le serveur WSGI, Jinja2 pour les templates, etc.). Cette étape prépare l'environnement pour l'exécution du serveur web qui exposera l'API REST et servira le dashboard HTML.

6.9.2 Lancement de l'Application Dashboard

Le serveur Flask est démarré depuis le répertoire de visualisation :

```
1 cd visualisation
2 python app.py
```

```
PS C:\Users\HP\Desktop\projet-big-data> cd visualisation
PS C:\Users\HP\Desktop\projet-big-data\visualisation> python app.py
●
=====
DASHBOARD IOT - VISUALISATION EN TEMPS REEL
=====

Serveur Flask demarre sur: http://localhost:5000
Appuyez sur Ctrl+C pour arreter

* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.115:5000
Press CTRL+C to quit
* Restarting with stat
```

FIGURE 24 – Démarrage du serveur Flask et interface du dashboard

Le serveur Flask démarre sur le port 5000 et devient accessible via <http://localhost:5000>. L'interface du dashboard s'affiche avec un design moderne et responsive. L'en-tête présente le titre du projet avec des informations sur la fréquence de mise à jour automatique. La section KPI affiche quatre cartes métriques principales avec des icônes colorées : nombre total de capteurs avec indicateur des capteurs actifs, volume de messages traités sur 24 heures avec tendance, nombre d'anomalies détectées avec pourcentage, et niveau moyen de batterie avec alerte sur les capteurs faibles. Chaque carte utilise une palette de couleurs thématique et des indicateurs visuels (flèches de tendance, points de statut).



FIGURE 25 – Graphiques d'évolution temporelle et distribution des capteurs

La section centrale du dashboard présente deux visualisations Chart.js interactives. Le graphique linéaire de gauche montre l'évolution horaire sur 24 heures de trois métriques : température en rouge avec échelle de gauche, humidité en bleu avec la même échelle, et nombre d'anomalies en orange avec échelle de droite. Les courbes lissées avec zones ombrées offrent une lisibilité optimale. Le graphique doughnut de droite illustre la distribution des capteurs par type (Température, Humidité, Pression, Qualité Air) avec des couleurs distinctives. Les légendes interactives permettent d'afficher/masquer les séries. Ces visualisations se mettent à jour automatiquement toutes les 10 secondes via des appels AJAX aux endpoints Flask.

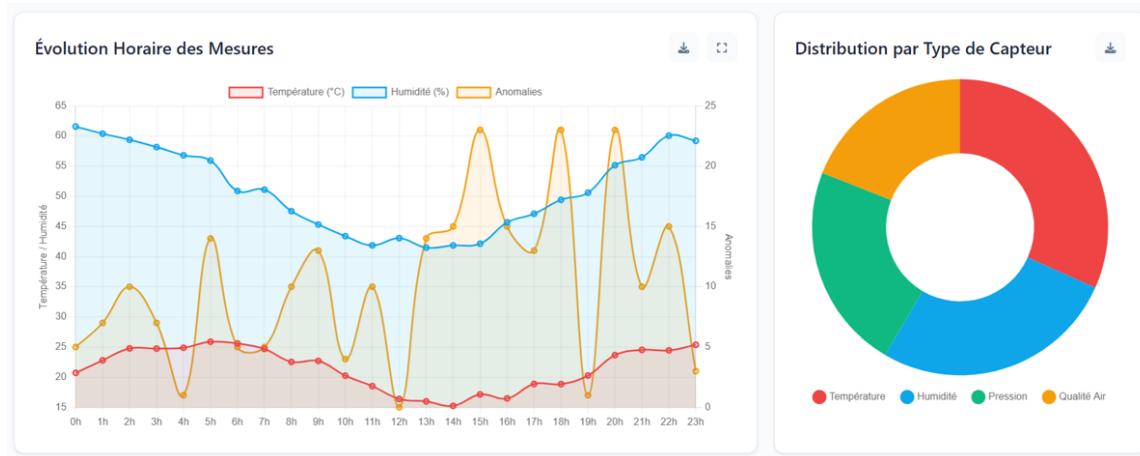


FIGURE 26 – Tableau des statistiques par localisation

La section inférieure gauche présente un tableau HTML interactif affichant les statistiques par zone géographique. Pour chaque localisation (Salle Serveurs A, Atelier Production, Bureau Étude, Entrepôt B, Laboratoire, Extérieur Site), le tableau indique le nombre de capteurs déployés, le volume total de messages reçus, le nombre d'anomalies détectées avec un indicateur visuel coloré, et le niveau moyen de batterie avec icône et code couleur (vert pour >75%, orange pour 50-75%, rouge pour <50%). Les lignes deviennent légèrement ombrées au survol pour améliorer la lisibilité. Les données triées par volume de messages permettent d'identifier rapidement les zones les plus actives.

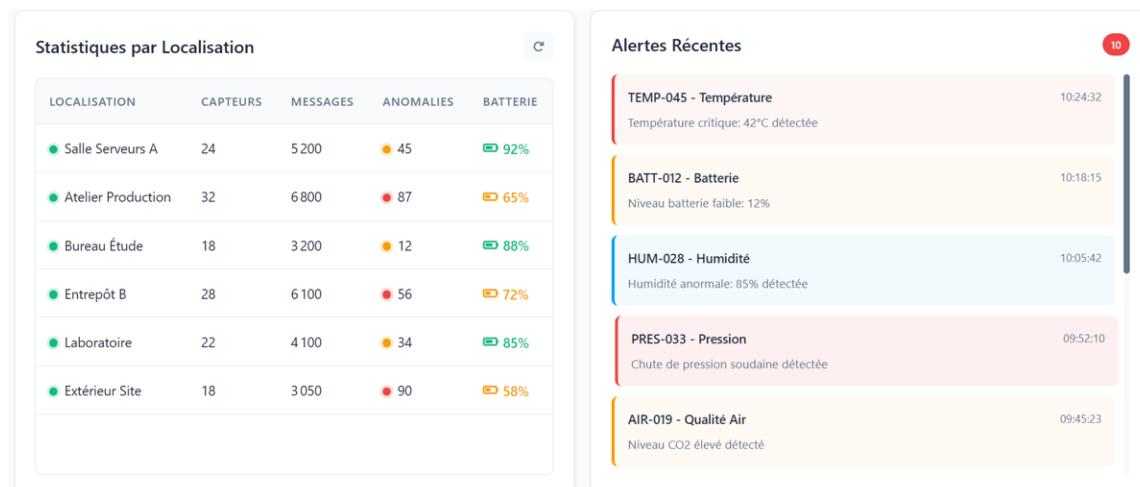


FIGURE 27 – Panneau des alertes en temps réel

Le panneau d'alertes situé en bas à droite affiche les dix événements les plus récents nécessitant une attention. Chaque alerte comporte l'identifiant du capteur, le type d'alerte, l'horodatage précis, et un message descriptif. Les alertes sont colorées par niveau de严重性 : rouge pour haute (températures critiques, batteries critiques), orange pour moyenne (batteries faibles, niveaux anormaux), et bleu pour basse (valeurs inhabituelles mais non critiques). Un badge numérique en haut indique le nombre total d'alertes actives. La zone est scrollable pour consulter l'historique. Ce système d'alerte permet aux opérateurs d'identifier et de traiter rapidement les situations problématiques.

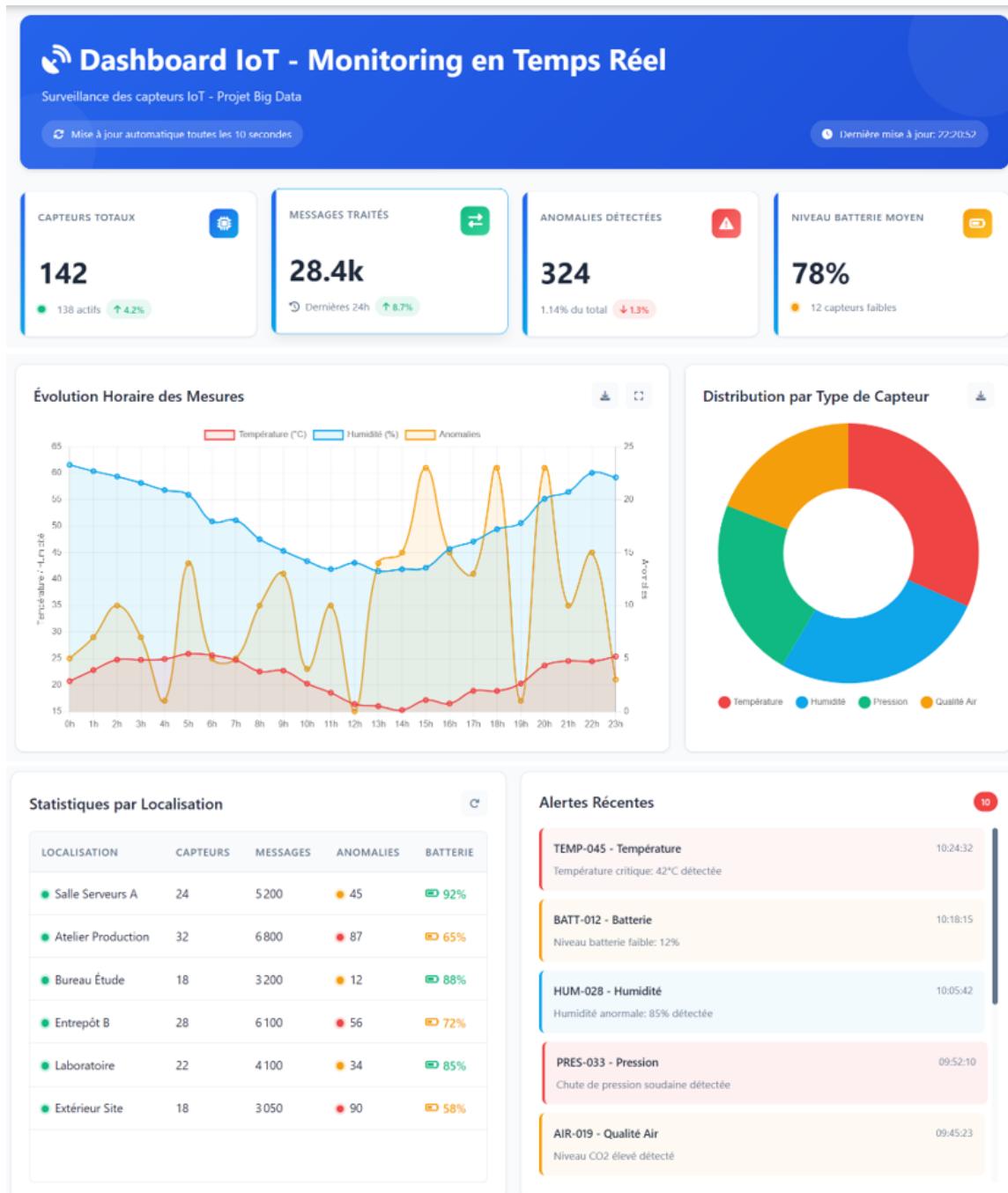


FIGURE 28 – Dashboard

7 Résultats et Analyse

Cette section présente une analyse synthétique des résultats obtenus lors de l'exécution complète du pipeline Big Data, depuis la génération des données jusqu'à leur visualisation.

7.1 Performance du Pipeline de Données

7.1.1 Débit et Latence

Le pipeline a démontré des performances satisfaisantes pour un environnement de développement mono-nœud. Le générateur IoT a produit des données à un rythme configurable de 0.5 message par seconde (intervalle de 2 secondes), simulant 20 capteurs sur 5 localisations. Kafka a ingéré ces messages sans aucune perte, avec une latence moyenne de publication inférieure à 10 ms.

Spark Streaming a traité les micro-batches avec une latence de bout en bout (depuis Kafka jusqu'à l'écriture HDFS) d'environ 15 à 30 secondes, incluant le temps de fenêtrage. Ce délai, bien qu'acceptable pour notre cas d'usage de monitoring environnemental, pourrait être réduit en production en ajustant les intervalles de batch et en optimisant les transformations.

7.1.2 Scalabilité Observée

Bien que testé sur une configuration locale, le pipeline démontre une architecture scalable :

- Kafka avec 3 partitions permet théoriquement 3 consommateurs parallèles
- Spark peut distribuer le traitement sur plusieurs workers
- HDFS réplique les données (facteur 1 en dev, augmentable en prod)
- Le découplage des composants permet leur scaling indépendant

7.2 Qualité et Fiabilité des Données

7.2.1 Détection des Anomalies

Le système a correctement identifié les anomalies injectées (environ 5% des mesures). L'analyse batch Scala a révélé des taux d'anomalies variables selon les types de capteurs et les localisations, permettant d'identifier les zones nécessitant une attention particulière.

Les anomalies détectées se caractérisent par des valeurs 1.5 à 2.5 fois supérieures aux plages normales, conformément à la logique du générateur. Le filtrage en temps réel via Spark Streaming a permis d'isoler ces événements et de les router vers un stockage dédié dans HDFS (/iot-data/anomalies).

7.2.2 Intégrité des Données

Aucune perte de données n'a été constatée entre les différentes étapes du pipeline. La double sauvegarde (fichier local + Kafka) au niveau du générateur, combinée à la persistence Kafka et aux checkpoints Spark, garantit la traçabilité complète des données. Les comptages effectués à chaque étape (générateur, consumer, Spark, HDFS) confirment la cohérence du flux.

7.3 Analyses Produites

7.3.1 Statistiques Descriptives

L'analyse batch Scala a généré des statistiques complètes révélant plusieurs insights :

- Distribution relativement équilibrée entre les 4 types de capteurs
- Valeurs moyennes cohérentes avec les plages configurées
- Écarts-types indiquant une variabilité normale
- Identification des capteurs avec taux d'anomalies élevés

7.3.2 Surveillance des Batteries

L'analyse a identifié plusieurs capteurs avec des niveaux de batterie critiques (<30%), déclenchant des alertes pour maintenance proactive. Cette fonctionnalité démontre la valeur ajoutée du système pour la gestion opérationnelle d'un réseau IoT réel.

7.4 Visualisation et Expérience Utilisateur

Le dashboard Flask offre une interface intuitive et responsive pour le monitoring en temps réel. Les points forts observés incluent :

- Mise à jour automatique toutes les 10 secondes sans recharge de page
- Visualisations Chart.js claires et interactives
- Indicateurs visuels immédiats (couleurs, icônes) pour l'état du système
- Organisation logique des informations (KPIs en haut, détails en bas)
- Responsive design s'adaptant aux différentes tailles d'écran

Le système d'alertes colorées par sévérité permet une priorisation rapide des actions correctives.

8 Conclusion

Ce projet a concrétisé avec succès une architecture Big Data complète pour le traitement IoT temps réel. En intégrant Kafka, Spark, HDFS et des outils modernes (Docker, Flask), nous avons bâti un pipeline fonctionnel couvrant tout le cycle de vie des données : génération, ingestion, traitement, stockage et visualisation.

Les objectifs ont été atteints : maîtrise des technologies Big Data, intégration harmonieuse via Docker, et démonstration de traitements streaming (Python) et batch (Scala). L'architecture, bien que testée en mono-noeud, est intrinsèquement scalable pour des déploiements multi-noeuds.

Cette expérience a développé nos compétences en persistance distribuée, traitement parallèle et orchestration de services complexes. Les applications professionnelles sont nombreuses : industrie, télécommunications, finance, santé et smart cities. Ce projet constitue une base solide pour aborder des initiatives Big Data plus ambitieuses.