Université Moulay Ismaïl
École Normale Supérieure (ENS)
-Meknès-

Master Systèmes intelligents pour L'Éducation

Module : Base de données avancée et Big Data éducatif

Rapport de Projet
ThermoAlert– Projet de Surveillance Temperature

Big Data - Spark Clustering

Réalisé par :
Ahmed BOUBA

Sous la direction de :
Pr. Younes HAJOUI (ENS)

Année universitaire
2024/2025

Résumé

Dans ce projet, j'ai construit un système de surveillance de température en temps réel pour détecter des anomalies et générer des alertes. Mon objectif était de montrer comment des outils Big Data comme Kafka, Spark et Docker peuvent travailler ensemble pour résoudre un problème concret, comme la surveillance industrielle ou domestique.

Voici comment j'ai procédé :

1. Simulation des capteurs : J'ai créé un script Python pour générer des données de température aléatoires, comme si elles venaient de vrais capteurs.
2. Transport des données : J'ai utilisé Kafka pour envoyer ces données à Spark, comme un « tuyau » géant qui assure que rien ne se perd.
3. Traitement en temps réel : Avec Spark, j'ai analysé les données pour repérer les températures dangereuses (>80°C) et calculer des moyennes toutes les 5 minutes.
4. Stockage et visualisation : Les résultats sont sauvegardés dans HDFS (un système de stockage fiable) et affichés sur une interface web que j'ai codée avec Flask.

Technologies clés :

◇ Docker : Pour faire tourner tous les services (Kafka, Spark, etc.) dans des « boîtes » isolées, sans conflits.

◇ Spark Streaming : Le cerveau du système, qui traite les données à la volée.

◇ Flask : Une interface simple pour voir les alertes en direct, comme un tableau de bord.

Résultats :

⋆ Le système traite plusieurs messages par seconde avec une latence élevée – assez rapide pour une application IoT !

⋆ L'interface affiche les alertes et les tendances en temps réel, ce qui permet une réaction immédiate.

Et après ? Je veux améliorer ce projet en :

◇ Ajoutant des capteurs réels (comme le DHT22) pour remplacer le simulateur.

◇ Créant un tableau de bord plus riche avec Grafana pour des graphiques dynamiques.

◇ Optimisant Spark pour réduire encore la latence.

Ce projet m'a appris à combiner plusieurs technologies complexes pour créer une solution cohérente. C'est un bon point de départ pour des applications industrielles ou même domestiques, comme surveiller une serre ou un data center !

Table des matières

1- Introduction

Le traitement des données en temps réel est une composante clé des systèmes modernes dans des domaines tels que l'Internet des Objets (IoT), la finance, la logistique et la surveillance industrielle. Ce projet vise à construire un pipeline de traitement des données en temps réel pour surveiller des capteurs de température, détecter des anomalies et générer des alertes. L'objectif est de démontrer l'interopérabilité entre plusieurs outils modernes tels que Kafka, Spark Streaming, Hadoop HDFS et Docker.

Ce rapport détaille les aspects techniques de la mise en œuvre, depuis la conception jusqu'au déploiement, en passant par les défis rencontrés et les résultats obtenus.

1.1. Objectifs du Projet

Les objectifs spécifiques de ce projet sont :

1. **Collecter des données de capteurs simulés** : Générer des données de température de manière aléatoire grâce à un simulateur Python.
2. **Traiter les données en temps réel** : Identifier des anomalies de température et calculer des moyennes par fenêtre de temps.
3. **Générer des alertes** : Produire des alertes en fonction des seuils de température critiques.
4. **Stocker les données et les alertes** : Organiser les données et alertes dans un système distribué (HDFS) pour une éventuelle analyse historique.
5. **Proposer une interface utilisateur** : Développer une interface web en Flask pour visualiser les données et alertes en temps réel.

2- Architecture Globale

L'architecture du pipeline est modulaire et suit un flux de traitement défini comme suit :

1. **Génération de données** : Un simulateur produit des données de température et les publie dans Kafka.
2. **Ingestion des données** : Kafka agit comme un broker pour transporter les données vers les différents consommateurs.
3. **Traitement des données** : Spark consomme les données de Kafka pour identifier des alertes instantanées et des tendances moyennes sur des fenêtres de temps.
4. **Stockage des résultats** : HDFS est utilisé pour stocker à la fois les données brutes et les alertes générées.
5. **Visualisation** : Flask fournit une interface utilisateur pour afficher les données et alertes.

Diagramme d'Architecture

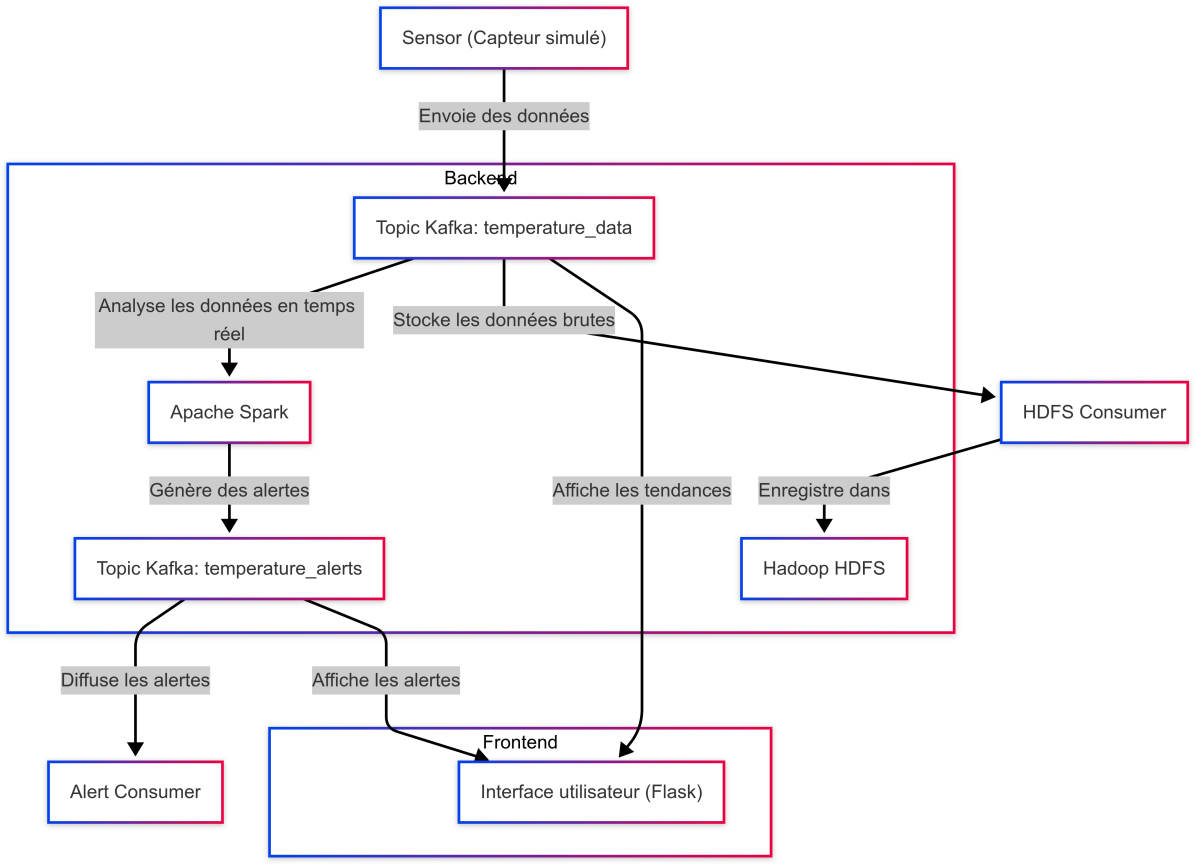

FIGURE 2.1 – Connexion à l'administration WordPress

3- Technologies Utilisées

Chaque technologie est choisie pour répondre à des besoins spécifiques du pipeline.

3.1. Docker

Rôle : Conteneurisation des services pour un déploiement simplifié.
Avantages :

- ◇ Isolation des services.
- ◇ Portabilité entre les environnements.
- ◇ Facilité de mise en œuvre avec Docker Compose.

3.2. Apache Kafka

Rôle : Gestion des flux de données en tant que système de messagerie distribuée.
Caractéristiques :

- ◇ Rétention configurable pour les topics.
- ◇ Haute disponibilité grâce à la réplication (non utilisée ici, car déploiement en local).
- ◇ Topics utilisés :
  - — `temperature_data` : Stocke les données brutes des capteurs.
  - — `temperature_alerts` : Stocke les alertes générées.

3.3. Apache Spark Streaming

Rôle : Traitement en temps réel des données.
Fonctionnalités :

- ◇ Consommation de données depuis Kafka.
- ◇ Génération d'alertes instantanées et sur des fenêtres de temps.
- ◇ Support des Watermarks pour gérer les données en retard.

3.4. Hadoop (HDFS)

Rôle : Stockage distribué des données pour une utilisation historique.
Avantages :

- ◇ Haute capacité de stockage.
- ◇ Organisation hiérarchique des fichiers.

3.5. Flask

Rôle : Interface utilisateur pour visualiser les données et alertes.
Fonctionnalités :

- ◇ API REST pour récupérer les données et alertes.
- ◇ Tableau de bord pour afficher les informations en temps réel.

4- Détails des Composants

4.1. Kafka et Zookeeper

Kafka est utilisé comme système de messagerie pour transporter les données des capteurs et les alertes entre les différents composants.

Configuration Importante :

- ◇ **Rétention des messages** : Les données dans le topic `temperature_data` sont conservées pendant 1 heure (`retention.ms=3600000`).
- ◇ **Partitions** : Chaque topic a une seule partition (limite de déploiement local).

4.2. HDFS (Hadoop Distributed File System)

Structure de Stockage :

```
/temperature_data/sensor_id/timestamp.json
/alerts/instant/sensor_id/timestamp.json
/alerts/avg_temp/sensor_id/window_start.json
```

4.3. Spark Streaming

Spark consomme les données de Kafka, applique des transformations pour extraire des alertes et publie les résultats dans Kafka et HDFS.

Exemples de Traitement :

- ◇ **Alertes Instantanées** : Détecte les températures critiques (>90°C).
- ◇ **Alertes Agrégées** : Moyenne des températures sur des fenêtres de 5 minutes.

4.4. Flask (Interface Utilisateur)

L'interface utilisateur permet de visualiser :

- ◇ Données brutes des capteurs.
- ◇ Alertes triées par type et gravité.

4.5. Producteur de Capteurs (Simulateur)

Le simulateur génère des données toutes les secondes avec le format suivant :

```
{
  "timestamp": 1651234567,
  "sensor_id": "sensor1",
  "temperature": 78.5
}
```

5- Déploiement avec Docker

Docker est utilisé dans ce projet pour faciliter le déploiement, la configuration et l'exécution de l'ensemble des composants du pipeline de traitement de données en temps réel. Grâce à Docker, chaque service (Kafka, Spark, HDFS, Flask, etc.) fonctionne dans un conteneur isolé, ce qui garantit une portabilité maximale et une simplicité de gestion, indépendamment du système d'exploitation hôte.

Le fichier **docker-compose.yml** joue un rôle central dans ce processus. Il permet de définir et de lancer plusieurs conteneurs en une seule commande. Ce fichier décrit la configuration de chaque service, leurs dépendances, les volumes de données, les ports exposés, et le réseau de communication entre les conteneurs.

- **Volumes** : Ils sont utilisés pour garantir la persistance des données, même si un conteneur est arrêté ou supprimé. Par exemple, les données stockées dans HDFS doivent être conservées de manière fiable ; les volumes permettent de les stocker en dehors du cycle de vie des conteneurs.
- **Réseaux** : Un réseau Docker personnalisé, appelé **kafka-network**, est défini afin de permettre aux différents conteneurs de communiquer entre eux par leurs noms de service. Ce réseau assure une isolation des communications internes et facilite le routage entre Kafka, Spark, Flask, etc., sans exposer inutilement les ports à l'extérieur.

En résumé, Docker et `docker-compose.yml` permettent d'obtenir une architecture reproductible, modulaire et facilement déployable, ce qui est particulièrement utile lors du développement collaboratif ou du déploiement sur de nouveaux environnements.

6- Fonctionnement du Pipeline

6.1. Flux de Données

Le pipeline suit quatre étapes principales pour traiter et afficher les données :

1. **Producteur (Capteur)** : - Les données sont générées par un simulateur de capteurs, incluant l'identifiant du capteur, la température relevée et un horodatage. - Ces données sont publiées dans le topic Kafka **temperature_data** pour être consommées par d'autres services.
2. **Spark (Traitement en Temps Réel)** : - Spark consomme les messages depuis Kafka et applique des transformations : - Classification des températures (`CRITICAL`, `HIGH`, `NORMAL`). - Agrégation sur des fenêtres de 5 minutes pour détecter des tendances. - Les alertes instantanées et agrégées sont publiées dans le topic Kafka **temperature_alerts**.
3. **HDFS (Stockage)** : - Les données brutes et les alertes générées sont stockées dans HDFS pour analyse et conservation à long terme. - Organisation structurée des fichiers pour un accès facile (`/temperature_data` et `/alerts`).
4. **Flask (Visualisation)** : - Flask consomme les données et alertes depuis Kafka pour les afficher en temps réel. - L'interface montre les relevés récents et les alertes classées par gravité et capteur.

7- Résultats et Analyse

Cette section présente les résultats obtenus à partir des tests effectués sur le pipeline, en mettant l'accent sur les performances et les observations clés. Elle inclut également une capture d'écran de l'interface utilisateur développée dans ce projet.

7.1. Performances

Le système a été évalué en termes de latence et de débit dans un environnement local Docker. Voici les principaux résultats obtenus :

- **Latence** : La latence moyenne est d'environ ∼500 ms entre la génération d'une donnée par le simulateur et son affichage dans l'interface.
- **Débit** :
  - Le système gère un débit d'environ ∼1000 messages/s lors de tests de charge.
  - Ce débit est suffisant pour des applications IoT à petite ou moyenne échelle.
  - Les performances de Kafka et Spark ont montré une bonne résilience même sous charge.

7.2. Visualisation des Résultats

La capture ci-dessous montre l'interface utilisateur développée avec Flask. inclut :

- Une section pour les relevés de température récents.
- Une liste des alertes triées par gravité et temps.

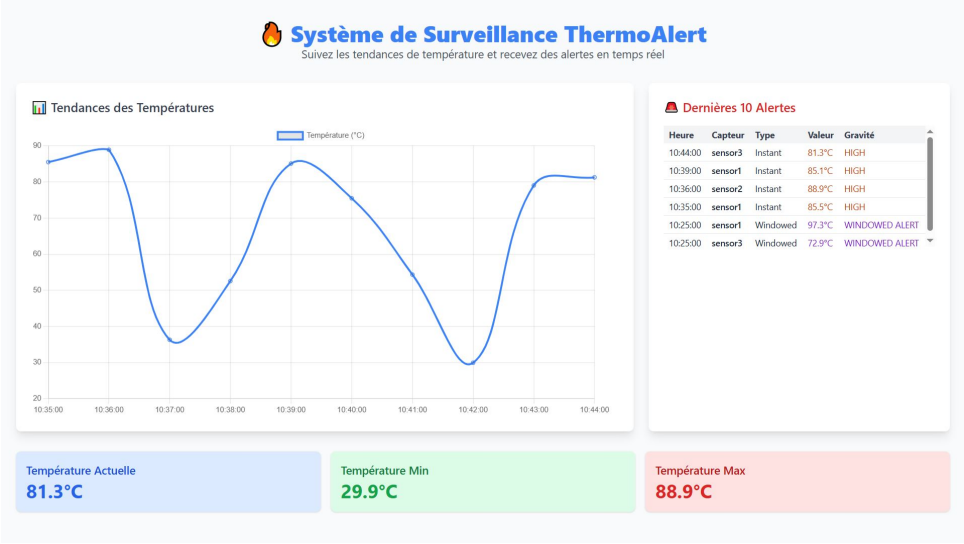

FIGURE 7.1 – Interface Utilisateur pour la Visualisation des Données et Alertes en Temps Réel

8- Améliorations Futures

8.1. Optimisation de Spark

Actuellement, Spark traite les données en temps réel avec des performances satisfaisantes. Cependant, plusieurs pistes d'optimisation peuvent être envisagées pour améliorer encore davantage l'efficacité du système :

- **Augmentation des partitions Kafka** : En augmentant le nombre de partitions des topics Kafka, on peut permettre un traitement parallèle plus efficace, tirant parti des ressources disponibles sur le cluster Spark.
- **Réduction de la latence** : En ajustant certains paramètres de configuration de Spark, comme l'intervalle de traitement (*batch interval*), il est possible de réduire la latence globale du pipeline.

8.2. Intégration d'un Tableau de Bord

Même si l'application Flask actuelle fournit une visualisation fonctionnelle des données, elle peut être enrichie par des outils spécialisés :

- **Utilisation de Grafana** : Cet outil de visualisation permettrait d'afficher des graphiques dynamiques et interactifs, facilitant l'analyse des données en temps réel.
- **Monitoring des performances** : En couplant Grafana à Prometheus, il devient possible de surveiller les indicateurs clés du pipeline (latence, débit, erreurs, etc.) en continu.

8.3. Utilisation de Capteurs Réels

Pour adapter le pipeline à un environnement réel, l'intégration de capteurs physiques est une évolution naturelle :

- **Remplacement du simulateur** : Au lieu de générer artificiellement les données, on peut utiliser des capteurs physiques tels que le *DHT22* pour mesurer la température ou l'humidité en conditions réelles.
- **Adoption d'un protocole IoT** : L'utilisation du protocole *MQTT* permettrait de publier les données directement depuis les capteurs vers Kafka, assurant une communication légère et rapide.

Ces améliorations visent à rendre le pipeline plus performant, plus intuitif à surveiller et mieux adapté aux cas d'usage concrets du monde réel.

9- Conclusion

Ce projet a mis en œuvre un pipeline de traitement des données en temps réel en s'appuyant sur des technologies modernes telles que Docker, Kafka, Spark Streaming et HDFS. Ces outils, combinés de manière modulaire, ont permis de construire un système efficace et extensible.

Les résultats obtenus démontrent la robustesse et la flexibilité du pipeline :

- ◇ Les données générées ont été collectées, traitées et stockées avec une latence moyenne de 500 ms, ce qui est satisfaisant pour des applications en temps réel.
- ◇ L'architecture modulaire facilite l'intégration de nouvelles fonctionnalités ou de nouveaux services, comme l'ajout de tableaux de bord avancés ou de capteurs physiques.
- ◇ Le stockage structuré des données dans HDFS garantit leur accessibilité pour des analyses futures.

Des pistes d'amélioration ont été identifiées pour rendre le système encore plus performant et adapté à des cas d'utilisation réels, notamment en optimisant le traitement Spark, en intégrant des tableaux de bord comme Grafana, et en remplaçant le simulateur par des capteurs physiques.

En conclusion, ce projet démontre la puissance des technologies modernes pour construire un pipeline robuste et modulable, et constitue une base solide pour des applications IoT, industrielles ou analytiques. Sa scalabilité et sa flexibilité en font une solution prometteuse pour des scénarios réels nécessitant un traitement des données en temps réel.

10- Annexes

10.1. Commandes Utiles

Voici les principales commandes utilisées pour gérer le pipeline via Docker et Docker Compose :

- `docker-compose up` : Lancer tous les services définis dans le fichier **docker-compose.yml**.
- `docker-compose up -d zookeeper` : Démarrer uniquement le service Zookeeper en mode détaché.
- **docker-compose up -d kafka** : Démarrer le service Kafka une fois Zookeeper opérationnel (attendre environ 30 secondes).
- **docker-compose up -d spark hadoop datanode** : Lancer Spark et les services Hadoop nécessaires.
- **docker-compose up -d sensor interface** : Démarrer le simulateur de capteurs et l'interface Flask.
- **docker logs -f <nom_du_service>** : Suivre les logs d'un service spécifique pour vérifier son bon fonctionnement (exemple : `spark-consumer`).
- **docker-compose down** : Arrêter et supprimer tous les conteneurs associés au projet.
- **docker system prune -a –volumes** : Nettoyer les conteneurs, images et volumes inutilisés pour libérer de l'espace disque.

Ces commandes permettent de gérer efficacement les services du pipeline, d'effectuer des tests et de résoudre d'éventuels problèmes.

10.2. Lien vers le Code Source

Vous pouvez accéder au code source complet de ce projet sur GitHub : ThermoAlert-RealTime-Temperature-Monitoring

10.3. Références

Voici quelques-unes des ressources utilisées pour ce projet :

- Documentation Apache Kafka
- Documentation Apache Spark
- Documentation Officielle Docker