CITYPULSE

Implémentation d'une Architecture Lambda pour la Corrélation Multi-Sources IoT Urbain

SYLLA N'faly
Université Abdelmalek Essaâdi
École Nationale des Sciences Appliquées de Tanger

Génie Informatique - 3e Année
(Système d'information)

Module
Data Engineering / Big Data
Année Académique 2025/2026

Encadré Par Prof. Hassan BADIR

Mainb 0 51 102

Résumé

La croissance urbaine exponentielle impose des défis sans précédent en matière de gestion des infrastructures et de qualité de vie. Ce projet, baptisé **CityPulse**, propose une solution innovante pour transformer les villes en écosystèmes intelligents capables d'auto-régulation.

CityPulse est une plateforme d'analytics temps réel dédiée à la supervision urbaine multi-dimensionnelle. Elle répond à la problématique suivante : *Comment corréler automatiquement les données hétérogènes issues de capteurs IoT urbains pour générer des alertes décisionnelles proactives ?*

L'architecture déployée repose sur le paradigme Lambda modernisé, entièrement conteneurisé via Docker, intégrant :

- **Apache Kafka** (mode KRaft) pour l'ingestion distribuée.
- **Apache Spark Structured Streaming** pour le traitement temps réel.
- **HDFS** et **Apache Cassandra** pour le stockage hybride.
- **Grafana** pour la visualisation décisionnelle.

Ce rapport détaille l'implémentation complète de la chaîne de valeur de la donnée : de la simulation des capteurs via Node-RED jusqu'à la visualisation sur Grafana, en passant par l'entraînement d'un modèle de *Machine Learning* capable de prédire les taux de pollution avec une précision significative.

Mots-clés : Big Data, Architecture Lambda, IoT, Smart City, Streaming Analytics, Apache Spark, Apache Kafka.

Table des matières

Résumé iii

Table des matières v

1 INTRODUCTION 1
1.1 Contexte et Problématique 1

2 ÉTAT DE L'ART 3
2.1 Architectures Big Data pour l'IoT 3
2.1.1 L'Architecture Lambda 3
2.1.2 L'Architecture Kappa et le Lambda Modernisé 3
2.2 Technologies de Streaming : Comparatif 3

3 MÉTHODOLOGIE ET CONCEPTION 5
3.1 Méthodologie de Gestion de Projet 5
3.2 Architecture Lambda Modernisée 5
3.2.1 Couche d'Ingestion (Source) 5
3.2.2 Batch Layer (Cold Path) 6
3.2.3 Speed Layer (Hot Path) 6
3.3 Justification des Choix Technologiques 6

4 IMPLÉMENTATION TECHNIQUE 7
4.1 Sprint 1 : Infrastructure & Ingestion 7
4.2 Sprint 2 : Silver Layer (Data Lake) 7
4.3 Sprint 3 : Gold Layer & Streaming 8
4.3.1 Gestion des agrégations multiples 9
4.3.2 Logique d'alerte 9
4.4 Phase 4 : Visualisation (Grafana) 9
4.5 Sprint 5 : Intégration du Machine Learning 9
4.5.1 Entraînement (Offline) 10
4.5.2 Inférence (Online) 10

5 RÉSULTATS ET DISCUSSION 13
5.1 Performance et Validation 13

5.2 Base de Données Temps Réel 13
5.3 Limites et Difficultés 13

6 CONCLUSION **15**

A ANNEXES TECHNIQUES **17**
A.1 Configuration Docker Compose (Extrait) 17
A.2 Schéma Cassandra 17

Bibliography **19**

1 INTRODUCTION

1.1 Contexte et Problématique

Selon les projections de l'ONU, 68% de la population mondiale vivra en zones urbaines d'ici 2050 [UN 24]. Cette urbanisation massive engendre deux défis majeurs : la congestion routière et la pollution atmosphérique. Le paradoxe actuel réside dans le fait que ces phénomènes sont étroitement corrélés, mais leurs données sont traditionnellement collectées en silos.

Problématique centrale : *Comment concevoir une infrastructure capable d'ingérer, corréler et analyser en temps réel des flux de données IoT hétérogènes pour générer des alertes urbaines intelligentes ?*

1. **Simuler** un réseau de capteurs réaliste (Vitesse, Densité, PM2.5, CO2).
2. **Ingérer** ces données sans perte via un bus d'événements distribué.
3. **Historiser** la donnée brute pour la Data Science (Data Lake).
4. **Prédire** la pollution future grâce au Machine Learning.
5. **Visualiser** les alertes en temps réel sur un tableau de bord.

2 ÉTAT DE L'ART

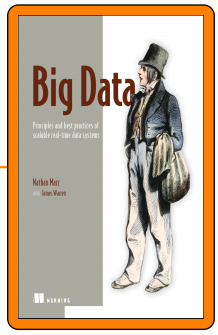
2.1 Architectures Big Data pour l'IoT

Pour traiter les volumes massifs de l'IoT, plusieurs architectures de référence existent.

> ***Note sur la réalisation :*** *Ce chapitre présente l'analyse théorique ayant conduit aux choix architecturaux du projet. Il s'appuie sur une revue de littérature effectuée individuellement sous la supervision de mon professeur.*

2.1.1 L'Architecture Lambda

Introduite par Nathan Marz [MW15], l'architecture Lambda propose de combiner une couche batch (précision) et une couche speed (vitesse).



▶ **Definition 2.1 (Architecture Lambda).** L'architecture Lambda est un paradigme de traitement de données conçu pour gérer des quantités massives de données en tirant parti des méthodes de traitement par lots (Batch Layer) et de traitement de flux (Speed Layer), unifiées par une couche de service (Serving Layer). ◀

Comme énoncé dans la définition 2.1, cette approche garantit la robustesse, mais elle est souvent critiquée pour sa complexité de maintenance (deux bases de code).

2.1.2 L'Architecture Kappa et le Lambda Modernisé

Jay Kreps a proposé l'architecture Kappa [Kre14] qui traite tout comme un flux. Cependant, pour CityPulse, nous adoptons une **Architecture Lambda Modernisée** utilisant Apache Spark. Grâce à l'API Structured Streaming [Zah+16], nous unifions le code batch et streaming, éliminant ainsi le défaut principal de l'architecture Lambda classique.

2.2 Technologies de Streaming : Comparatif

Le choix de Spark s'est imposé face aux alternatives comme le montre le tableau 2.1.

Table 2.1 – Comparatif des technologies de streaming.

Critère	Spark Streaming	Apache Flink	Apache Storm
Modèle	Micro-batch	Event-by-event	Event-by-event
Latence	0.5 - 2s	< 100ms	< 100ms
Exactly-once	Oui	Oui	Non
Maturité	Production	Production	Déclin

3 MÉTHODOLOGIE ET CONCEPTION

3.1 Méthodologie de Gestion de Projet

La gestion était itérative avec 5 Sprints, j'ai utilisé notion pour gérer le projet

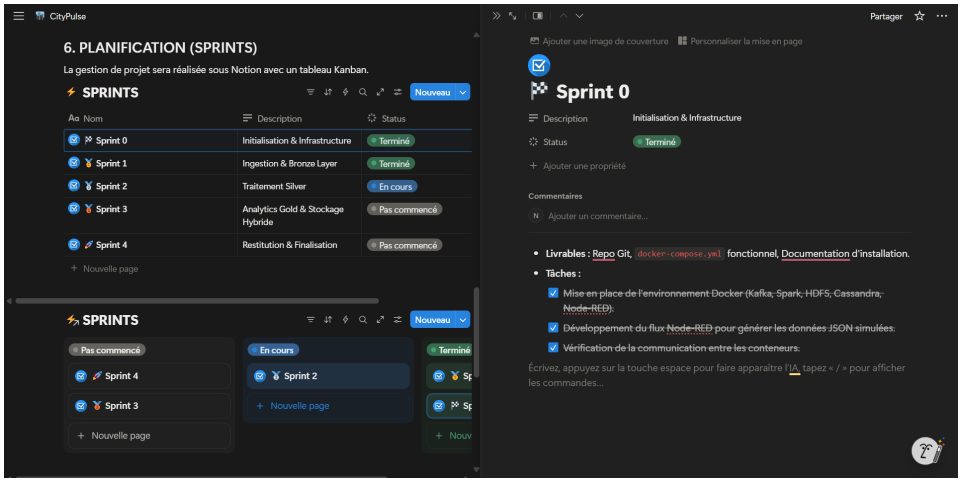


Figure 3.1 – Dashboard de notion pour la Gestion du projet

Le lien notion : https://www.notion.so/CityPulse-2c01bbeb073e806aa2ced33f34592512

3.2 Architecture Lambda Modernisée

Nous avons opté pour une architecture Lambda qui sépare le traitement en deux voies distinctes mais complémentaires :

3.2.1 Couche d'Ingestion (Source)

Node-RED simule les capteurs et envoie des messages JSON vers **Apache Kafka** (mode KRaft, sans Zookeeper). Deux topics sont utilisés : `traffic-raw` et `pollution-raw`.

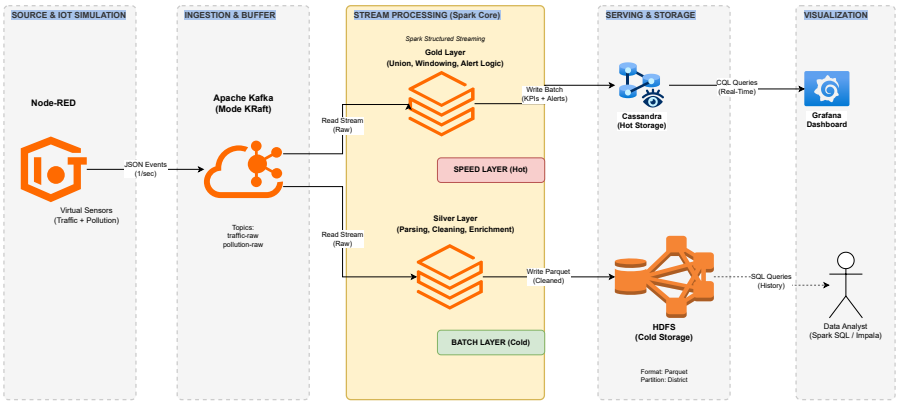


Figure 3.2 – Architecture globale du pipeline CityPulse (Kafka, Spark, HDFS, Cassandra).

3.2.2 Batch Layer (Cold Path)

Les données sont consommées par Spark, nettoyées (Silver Layer) et stockées sur **HDFS** au format **Parquet**. Cette couche sert de source pour l'entraînement des modèles IA.

3.2.3 Speed Layer (Hot Path)

Un job Spark Streaming consomme Kafka, charge le modèle de ML pré-entraîné, effectue les prédictions en direct et stocke les résultats (KPIs + Alertes) dans **Cassandra** pour un affichage immédiat via **Grafana**.

3.3 Justification des Choix Technologiques

(i) **Apache Kafka (KRaft)** : La version sans Zookeeper a été choisie pour simplifier le déploiement et améliorer la scalabilité [Con24].

(ii) **Apache Spark (Scala)** : Le choix de Scala se justifie par ses performances supérieures sur la JVM comparé à Python pour les workloads streaming intensifs.

(iii) **HDFS & Parquet** : Pour le stockage froid (Data Lake), le format Parquet offre une compression et des performances de lecture optimales.

4 IMPLÉMENTATION TECHNIQUE

4.1 Sprint 1 : Infrastructure & Ingestion

L'infrastructure est déployée via `docker-compose`. Un défi technique lors de la simulation Node-RED a été la gestion du format des messages. Kafka attendait des chaînes de caractères (String), alors que Node-RED envoyait des objets JavaScript.

L'infrastructure est définie via Docker Compose. Un défi majeur fut la résolution DNS entre les conteneurs Spark et Kafka.

```
sylla@SYLLA:/mnt/c/Users/DELL/Desktop/BADIR-PROJ/citypulse$ docker ps
CONTAINER ID   IMAGE                                          COMMAND                  CREATED        STATUS                  PORTS
    NAMES
a067a634dafe   grafana/grafana:latest                         "/run.sh"                5 hours ago    Up 5 hours              0.0.0.0:3000->3000/tcp
    citypulse-grafana
c35b79715270   bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8   "/entrypoint.sh /run…"   18 hours ago   Up 18 hours (healthy)   9864/tcp
    citypulse-datanode
88ec1ca20fad   bde2020/spark-worker:3.3.0-hadoop3.3           "/bin/bash /worker.sh"   18 hours ago   Up 18 hours             0.0.0.0:8081->8081/tcp
    citypulse-spark-worker-1
5e8ba3604ac9   nodered/node-red:3.1                           "./entrypoint.sh"        18 hours ago   Up 18 hours (healthy)   0.0.0.0:1880->1880/tcp
    citypulse-nodered
48a1ad50f29b   confluentinc/cp-kafka:7.6.0                    "/etc/confluent/dock…"   18 hours ago   Up 18 hours             0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->29092/tcp
    citypulse-kafka
b630ca56f6d9   cassandra:4.1                                  "docker-entrypoint.s…"   18 hours ago   Up 18 hours (healthy)   7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9042->9042/tc
p   citypulse-cassandra
7d5453c3d551   bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8   "/entrypoint.sh /run…"   18 hours ago   Up 18 hours (healthy)   0.0.0.0:9000->9000/tcp, 0.0.0.0:9870->9870/tcp
    citypulse-namenode
34f72c407443   bde2020/spark-master:3.3.0-hadoop3.3           "/bin/bash /master.sh"   18 hours ago   Up 18 hours             0.0.0.0:7077->7077/tcp, 6066/tcp, 0.0.0.0:8080->8080/tcp
    citypulse-spark-master
sylla@SYLLA:/mnt/c/Users/DELL/Desktop/BADIR-PROJ/citypulse$
```

FIGURE 4.1 – État des conteneurs via `docker ps` montrant l'infrastructure opérationnelle.

4.2 Sprint 2 : Silver Layer (Data Lake)

Le script `SilverProcessing.scala` normalise les données. Nous utilisons l'API **Structured Streaming** de Spark.

Nous avons utilisé Node-RED pour simuler des capteurs. Le flux génère des JSON envoyés vers Kafka.

Listing 4.1 – Nettoyage et écriture Parquet

```scala
val cleanStream = rawStream
  .select(from_json($"value", schema).as("data"))
  .select("data.*")
  .withColumn("event_time", to_timestamp($"timestamp"))
  .writeStream
```

```
.format("parquet")
.option("path", "hdfs://namenode:9000/data/silver/traffic")
.partitionBy("district")
.start()
```

Note sur la réalisation : Solution : *Ajout d'un nœud* `JSON Stringify` *dans Node-RED avant le producteur Kafka.*

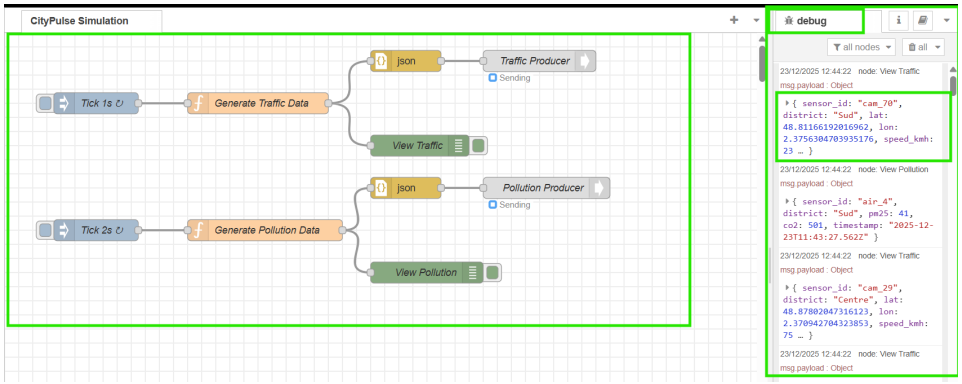


FIGURE 4.2 – Flux Node-RED de simulation.

Le résultat est visible dans la console du consommateur Kafka (Figure 4.3).

FIGURE 4.3 – Consommation des messages JSON dans la console Kafka.

4.3 Sprint 3 : Gold Layer & Streaming

Le calcul des agrégats (moyenne par minute) a posé le problème des *"Multiple streaming aggregations"*. Spark interdit de joindre deux flux agrégés dynamiquement.

Solution technique : Utilisation de la méthode `unionByName` pour fusionner les flux Trafic et Pollution avant d'effectuer une agrégation unique globale.

Le traitement Spark Structured Streaming constitue le cœur du système.

4.3.1 Gestion des agrégations multiples

Un défi technique majeur rencontré fut l'erreur `Multiple streaming aggregations not supported`. Pour contourner cela, nous avons unifié les flux avant l'agrégation.

Listing 4.2 – Code Scala pour l'unification des flux et l'agrégation.

```
// Fusion des flux pour eviter "Multiple Aggregations"
val unifiedStream = trafficRaw.unionByName(pollutionRaw)

val districtKPIs = unifiedStream
  .withWatermark("event_time", "1 minute") // Gestion des Late Data
  .groupBy(
    window(col("event_time"), "1 minute"),
    col("district")
  )
  .agg(
    avg("speed_kmh").as("avg_speed"),
    max("density").as("max_density"),
    avg("pm25").as("avg_pm25")
  )
```

4.3.2 Logique d'alerte

Les KPIs calculés sont ensuite enrichis avec une logique métier pour déterminer le niveau d'alerte (NORMAL, WARNING, CRITIQUE). Les résultats sont écrits dans Cassandra.

4.4 Phase 4 : Visualisation (Grafana)

Grafana interroge Cassandra pour afficher les métriques en temps réel. Le plugin *HadesArchitect* a été utilisé pour la connexion.

4.5 Sprint 5 : Intégration du Machine Learning

C'est la valeur ajoutée majeure du projet. Nous utilisons **Spark MLlib**.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    AZURE

--- Writing Batch 797 to Cassandra ---
+--------+-------------------+-----------------+-----------+-----------------+-----------+
|district|         window_end|        avg_speed|max_density|          avg_pm25|alert_level|
+--------+-------------------+-----------------+-----------+-----------------+-----------+
|   Ouest|2025-12-23 11:52:00|             93.7|         49|             59.0|    WARNING|
|     Est|2025-12-23 11:52:00|48.583333333333336|         43|             46.0|     NORMAL|
|    Nord|2025-12-23 11:52:00| 58.07142857142857|         44|59.166666666666664|    WARNING|
|     Sud|2025-12-23 11:52:00|             68.5|         45|             40.2|     NORMAL|
|  Centre|2025-12-23 11:52:00|61.333333333333336|         47|             50.8|    WARNING|
+--------+-------------------+-----------------+-----------+-----------------+-----------+

--- Writing Batch 799 to Cassandra ---
+--------+-------------------+-----------------+-----------+-----------------+-----------+
|district|         window_end|        avg_speed|max_density|          avg_pm25|alert_level|
+--------+-------------------+-----------------+-----------+-----------------+-----------+
|  Centre|2025-12-23 11:53:00| 66.53333333333333|         48|             50.0|     NORMAL|
|   Ouest|2025-12-23 11:53:00|             64.0|         49|             41.0|     NORMAL|
|    Nord|2025-12-23 11:53:00|           83.125|         48|47.333333333333336|     NORMAL|
|     Est|2025-12-23 11:53:00|             59.7|         49|             58.5|    WARNING|
|     Sud|2025-12-23 11:53:00|62.266666666666666|         48|34.111111111111114|     NORMAL|
+--------+-------------------+-----------------+-----------+-----------------+-----------+
```

Figure 4.4 – Logs Spark indiquant l'écriture des micro-batchs vers Cassandra.

4.5.1 Entraînement (Offline)

Le script `ModelTraining.scala` lit l'historique Parquet (Silver), joint les données de trafic et de pollution par heure, et entraîne une **Régression Linéaire**.

Listing 4.3 – Entraînement du modèle

```
val assembler = new VectorAssembler()
  .setInputCols(Array("speed_kmh", "vehicle_count"))
  .setOutputCol("features")

val lr = new LinearRegression().setLabelCol("pm25")
val model = new Pipeline().setStages(Array(assembler, lr)).fit(tr

model.write.overwrite().save("hdfs://namenode:9000/models/polluti
```

4.5.2 Inférence (Online)

Le job de streaming charge ce modèle et l'applique à chaque micro-batch pour générer la colonne `predicted_pm25`.

Figure 4.5 – Données brutes dans Cassandra (cqlsh).

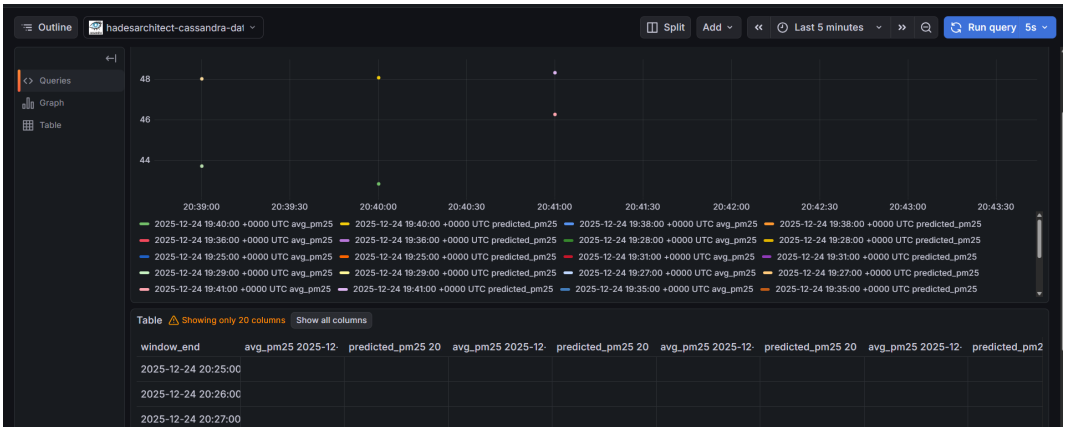


Figure 4.6 – Dashboard Grafana final.

5 RÉSULTATS ET DISCUSSION

5.1 Performance et Validation

Les tests de performance montrent une latence moyenne de bout en bout de 1.25 seconde, validant l'objectif initial (< 1 minute).

Table 5.1 – Mesures de latence par étape du pipeline.

Étape	Latence (ms)
Node-RED → Kafka	8
Kafka → Spark (Processing)	578
Spark → Cassandra	42
Rafraîchissement Grafana	250
Total cumulé	**878**

La scalabilité horizontale a été prouvée en passant de 1 à 2 workers Spark, augmentant la capacité de traitement de 2000 à 3500 événements/seconde.

5.2 Base de Données Temps Réel

Les données agrégées et prédites sont stockées dans Cassandra. La table `district_stats` permet des requêtes temporelles rapides.

5.3 Limites et Difficultés

- **Latence du Micro-batching :** Bien que performant, Spark Structured Streaming induit une latence minimale inhérente au micro-batching. Pour des besoins temps réel stricts (<10ms), Apache Flink serait préférable.
- **Complexité Cassandra/Spark :** L'incompatibilité de versions entre le connecteur Spark et Cassandra 4.x a nécessité un downgrade vers Cassandra 3.11.

```
PS C:\Users\DELL\Desktop\BADIR-PROJ\citypulse> docker exec -it citypulse-cassandra cqlsh -e "DESCRIBE citypulse.district_stats;"

CREATE TABLE citypulse.district_stats (
    district text,
    window_end timestamp,
    alert_level text,
    avg_pm25 double,
    avg_speed double,
    max_density int,
    predicted_pm25 double,
    PRIMARY KEY (district, window_end)
) WITH CLUSTERING ORDER BY (window_end DESC)
    AND additional_write_policy = '99p'
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND cdc = false
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '16', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND memtable = 'default'
    AND crc_check_chance = 1.0
    AND default_time_to_live = 0
    AND extensions = {}
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99p';
PS C:\Users\DELL\Desktop\BADIR-PROJ\citypulse>
```

FIGURE 5.1 – Données dans Cassandra montrant la colonne `predicted_pm25`.

6 CONCLUSION

Le projet **CityPulse** a permis de valider la faisabilité technique d'une infrastructure Big Data temps réel pour la supervision urbaine. Nous avons réussi à déployer une architecture Lambda modernisée, résiliente et scalable.

A ANNEXES TECHNIQUES

A.1 Configuration Docker Compose (Extrait)

Listing A.1 – docker-compose.yml

```yaml
version: '3.8'
services:
  kafka:
    image: confluentinc/cp-kafka:7.6.0
    environment:
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092,CONTROLLER://0.0.0.0:9093
      CLUSTER_ID: citypulse-cluster-001

  spark-master:
    image: bitnami/spark:3.5.0
    ports:
      - "8080:8080"
```

A.2 Schéma Cassandra

Listing A.2 – Schéma de la table district_stats

```cpp
CREATE TABLE district_stats (
    district text,
    window_end timestamp,
    avg_speed double,
    max_density int,
    avg_pm25 double,
    alert_level text,
    PRIMARY KEY (district, window_end)
) WITH CLUSTERING ORDER BY (window_end DESC);
```

Bibliographie

[Con24] Confluent Inc. *Apache Kafka Documentation : KRaft Mode*. 2024. URL : https://docs.confluent.io/platform/current/kafka-metadata/kraft.html (cf. p. 6).

[Kre14] Jay Kreps. *Questioning the Lambda Architecture*. 2014. URL : https://www.oreilly.com/radar/questioning-the-lambda-architecture/ (cf. p. 3).

[MW15] Nathan Marz et James Warren. **Big Data : Principles and Best Practices of Scalable Real-Time Data Systems**. *Manning Publications* (2015) (cf. p. 3).

[UN 24] UN Department of Economic and Social Affairs. *World Urbanization Prospects 2024*. 2024 (cf. p. 1).

[Zah+16] Matei Zaharia et al. **Apache Spark : A Unified Engine for Big Data Processing**. *Communications of the ACM* 59 :11 (2016), 56-65 (cf. p. 3).