Report: Big Data Framework for Real-Time Smart Meter Energy Monitoring

Faculty of Sciences of Tunis
Computer Sciences Departement

Student: Fares Dkhili
Teacher: Rihab Mersni

Contents

1 General Context

1.1 Introduction

The modernization of electrical grids into "Smart Grids" has led to an explosion in the volume of data generated by potential metering points. Traditional database systems are often ill-equipped to handle the high velocity and variety of data coming from millions of smart meters in real-time. This project, VoltStream, aims to build a robust Big Data framework capable of ingesting, processing, and visualizing this data to provide actionable insights for grid operators in Tunisia.

1.2 Objectives

The primary goals of this project are:

- **Scalability**: he system must handle increasing loads (from thousands to millions of meters).
- **Real-Time Monotoring**: Detect anomalies (e.g., voltage spikes, outages) instantly.
- **Historical Analytics**: Analyze trends over time to forecast demand.
- **Visualization**: Provide a clear dashboard for decision-makers showing consumption by region (Governorate).

2 System Architecture

The solution follows the Lambda Architecture priciples, combining a **Speed Layer** for real-time alerts and a **Batch Layer** for Deep historical analysis.

2.1 Architecture Diagram

The following diagram illustrates the interaction between components:

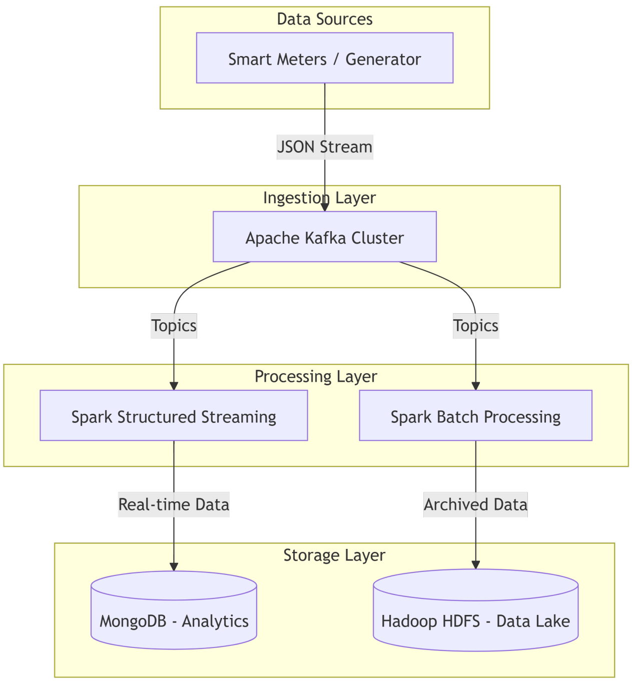


Figure 1: Architecture Diagram

2.2 Flow Description

1. **Generation**: The producer.py script simulates smart meters from all 24 Tunisian governorates, sending random readings (Voltage, Current, Power) to Kafka.

2. **Ingestion: Apache Kafka**: acts as the buffer, decoupling producers from consumers to ensure no data loss during bursts.

3. **Processing**:

   - **Spark Streaming**: consumes from Kafka, filters for anomalies (e.g., Power ¿ 10kW), and writes ”hot” data to MongoDB for the dashboard:
   - **Spark Batch**(periodically) reads raw data, aggregates it (e.g., Daily Average per Region), and stores it in HDFS for long-term archiving.

4. **Visualization**: A Flask web application polls MongoDB(or the mock API in this prototype) and rerders live charts using **Chart.js**

3 Technology Stack & Justification

Component	**Technology**	**Role**	**Justification**
Ingestion	**Apache Kafka**	Message Broker	Industry standard for high-throughput, fault-tolerant event streaming. Decouples processing from generation.
Processing	**Apache Spark**	Compute Engine	Unified engine for both SQL (Batch) and Streaming. In-memory processing makes it 100x faster than MapReduce.
Storage (Hot)	**MongoDB**	NoSQL DB	Schema-less design fits JSON data perfectly. Fast read/write performance for the real-time dashboard.
Storage (Cold)	**Hadoop HDFS**	Distributed FS	Cost-effective storage for massive amounts of historical CSV/Parquet data.
Backend	**Python (Flask)**	Web Server	Lightweight, easy to integrate with data science libraries, perfect for rapid prototyping.
Frontend	**HTML/JS**	UI	Simple, responsive design using Chart.js for rendering dynamic graphs on the client side.

Table 1: Technology Stack and Justification

4 Implementation Details

4.1 Data Augmentation (Tunisia Data)

File: `data/generate_tunisia_data.py`

To simulate a realistic scenario, we generated a dataset covering all 24 governorates of Tunisia (Ariana, Tunis, Sfax, etc.). The script randomizes consumption based on meter type (Industrial vs Residential).

Listing 1: Snippet from `generate_tunisia_data.py`

```
# Snippet from generate_tunisia_data.py
GOVERNORATES = ["Ariana", "Beja", "Ben Arous", ..., "Zaghouan"]
...
if meter_type == "Industrial":
    voltage = random.uniform(230.0, 240.0)
    current = random.uniform(30.0, 100.0) # Higher consumption
```

4.2 Data Ingestion (Kafka)

File `spark/producer.py` The rpducer reads the CSV and sends records as JSON messages to the `smart-meters` topic.

Listing 2: Snippet from spark/producer.py

```
#Snippet from spark/producer.py
producer = KafkaProducer(value_serializer=lambda v: json.dumps(v).encode('utf-8'))
for row in reader:
    producer.send('smart-meters', row)
```

4.3 Stream Processing (Spark Streaming)

File: `spark/stream_processing.py`

This job represents the "Speed Layer". It reads from Kafka, parses the JSON schema, and can trigger alerts.

Listing 3: Snippet from stream_processing.py (Speed Layer)

```
df = spark.readStream.format("kafka")...load()
parsed_df = df.select(from_json(col("value").cast("string"), schema)...)
# Real-time filtering
alerts = parsed_df.filter(col("power_kw") > 10.0)
```

4.4 Batch Processing (Spark Batch)

File: `spark/batch_processing.py`

Calculates the average power consumption per region for reporting.

Listing 4: Snippet from batch_processing.py (Batch Layer)

```
avg_power = df.groupBy("region").agg(avg("power_kw"))
```

4.5 Storage Layer (MongoDB & HDFS

Docker Compose: `storage/docker-compose.yml`

MongoDB is deployed in a container.

HDFS: Used for storing the raw `smart_meters.csv` and the output of batch jobs.

Command:

```
hdfs dfs -put ../data/smart_meters.csv /user/smart_meters/raw/
```

4.6 Visualization (Dashboard)

File: `dashboard/templates/index.html`

We utilize **Chart.js** to render two main visualizations:

1. **Line Chart**: Shows real-time total power consumption across Tunisia.
2. **Bar Chart**: Compares average power consumption by Governorate.

The Frontend polls the `/api/data` and `/api/stats` endpoints every 2 seconds.

5 Configuration & Deployment

5.1 Directory Structure

```
Big_data/
 dashboard/         # Flask Web App
 data/              # Datasets & Generators
 kafka/             # Docker-compose for Kafka
 spark/             # PySpark Scripts
 storage/           # Storage configs (Mongo/HDFS)
```

5.2 Prerequisites

- Ubuntu Linux (VM)
- Docker & Docker Compose
- Python 3.8+
- Apache Spark 3.x

5.3 Deployment Steps

1. **Start Services**:

```
cd kafka && docker-compose up -d
cd ../storage && docker-compose up -d
```

2. **Generate Data**:

```
python3 data/generate_tunisia_data.py
```

3. **Start Producer**:

```
python3 spark/producer.py
```

4. **Start Dashboard**:

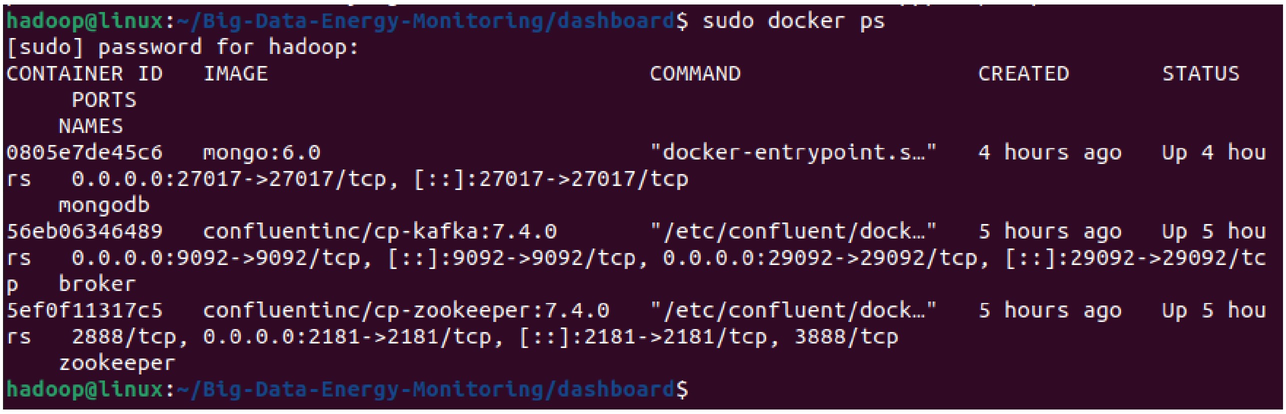
```
python3 dashboard/app.py
```

6 Execution Scenarios & Results

6.1 Scenario 1: Infrastructure Startup

We verify that Kafka Brokers and Zookeeper are running along with MongoDB.

- **Command**: `docker ps`
- **Result**: 3 Containers Active (broker, zookeeper, mongodb).



```
hadoop@linux:~/Big-Data-Energy-Monitoring/dashboard$ sudo docker ps
[sudo] password for hadoop:
CONTAINER ID   IMAGE                            COMMAND                  CREATED       STATUS
     PORTS
    NAMES
0805e7de45c6   mongo:6.0                        "docker-entrypoint.s…"   4 hours ago   Up 4 hou
rs   0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp
    mongodb
56eb06346489   confluentinc/cp-kafka:7.4.0      "/etc/confluent/dock…"   5 hours ago   Up 5 hou
rs   0.0.0.0:9092->9092/tcp, [::]:9092->9092/tcp, 0.0.0.0:29092->29092/tcp, [::]:29092->29092/tc
p   broker
5ef0f11317c5   confluentinc/cp-zookeeper:7.4.0   "/etc/confluent/dock…"   5 hours ago   Up 5 hou
rs   2888/tcp, 0.0.0.0:2181->2181/tcp, [::]:2181->2181/tcp, 3888/tcp
    zookeeper
hadoop@linux:~/Big-Data-Energy-Monitoring/dashboard$
```

Figure 2: Infrastructure Startup

6.2 Scenario 2: Data Streaming

When the producer runs, it prints `Sending: SM001...`. The Spark Streaming job (running in a separate terminal) instantly picks up these messages.

- **Result**: Console output showing batches of processed JSON data.

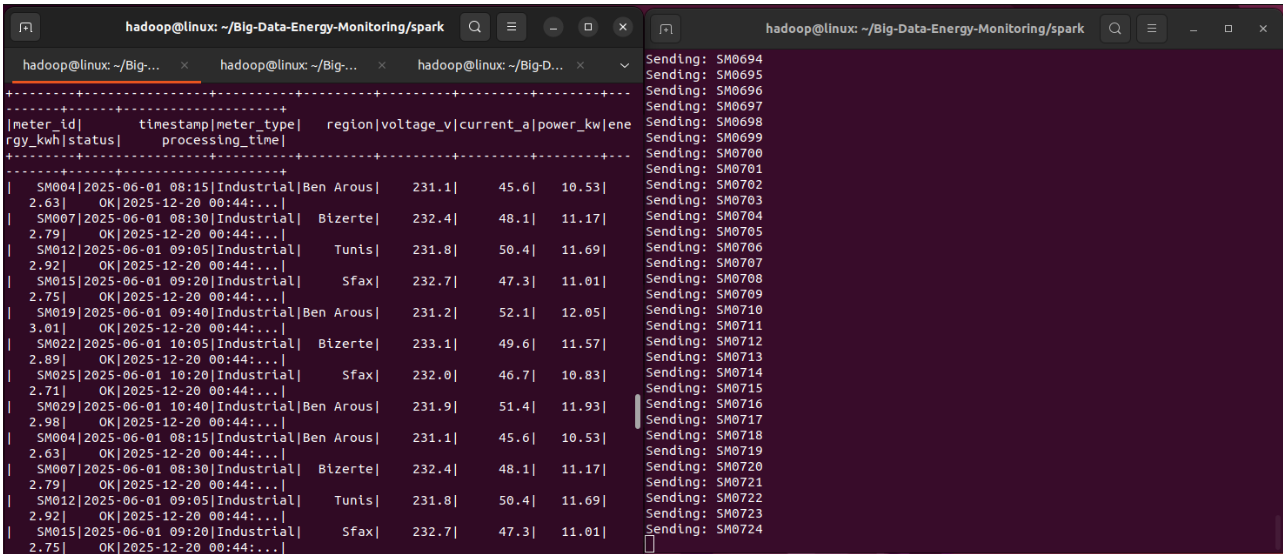


Figure 3: Spark terminal & Producer terminal

6.3 Scenario 3: Real-Time Dashboard

Accessing `http://localhost:5000` reveals the operational dashboard.

- **The Line Chart**: updates dynamically as new data arrives.

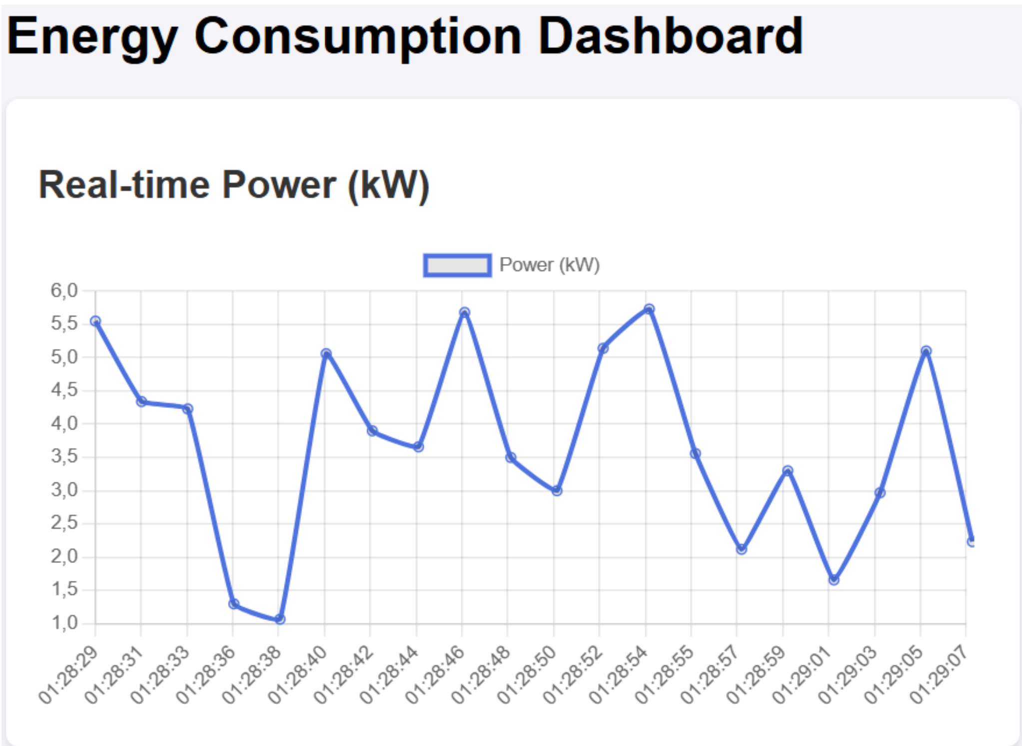


Figure 4: Energy Consumption Variation

- **The Bar Chart**: The Bar Chart shows “Sfax” and “Tunis” having higher bars due to simulated Industrial activity.

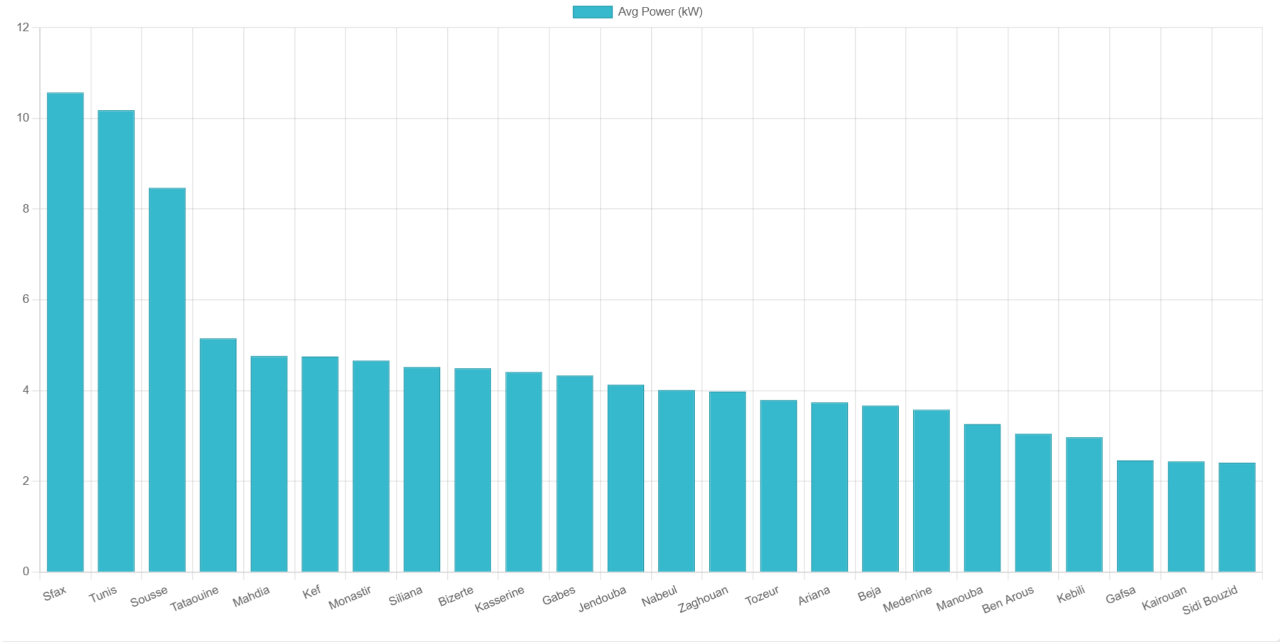


Figure 5: Power consumption accross states

- **Scrollable Live Feed**: showing all 24 governorates receiving data.

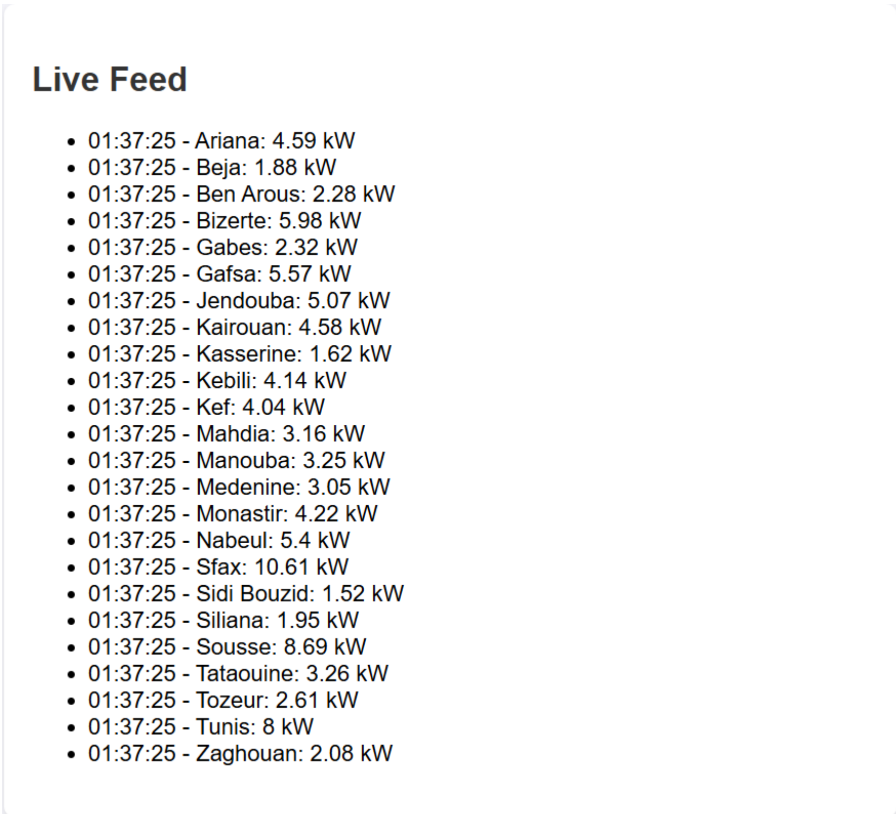


Figure 6: Live Feed consumption

7 Conclusion

This project successfully demonstrated the implementation of a scalable Big Data pipeline. By leveraging **Kafka** for buffering, **Spark** for unified processing, and **MongoDB** for rapid access, we created a system capable of monitoring Tunisia's energy grid in near real-time.

7.1 Future Enhancements

- Integrate a Machine Learning model (e.g., ARIMA or LSTM) in Spark Batch to **forecast** future consumption.
- Secure the cluster using **Kerberos**.
- Deploy on a cloud provider (AWS EMR or Azure HDInsight) for true horizontal scaling.