Projet NoSQL : Kafka + Elasticsearch + Kibana

Un pipeline de données véhicules connectés

Amine, Samy, Mehdi
Étudiants BDML, EFREI

3 avril 2025

Table des matières

Chapitre 1

Introduction

Dans le cadre de notre projet NoSQL, nous avons choisi de travailler sur un écosystème combinant plusieurs technologies :

- **Apache Kafka** : une plateforme de streaming distribuée, open source, initialement développée par LinkedIn puis gérée par la fondation Apache. Elle est principalement écrite en Java et Scala.
- **Elasticsearch** : un moteur de recherche et d'analytics temps réel très performant, développé en Java par la société Elastic. Il est historiquement open source (sous licence Apache), puis a évolué vers la *Elastic License 2.0* (certaines fonctionnalités payantes).
- **Kibana** : un outil de visualisation et d'exploration des données pour Elasticsearch, également proposé par Elastic.

Notre projet consiste à mettre en place un **pipeline de données** où nous simulons des véhicules (positions GPS, vitesse, niveau de batterie, etc.) afin de tester la scalabilité, la résilience et les fonctionnalités de ces technologies. Nous avons choisi ces outils car ils sont très répandus dans le monde de la data (ingestion temps réel, indexation, analytics, monitoring...) et ils couvrent un large panel de cas d'usage : ingestion massive, stockage distribué, recherche full-text, création de dashboards, etc.

Comparaison avec d'autres NoSQL (MongoDB, Cassandra, Neo4j, Redis) :

Technologie	Modèle	Points forts	Points faibles
MongoDB	Document (JSON)	— Très flexible — Facile à installer	— Moins efficace pour la recherche full-text (comparé à Elastic) — Peut nécessiter un schéma plus rigoureux pour les grosses apps

Cassandra	Colonnes distribuées	— Très scalable en écriture — Faible latence	— Langage de requête (CQL) plus limité que SQL — Pas idéal pour la recherche textuelle
Neo4j	Graphes	— Excellente gestion des relations — Cypher = requêtes graphiques puissantes	— Moins adapté à l'ingestion massive temps réel — Moins efficace pour l'analytics basique
Redis	Clé-Valeur en mémoire	— Extrêmement rapide — Bonne pour le cache	— Stockage en RAM = coûts élevés — Pas d'analytics avancé
Kafka + Elasticsearch	*(Middleware + Moteur de recherche)*	— Ingestion temps réel (Kafka) — Recherche/analytics (Elastic) — Visualisation (Kibana)	— Plus complexe à mettre en place (plusieurs services) — Infrastructure distribuée (maintenance, monitoring)

Ainsi, Kafka + Elasticsearch + Kibana présentent un excellent compromis entre ingestion temps réel et analyses rapides, notamment sur des volumes importants de données.

Chapitre 2

Installation

2.1 Prérequis

— Docker Desktop (ou Docker Engine) installé sur la machine.
— Python 3 (pour simuler les données).

2.2 Étapes d'installation et de configuration

Nous avons créé un fichier `docker-compose.yml` permettant de lancer :
— **Zookeeper** (nécessaire pour Kafka)
— **Kafka**
— **Elasticsearch**
— **Kibana**
Voici le contenu principal (résumé) :

Listing 2.1 – Extrait du `docker-compose.yml`

```
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    ...

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    ...

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.11.1
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    networks:
      - elk

  kibana:
    image: docker.elastic.co/kibana/kibana:8.11.1
    environment:
```

```
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - "5601:5601"
    networks:
      - elk

networks:
  elk:
```

Une fois ce fichier en place, on lance la commande : `docker-compose up -d`

2.2.1 Captures d'écran (Installation)

Nous avons réalisé des captures d'écran pendant l'installation. Ci-dessous, **nous incluons les images sans réécrire leur contenu**, conformément aux consignes :

FIGURE 2.1 – Création du dossier `kafka-vehicle-project` et exécution de `docker-compose up -d`

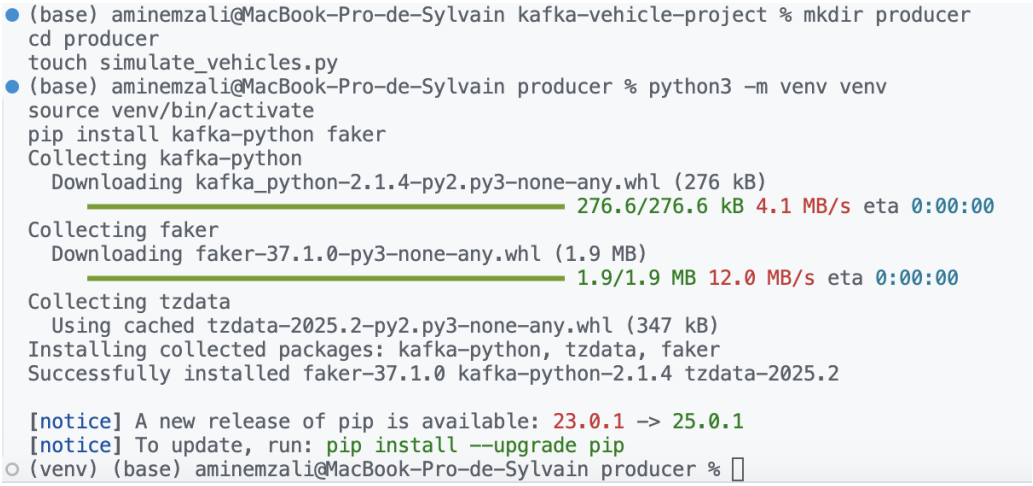

FIGURE 2.2 – Installation de Python `kafka-python` et `faker`

Chapitre 3

Création et manipulation des données (CRUD)

3.1 Mise en place du producteur de données

Pour simuler des données de véhicules, nous avons un script `simulate_vehicles.py` qui envoie des messages vers Kafka.

— **Création du topic** :

```
docker exec -it kafka \
  kafka-topics --create --topic vehicle-data \
  --bootstrap-server localhost:9092 \
  --partitions 1 \
  --replication-factor 1
```

— **Producteur Python** :

Listing 3.1 – Extrait de `simulate_vehicles.py`

```
from kafka import KafkaProducer
from faker import Faker
import json, time, random

fake = Faker()
producer = KafkaProducer(
  bootstrap_servers='localhost:9092',
  value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

def generate_vehicle_data(vehicle_id):
    return {
        "vehicle_id": vehicle_id,
        "@timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
        "latitude": float(fake.latitude()),
        "longitude": float(fake.longitude()),
        "speed_kmh": round(random.uniform(0, 120), 2),
        "battery_level": random.randint(10, 100)
    }

if __name__ == "__main__":
    vehicle_ids = [f"veh-{i}" for i in range(1, 6)]
```

```
while True:
    vehicle = random.choice(vehicle_ids)
    data = generate_vehicle_data(vehicle)
    print(f"Sending: {data}")
    producer.send("vehicle-data", data)
    time.sleep(1)
```

Quelques extraits de la console :

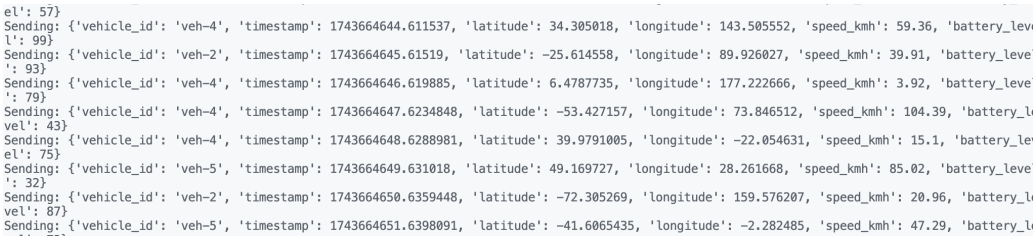

FIGURE 3.1 – Console du producteur : envoi des données simulées

3.2 Consommation et insertion dans Elasticsearch

Pour **lire les données** depuis Kafka et les envoyer à Elasticsearch, nous utilisons un `consumer_to_es.py` :

Listing 3.2 – Extrait de `consumer_to_es.py`

```
from kafka import KafkaConsumer
from elasticsearch import Elasticsearch
import json

# Connexion Elasticsearch
es = Elasticsearch("http://localhost:9200")

# Connexion au topic Kafka
consumer = KafkaConsumer(
    "vehicle-data",
    bootstrap_servers="localhost:9092",
    value_deserializer=lambda v: json.loads(v.decode("utf-8")),
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id="vehicle-consumer-group"
)

for message in consumer:
    data = message.value
    print(f"Received: {data}")
    # Indexation
    es.index(index="vehicle-data", document=data)
```

FIGURE 3.2 – Console du consommateur : réception et indexation dans Elasticsearch

3.3 CRUD : Créer, Lire, Mettre à jour, Supprimer

3.3.1 Créer (Create)

L'instruction `es.index(index="vehicle-data", document=data)` permet de *créer* automatiquement un document dans l'index `vehicle-data`. Si l'index n'existe pas, il est créé dynamiquement.

3.3.2 Lire (Read)

Nous pouvons lire les données via Kibana (section "Discover") ou via une requête REST :

```
curl -X GET http://localhost:9200/vehicle-data/_search?pretty
```

3.3.3 Mettre à jour (Update)

Pour modifier un document précis (connaissant son `_id`) :

```
curl -X POST "localhost:9200/vehicle-data/_update/<document_id>" \
     -H 'Content-Type: application/json' -d '{
  "doc": {
    "speed_kmh": 42.0
  }
}'
```

3.3.4 Supprimer (Delete)

```
curl -X DELETE "localhost:9200/vehicle-data/_doc/<document_id>"
```

Ou pour supprimer l'index complet :

```
curl -X DELETE http://localhost:9200/vehicle-data
```

Chapitre 4

Fonctionnalités avancées : agrégations et visualisations

Nous utilisons Kibana pour créer des **Index Patterns** (Data Views) et des **dashboards**. Une fois que `vehicle-data` est créé et que le champ `@timestamp` est reconnu comme date, nous pouvons :

— Explorer les données dans Kibana (Discover)
— Créer des visualisations (histogramme, pie chart, bar chart, etc.)

4.1 Exemple de dashboard

Nous avons créé trois visualisations :

1. Un **histogramme** du nombre de messages par tranche de temps
2. Un **pie chart** pour la répartition par `vehicle_id`
3. Un **bar chart** de la vitesse moyenne par véhicule

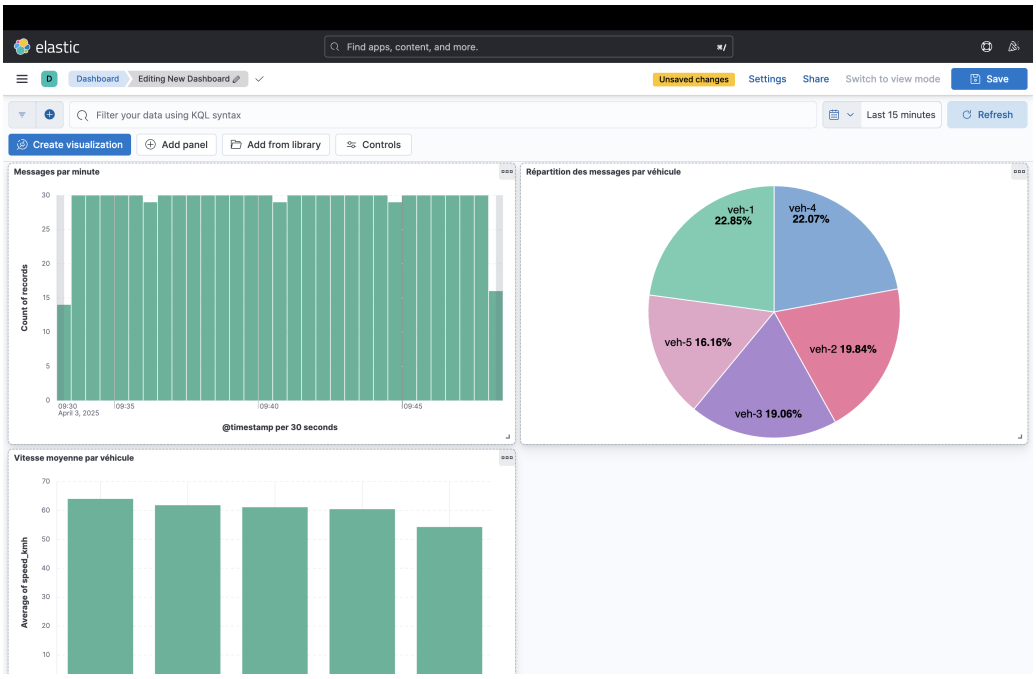

FIGURE 4.1 – Dashboard dans Kibana : histogramme, pie chart et bar chart

Chapitre 5

Conclusion et recommandations

5.1 Avantages de notre pipeline

— **Ingestion temps réel** : Kafka gère un flux continu de données et permet de scaler horizontalement en augmentant le nombre de partitions.
— **Indexation et recherche rapide** : Elasticsearch permet de chercher dans les documents quasi instantanément, et de faire des agrégations complexes.
— **Visualisation ergonomique** : Kibana fournit une interface simple pour créer des dashboards, analyser les données et effectuer des recherches.

5.2 Limites

— L'**infrastructure** est plus lourde que des solutions "simples" (ex. MongoDB). Il faut maintenir Kafka, Elasticsearch, Kibana, Zookeeper.
— Des **coûts d'hébergement** plus élevés dès lors qu'on veut de la haute disponibilité (clusters distribués).

5.3 Comparaison succincte

En comparaison à MongoDB, Cassandra, Neo4j ou Redis, ce pipeline met l'accent sur :
— **Consistance et scalabilité** (Kafka + ES sont distribués)
— **Recherche avancée et analytics temps réel** (Elasticsearch excelle en full-text et agrégations)
— **Résistance aux pannes** via la réplication Kafka et Elasticsearch.
D'autres solutions NoSQL (MongoDB, Redis, etc.) sont parfois plus simples à installer et maintenir, mais moins performantes pour un cas "ingestion massive + requêtes complexes + visualisation temps réel".

5.4 Perspectives

Pour aller plus loin, nous pourrions :
— Ajouter un **Kafka Connect** pour automatiser l'ingestion vers Elasticsearch
— Jouer sur le **nombre de partitions** Kafka et de **shards/réplicas** ES pour mesurer les performances
— Gérer la **géolocalisation** avec un type `geo_point` pour afficher les véhicules sur une carte.

5.5 Mot de la fin

Ce pipeline répond aux besoins d'une entreprise souhaitant *collecter et analyser en temps réel* une grande quantité de données de capteurs ou de logs. Grâce à l'élasticité de Kafka et la puissance d'Elasticsearch, associées à la visualisation de Kibana, nous obtenons une solution performante, évolutive et relativement simple à déployer via Docker.

— Fin du Rapport —