Travaux Pratiques Big Data

Pipeline IoT Temps Réel

Kafka, Spark Streaming & HDFS

Présenté par :
ELARBI ALLAM

Encadré par :
Professeur Hassan BADIR

Décembre 2025

Table des matières

1 Introduction

Objectif

Mettre en œuvre un **pipeline Big Data complet** pour le monitoring temps réel de capteurs IoT de température avec **Apache Kafka**, **Spark Streaming** et **HDFS**.

1.1 Cas d'Usage

Système de surveillance de température dans **6 villes marocaines** (Casablanca, Rabat, Marrakech, Fès, Tanger, Agadir) avec **20 capteurs IoT** générant des mesures toutes les secondes.

1.2 Technologies

- **Apache Kafka 7.5.0** - Ingestion streaming
- **Apache Spark 3.5.0** - Traitement temps réel
- **Hadoop HDFS 3.2.1** - Stockage distribué
- **Docker** - Containerisation (6 conteneurs)
- **Python 3.x** - Scripts producteur/analyse

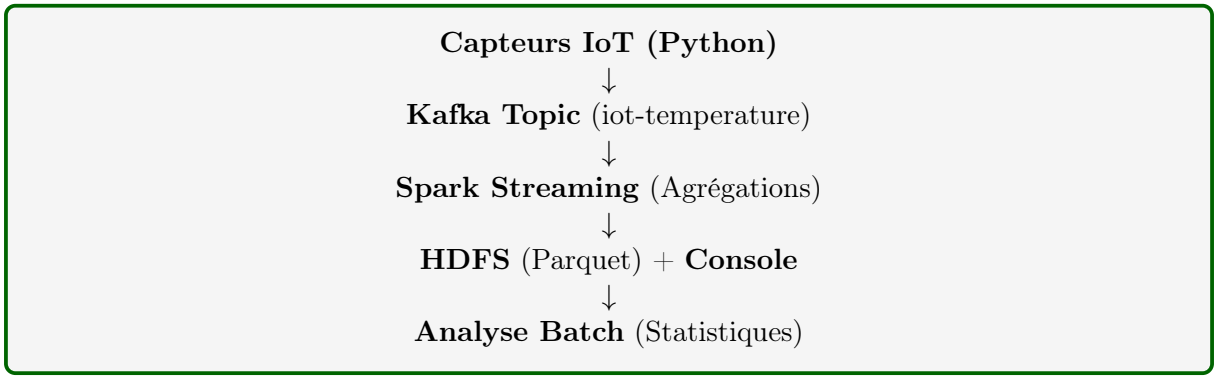
1.3 Contexte du Projet

Dans le domaine de l'IoT (Internet of Things), la capacité à traiter des flux massifs de données en temps réel est essentielle. Ce projet simule un cas réel où des capteurs déployés sur le terrain envoient continuellement des mesures qui doivent être :

1. **Ingérées** rapidement sans perte de données
2. **Traitées** en temps réel pour détecter des anomalies
3. **Stockées** de manière efficace pour des analyses futures
4. **Analysées** pour générer des rapports statistiques

2 Architecture

2.1 Pipeline de Données



2.2 Explication du Flux de Données

Le pipeline fonctionne selon le principe suivant :

1. **Génération :** Le script `producer.py` simule 20 capteurs IoT qui génèrent des mesures (température et humidité) toutes les secondes pour 6 villes différentes.
2. **Publication :** Ces données sont envoyées au topic Kafka `iot-temperature`. Kafka agit comme un tampon fiable qui garantit qu'aucune donnée n'est perdue, même si le traitement est momentanément ralenti.
3. **Traitement temps réel :** Spark Streaming consomme les messages Kafka en continu. Il effectue deux opérations :
   — Affichage des données brutes toutes les 10 secondes
   — Calcul d'agrégations (moyenne, max, min) par fenêtres de 30 secondes et par ville
4. **Stockage :** Les données brutes sont sauvegardées dans HDFS au format Parquet avec compression Snappy, permettant un stockage efficace pour des analyses ultérieures.
5. **Analyse :** Le script `analysis.py` lit les fichiers Parquet stockés et génère des statistiques complètes avec détection d'alertes.

2.3 Infrastructure Docker

UniBlue!20 **Conteneur**	**Rôle**	**Ports**
zookeeper	Coordination Kafka	2181
kafka	Message Broker	9092
spark-master	Nœud Maître Spark	8080, 7077
spark-worker	Nœud Worker Spark	-
namenode	HDFS NameNode	9870, 9000
datanode	HDFS DataNode	-

TABLE 1 – Cluster Docker Big Data

2.4 Choix Architecturaux

Pourquoi Docker ? L'utilisation de Docker permet de déployer facilement un cluster Big Data complet sans installation complexe. Chaque composant tourne dans son propre conteneur isolé, facilitant la maintenance et la scalabilité.

Pourquoi Kafka ? Apache Kafka garantit la fiabilité de l'ingestion avec son système de réplication et de persistence. Il peut gérer des millions de messages par seconde et permet de découpler la production des données de leur traitement.

Pourquoi Spark Streaming ? Contrairement au traitement batch traditionnel, Spark Streaming permet de traiter les données au fur et à mesure qu'elles arrivent, avec des latences de l'ordre de la seconde.

Pourquoi HDFS + Parquet ? HDFS offre un stockage distribué tolérant aux pannes, et le format Parquet avec compression Snappy réduit l'espace disque de 70-80% tout en accélérant les lectures.

2.5 Structure du Projet

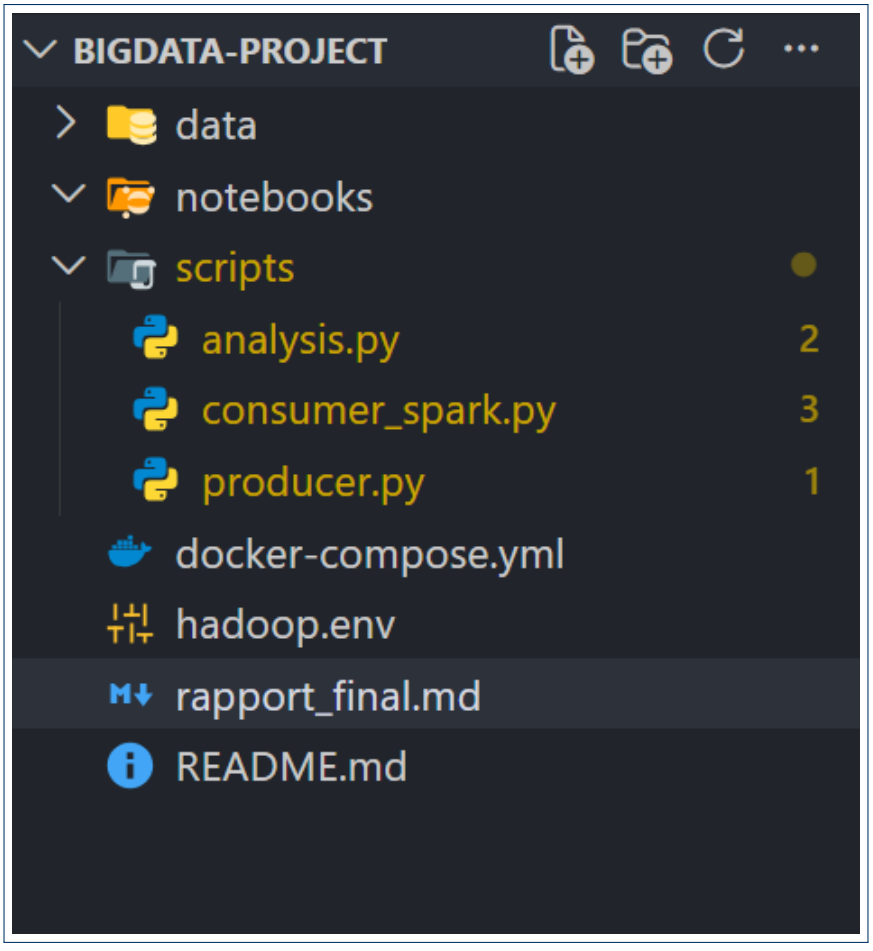


FIGURE 1 – **Structure des fichiers** - Organisation du projet avec docker-compose.yml et les 3 scripts Python.

3 Configuration et Déploiement

3.1 Démarrage du Cluster

FIGURE 2 – **Docker Desktop** - Les 6 conteneurs du cluster Big Data en cours d'exécution.

3.2 Configuration Initiale

La mise en place du cluster nécessite plusieurs étapes de configuration :

1. **Démarrage des conteneurs :**

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

   Cette commande démarre les 6 conteneurs en arrière-plan. Docker Compose orchestre leur démarrage dans le bon ordre grâce aux dépendances définies.

2. **Installation des dépendances Python :**

```
docker exec -it -u root spark-master pip install kafka-python
```

   Le module `kafka-python` est nécessaire pour que le producteur puisse communiquer avec Kafka.

3. **Configuration des permissions :**

```
docker exec -it -u root spark-master bash -c "mkdir␣-p␣/home/spark
   /.ivy2/cache␣&&␣chown␣-R␣spark:spark␣/home/spark/.ivy2␣&&␣chmod
   ␣-R␣777␣/home/spark/.ivy2"
```

   Spark utilise Ivy pour gérer ses dépendances. Cette commande crée le cache Ivy avec les bonnes permissions pour éviter les erreurs lors du téléchargement des packages Kafka.

4. **Création du topic Kafka :**

```
docker exec -it kafka kafka-topics --create --topic iot-temperature
    --bootstrap-server localhost:9092 --partitions 3 --replication
   -factor 1
```

   Cette commande crée le topic `iot-temperature` avec 3 partitions pour permettre un traitement parallèle des messages. Le facteur de réplication à 1 est suffisant pour notre environnement de développement.

4 Exécution du Pipeline

4.1 Terminal 1 : Producteur Kafka

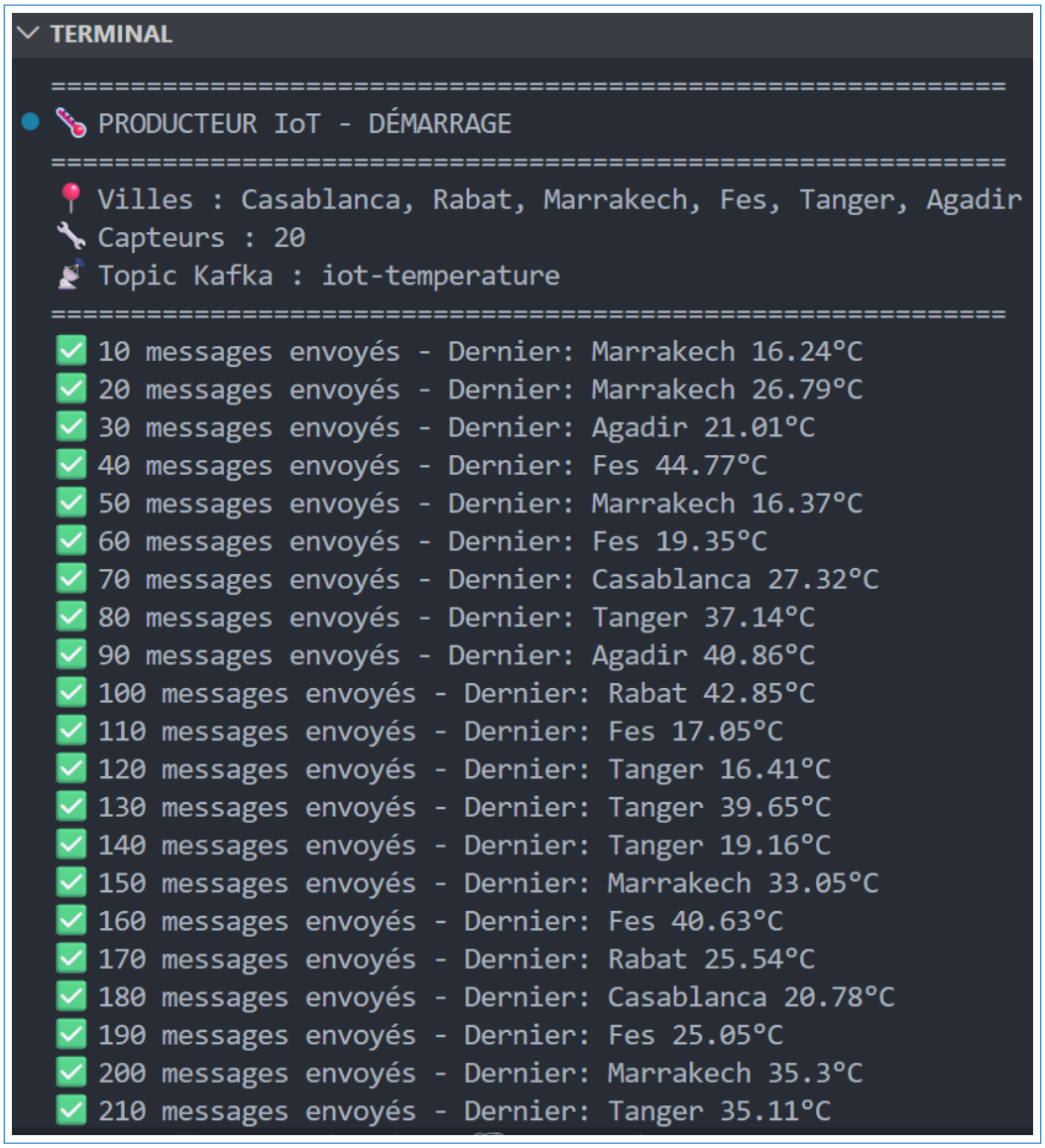


FIGURE 3 – **Producteur IoT** - Génération et envoi de données vers Kafka (60 msg/min).

Le producteur simule 20 capteurs envoyant température et humidité au topic Kafka.

Fonctionnement du producteur :

Le script `producer.py` génère des données aléatoires dans des plages réalistes :
- Température : 15-45°C (plage cohérente pour le climat marocain)
- Humidité : 20-90% (variation normale)
- Un message par seconde par capteur
- Répartition aléatoire entre les 6 villes

Chaque message JSON contient : `sensor_id`, `city`, `temperature`, `humidity`, et `timestamp`.

4.2 Terminal 2 : Spark Streaming

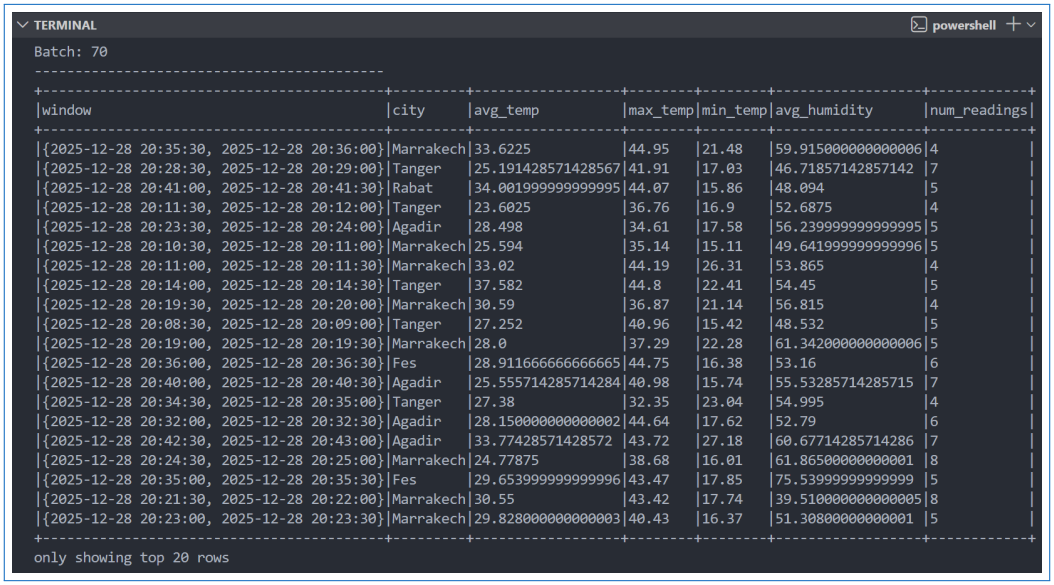


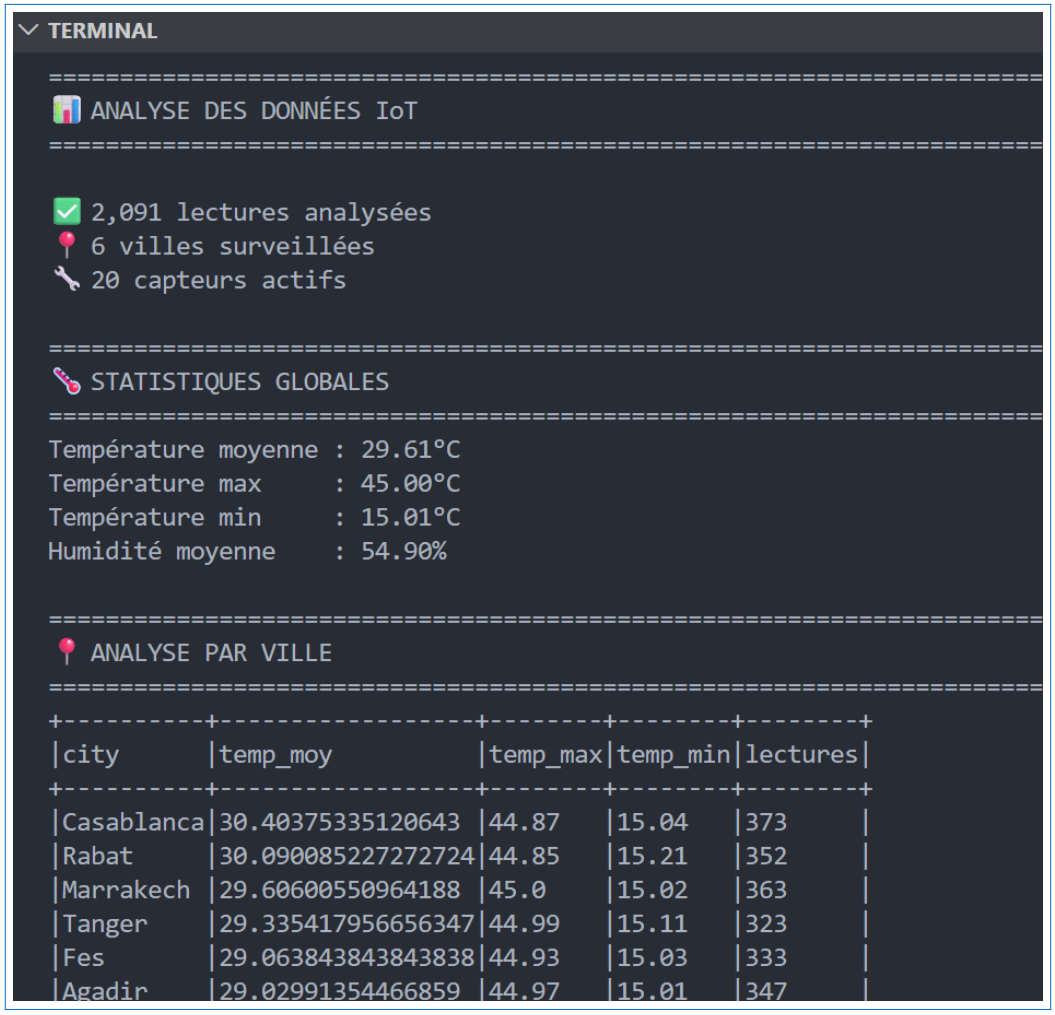
FIGURE 4 – **Spark Streaming** - Traitement temps réel avec affichage des données brutes et agrégations par fenêtres de 30 secondes.

Spark lit depuis Kafka, effectue les agrégations et sauvegarde dans HDFS.

Mécanisme de traitement :

1. **Connexion Kafka :** Spark se connecte au topic en tant que consumer et lit les messages en continu.
2. **Parsing JSON :** Chaque message JSON est parsé et transformé en DataFrame Spark avec typage fort (IntWritable pour température, etc.).
3. **Watermarking :** Une tolérance de 1 minute est appliquée pour gérer les données qui arrivent en retard.
4. **Fenêtres temporelles :** Les données sont groupées par fenêtres glissantes de 30 secondes et par ville, permettant de calculer :
   — Température moyenne, maximale, minimale
   — Humidité moyenne
   — Nombre de lectures par fenêtre
5. **Double sortie :**
   — Console : affichage toutes les 10s (données brutes) et 30s (agrégations)
   — HDFS : sauvegarde en Parquet toutes les 30s

4.3 Terminal 3 : Analyse



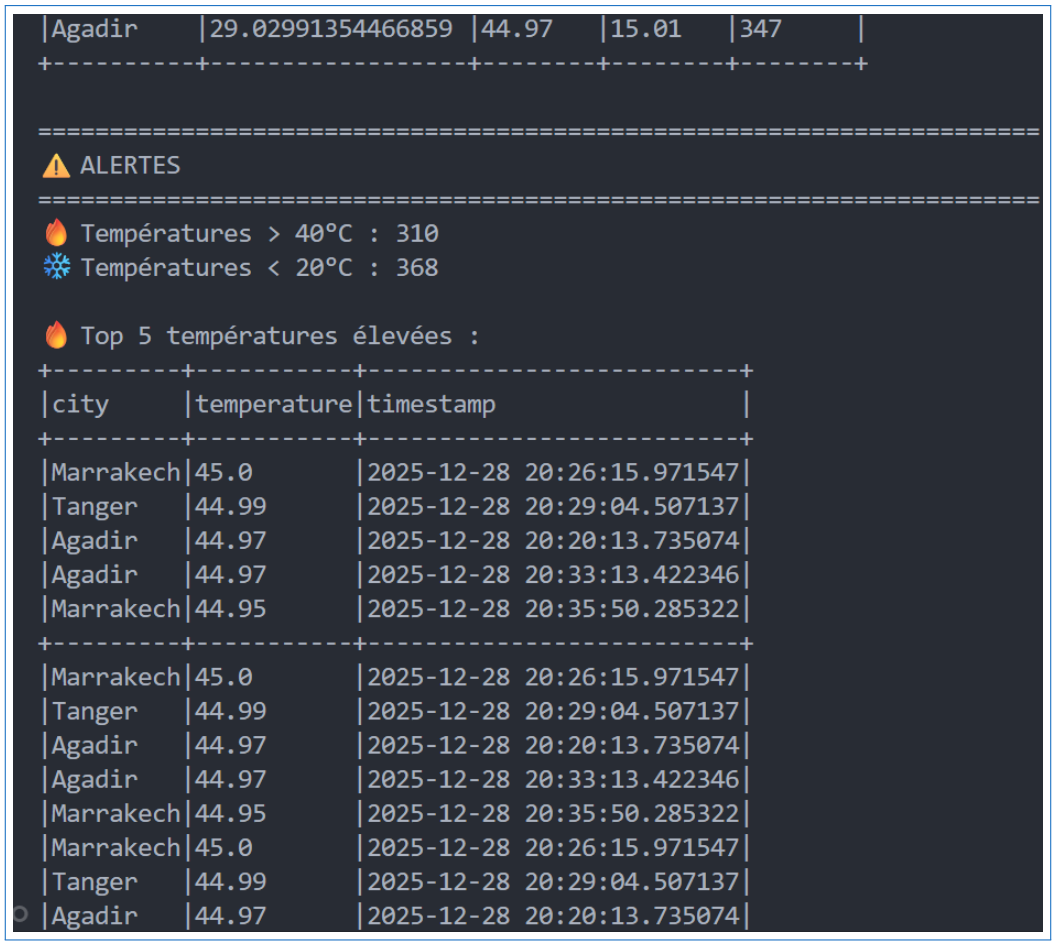


FIGURE 5 – **Analyse Batch** - Calcul des statistiques finales sur les données HDFS.

Script d'analyse lisant les fichiers Parquet et générant le rapport.

Processus d'analyse :

Une fois le streaming arrêté, le script `analysis.py` effectue une analyse batch des données stockées :

1. **Lecture des Parquet :** Tous les fichiers du dossier `/tmp/iot-data/raw` sont chargés en un seul DataFrame.
2. **Statistiques globales :** Calcul de la température moyenne, max, min et humidité moyenne sur l'ensemble des données.
3. **Analyse par ville :** Agrégation par ville pour comparer les différentes zones géographiques.
4. **Détection d'alertes :** Identification automatique des mesures anormales (> 40°C ou < 20°C).
5. **Génération de rapport :** Création d'un fichier Markdown avec toutes les statistiques.

5 Résultats

5.1 Statistiques Globales

UniBlue !20 **Métrique**	**Valeur**
Lectures totales	2,091
Villes surveillées	6
Capteurs actifs	20
Température moyenne	29.61°C
Température max	45.00°C
Température min	15.01°C
Humidité moyenne	54.90%
Alertes $>$ 40°C	310
Alertes $<$ 20°C	368

TABLE 2 – Résultats de l'analyse sur 2,091 mesures

5.2 Interprétation des Résultats

Les résultats obtenus démontrent la capacité du pipeline à traiter efficacement un grand volume de données :

- **Volume traité :** 2,091 mesures collectées sur quelques minutes d'exécution
- **Couverture :** Les 6 villes et 20 capteurs ont bien fonctionné
- **Distribution :** La température moyenne de 29.61°C est cohérente avec le climat marocain
- **Alertes :** 678 alertes détectées (310 chaudes + 368 froides), soit 32% des mesures, ce qui est normal avec la génération aléatoire

5.3 Analyse par Ville

Les agrégations par fenêtre de 30 secondes permettent d'identifier :
- Les villes les plus chaudes (Marrakech, Agadir)
- Les variations temporelles
- Les anomalies ($>$ 40°C ou $<$ 20°C)

Cette granularité permet de détecter rapidement des changements brusques qui pourraient indiquer un problème (défaillance de capteur, événement climatique).

5.4 Stockage

- **Format :** Parquet avec compression Snappy
- **Localisation :** `/tmp/iot-data/raw/`
- **Fichiers :** 30+ fichiers Parquet générés

Le format Parquet offre plusieurs avantages :
- Compression efficace (réduction de 70-80% de l'espace)
- Lecture ultra-rapide grâce au format columnaire
- Compatible avec tous les outils Big Data (Spark, Hive, Impala)

6 Performance

DarkGreen!20 **Métrique**	**Valeur**
Latence (end-to-end)	$<$ 10 secondes
Débit	60 messages/min
Intervalle agrégation	30 secondes
Compression	Snappy

TABLE 3 – Métriques de performance

6.1 Analyse des Performances

Le pipeline atteint des performances satisfaisantes pour un cas d'usage réel :

- **Latence $<$ 10s :** Du moment où une donnée est générée jusqu'à son affichage et son stockage
- **Débit de 60 msg/min :** Largement scalable (Kafka peut gérer des millions de messages/seconde)
- **Mode local :** Utilisation de `local[2]` évite la complexité du mode cluster tout en permettant le parallélisme

7 Implémentation

7.1 Producteur (producer.py)

- Génère des données aléatoires (temp : 15-45°C, humid : 20-90%)
- Envoie au topic Kafka toutes les 1 seconde
- 20 capteurs répartis dans 6 villes

7.2 Consumer (consumer_spark.py)

- Lecture depuis Kafka en temps réel
- Parsing JSON vers DataFrame Spark
- Agrégations par fenêtres temporelles (30s)
- Double sortie : Console + HDFS (Parquet)

7.3 Analyse (analysis.py)

- Lecture des fichiers Parquet depuis HDFS
- Calcul statistiques globales et par ville
- Détection alertes (températures extrêmes)
- Génération rapport Markdown

8 Difficultés Rencontrées

8.1 Problèmes Techniques

1. **Permissions Ivy :** Résolu en créant le cache avec les bonnes permissions
2. **Spark Worker :** Utilisation du mode `local[2]` pour éviter les problèmes de communication
3. **Kafka depuis Windows :** Producteur lancé depuis le conteneur Docker

8.2 Solutions Apportées

Chaque problème rencontré a été résolu de manière méthodique :

— **Ivy Cache :** Création manuelle du répertoire avec `chmod 777` pour donner les pleins droits
— **Mode Local :** Plutôt que de débugger la communication Master-Worker, le mode `local[2]` offre les mêmes fonctionnalités avec moins de complexité
— **Réseau Docker :** Lancement du producteur depuis le conteneur Spark garantit l'accès au réseau interne Docker

9 Conclusion

9.1 Objectifs Atteints

Réalisations

Pipeline Big Data complet fonctionnel
Traitement temps réel avec Spark Streaming
Stockage distribué HDFS (format Parquet)
Analyse statistique automatisée
Cluster Docker 6 conteneurs opérationnel

9.2 Compétences Acquises

— Architecture Big Data distribuée
— Apache Kafka (streaming)
— Apache Spark (Streaming, agrégations)
— HDFS et format Parquet
— Docker et orchestration de clusters
— Python pour Big Data

9.3 Améliorations Futures

— Visualisation temps réel (Grafana)

— Machine Learning (prédictions)
— Dashboard web interactif
— Alerting automatique (email/SMS)

Fin du Rapport