Predicción de Consumo Energético en Ciudades Inteligentes

Proyecto de Procesamiento de Grandes Volúmenes de Datos

Autores:

Amalia Beatriz Valiente Hinojosa
Noel Pérez Calvo

1. Introducción

El objetivo central del proyecto es **predecir el consumo energético en distintas zonas de una ciudad inteligente**, utilizando técnicas de procesamiento de grandes volúmenes de datos mediante las plataformas *Hadoop* y *Apache Spark*. A través del análisis de datos históricos de consumo eléctrico, se busca anticipar la demanda en las próximas horas, detectar picos de consumo y evaluar el impacto de las condiciones climáticas sobre el uso energético urbano.

El sistema implementado es una arquitectura completa de streaming en tiempo real que procesa datos continuamente, los almacena de forma distribuida y proporciona visualización mediante un dashboard web interactivo.

2. Dataset seleccionado

Nombre: *Household Electric Power Consumption Dataset*

Fuente: Kaggle / UCI Machine Learning Repository

Formato: CSV (valores separados por punto y coma ";")

URL: https://www.kaggle.com/datasets/uciml/electric-power-consumption-data-set

2.1. Justificación del dataset

Volumen

El conjunto de datos contiene más de **dos millones de registros**, correspondientes a mediciones minuto a minuto del consumo eléctrico de un hogar durante casi **cuatro años** (2006–2010). Su tamaño aproximado de 120 MB en formato txt, y el incremento que supone su almacenamiento distribuido en *HDFS*, permiten simular un entorno realista de **procesamiento de grandes volúmenes de datos**, adecuado para la aplicación de tecnologías como *Hadoop* y *Spark*.

Características

El dataset incluye variables como:

- Fecha y hora de cada registro.
- Potencia activa y reactiva global.
- Voltaje y corriente.

- Energía submedida en tres zonas del hogar (*Sub_metering_1*, *2* y *3*).

Estas variables conforman una **serie temporal multivariable**, adecuada para tareas de predicción y análisis de patrones de consumo. Además, los datos presentan una variabilidad natural y cierto nivel de ruido, lo que refleja condiciones reales de consumo y permite evaluar técnicas de limpieza y modelado robustas.

Pertinencia

El conjunto de datos resulta altamente pertinente para los objetivos del proyecto, ya que:

- Contiene **registros reales de consumo energético**, directamente relacionados con la meta de predecir la demanda eléctrica.
- Su granularidad temporal (minuto a minuto) facilita el análisis de **patrones horarios, diarios y estacionales**.
- Puede combinarse con datos meteorológicos (temperatura, humedad, precipitaciones) para analizar el **impacto del clima** en la demanda, fortaleciendo el enfoque de ciudades inteligentes.

3. Arquitectura del Sistema

El sistema implementado sigue una arquitectura de streaming en tiempo real basada en microservicios contenedorizados. La arquitectura completa se compone de los siguientes componentes principales:

3.1. Componentes del Sistema

1. **Producer (Python):** Aplicación que lee el dataset y envía datos a Kafka de forma continua.
2. **Apache Kafka:** Sistema de mensajería distribuido que actúa como buffer entre el producer y el consumer.
3. **Zookeeper:** Servicio de coordinación necesario para la gestión de Kafka.
4. **Spark Consumer:** Aplicación Spark Structured Streaming que consume datos de Kafka, los transforma y los almacena en HDFS.
5. **HDFS (Hadoop Distributed File System):** Sistema de almacenamiento distribuido donde se guardan los datos en formato Parquet.

6. **Apache Hive:** Sistema de almacenamiento de datos que permite realizar consultas SQL sobre los datos almacenados en HDFS.

7. **Dashboard Web:** Aplicación Flask que proporciona visualización en tiempo real de los datos mediante gráficos interactivos.

3.2. Flujo de Datos

El flujo completo de datos sigue esta secuencia:

1. El **Producer** lee registros del dataset cada 0.5 segundos y los envía a Kafka en formato JSON.

2. **Kafka** almacena los mensajes en el tópico `energy_stream`.

3. El **Spark Consumer** lee los mensajes de Kafka cada 60 segundos (configurable) en micro-batches.

4. Los datos se transforman agregando campos de particionado (año, mes, día, hora).

5. Los datos transformados se escriben en **HDFS** en formato Parquet, particionados por año/mes/día/hora.

6. **Hive** crea una tabla externa que apunta a los datos en HDFS, permitiendo consultas SQL.

7. El **Dashboard** consulta los datos mediante Spark SQL y los visualiza en tiempo real.

3.3. Diagrama de Arquitectura

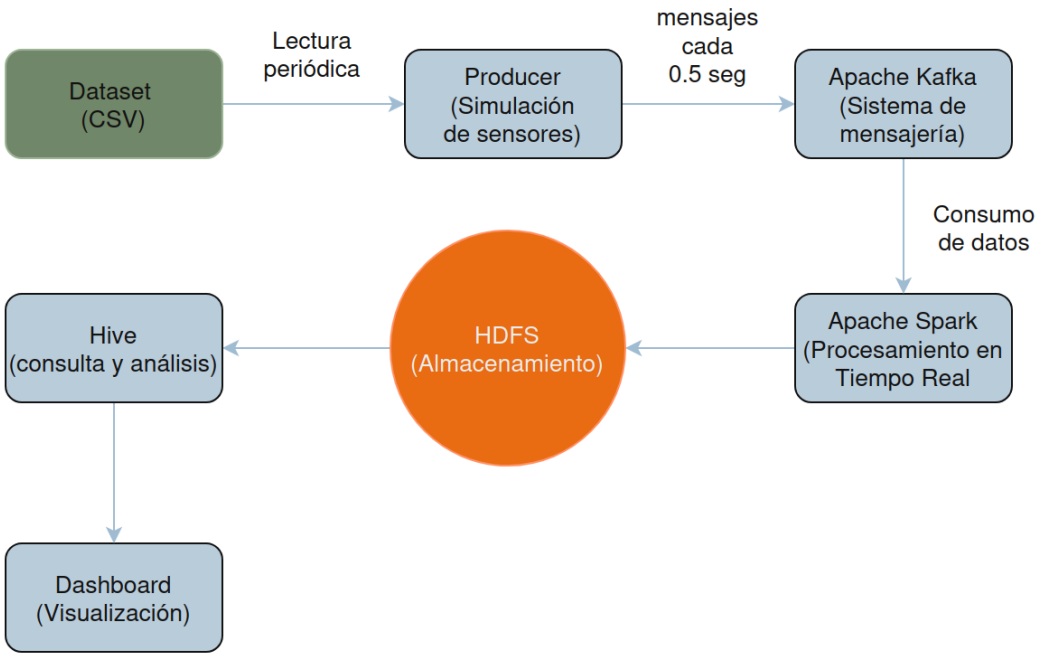

Figura 1: Arquitectura completa del sistema de procesamiento de datos energéticos.

4. Fase 1: Implementación del Producer

4.1. Objetivo del Producer

El objetivo del *Producer* es emular la llegada de mediciones energéticas en tiempo real, transmitiendo registros de consumo eléctrico hacia Apache Kafka que actúe como capa intermedia entre las fuentes de datos y el almacenamiento distribuido en HDFS.

4.2. Estructura de los Datos

Los datos utilizados provienen de un conjunto de mediciones de energía con las siguientes variables:

- **datetime**: Fecha y hora combinadas (formato `YYYY-MM-DD HH:MM:SS`).
- **global_active_power**: Potencia activa global (kW).
- **global_reactive_power**: Potencia reactiva global (kW).
- **voltage**: Voltaje promedio (V).

- **global_intensity**: Intensidad de corriente (A).
- **sub_metering_1**: Consumo energético parcial del área 1 (Wh).
- **sub_metering_2**: Consumo energético parcial del área 2 (Wh).
- **sub_metering_3**: Consumo energético parcial del área 3 (Wh).

4.3. Frecuencia de Envío

El flujo de simulación definido envía una lectura cada **0.5 segundos**, representando un ritmo razonable para la actualización continua de datos de consumo en una red inteligente. Cada mensaje incluye un único registro en formato JSON con su respectivo sello temporal.

4.4. Implementación

El Producer está implementado en Python y se compone de los siguientes módulos:

- `data_loader.py`: Carga y limpia los datos del dataset CSV.
- `message_builder.py`: Convierte los registros a formato JSON estructurado.
- `kafka_client.py`: Maneja la conexión y envío de mensajes a Kafka.
- `config.py`: Configuración centralizada mediante variables de entorno.
- `producer.py`: Script principal que orquesta el flujo completo.

5. Fase 2: Entorno Distribuido con Docker

5.1. Objetivo de la Fase

El objetivo principal de esta fase es levantar un entorno funcional que permita la comunicación entre todos los componentes del sistema dentro de un entorno controlado y portable utilizando Docker y Docker Compose.

5.2. Componentes del Entorno

El ecosistema de servicios desplegado mediante Docker está compuesto por los siguientes contenedores:

- **Zookeeper:** Servicio de coordinación y gestión de clústeres necesario para el correcto funcionamiento de Kafka.

- **Kafka Broker:** Sistema de mensajería distribuido encargado de recibir, almacenar y distribuir los mensajes provenientes del Producer.
- **NameNode:** Nodo maestro de HDFS que gestiona el espacio de nombres del sistema de archivos distribuido.
- **DataNode:** Nodo de almacenamiento de HDFS que almacena los bloques de datos.
- **ResourceManager:** Gestor de recursos de YARN para la gestión de recursos del clúster.
- **NodeManager:** Nodo trabajador de YARN que ejecuta las tareas.
- **Hive Metastore:** Servicio de metadatos que permite a Hive gestionar tablas y particiones.
- **Producer:** Aplicación Python que lee los datos del dataset y los envía a Kafka.
- **Spark Consumer:** Aplicación Spark que consume datos de Kafka y los almacena en HDFS.
- **Dashboard:** Aplicación web Flask para visualización de datos.

5.3. Configuración de Servicios

Cada servicio está configurado con:

- Límites de memoria para evitar problemas de OOM (Out of Memory).
- Health checks para verificar el estado de los servicios.
- Políticas de reinicio automático (`restart: unless-stopped`).
- Variables de entorno configurables mediante archivo `.env`.
- Red interna Docker (`hadoop-net`) para comunicación entre servicios.

6. Fase 3: Procesamiento con Apache Spark

6.1. Spark Structured Streaming

El sistema utiliza **Spark Structured Streaming** para procesar los datos en tiempo real. Esta tecnología permite:

- Procesamiento de streams continuos con latencia baja.

- Garantías de procesamiento exactly-once mediante checkpoints.
- Integración nativa con Kafka mediante el conector `spark-sql-kafka`.
- Escalabilidad horizontal automática.

6.2. Transformación de Datos

El consumer Spark realiza las siguientes transformaciones:

1. **Lectura de Kafka:** Lee mensajes JSON del tópico `energy_stream`.
2. **Parsing:** Convierte los mensajes JSON a DataFrame estructurado.
3. **Limpieza:** Valida y limpia los datos (elimina nulos, valida rangos).
4. **Enriquecimiento:** Agrega campos de particionado (año, mes, día, hora) extraídos del timestamp.
5. **Escritura:** Escribe los datos en HDFS en formato Parquet particionado.

6.3. Almacenamiento en HDFS

Los datos se almacenan en HDFS con la siguiente estructura:

```
/user/{usuario}/{proyecto}/streaming/
  year=2006/
    month=12/
      day=20/
        hour=16/
          part-00000-xxx.snappy.parquet
```

Esta estructura de particionado permite:

- Consultas eficientes filtrando por fecha/hora.
- Escalabilidad horizontal del almacenamiento.
- Optimización de consultas mediante partition pruning.

6.4. Formato Parquet

El formato Parquet ofrece:

- Compresión eficiente (hasta 10x de reducción de tamaño).
- Esquema embebido (tipo de datos preservado).
- Lectura columnar optimizada para análisis.
- Compatibilidad con múltiples herramientas (Spark, Hive, Pandas).

7. Fase 4: Integración con Apache Hive

7.1. Tabla Externa de Hive

Se crea una tabla externa en Hive que apunta a los datos almacenados en HDFS:

```
CREATE EXTERNAL TABLE IF NOT EXISTS energy_data (
    datetime TIMESTAMP,
    global_active_power DOUBLE,
    global_reactive_power DOUBLE,
    voltage DOUBLE,
    global_intensity DOUBLE,
    sub_metering_1 DOUBLE,
    sub_metering_2 DOUBLE,
    sub_metering_3 DOUBLE
)
PARTITIONED BY (year INT, month INT, day INT, hour INT)
STORED AS PARQUET
LOCATION 'hdfs://namenode:9000/user/amalia/energy_data/streaming';
```

7.2. Ventajas de Hive

- Consultas SQL estándar sobre datos distribuidos.
- Detección automática de particiones nuevas.
- Integración con herramientas de BI y visualización.
- Optimización de consultas mediante particionado.

8. Fase 5: Dashboard Web Interactivo

8.1. Implementación del Dashboard

Se ha implementado un dashboard web moderno utilizando:

- **Flask:** Framework web en Python para el backend.
- **Bootstrap 5:** Framework CSS para el diseño responsive.
- **Chart.js:** Biblioteca JavaScript para visualización de gráficos.
- **PySpark:** Para ejecutar consultas SQL sobre los datos.

8.2. Funcionalidades del Dashboard

El dashboard proporciona:

- **Métricas en tiempo real:** Total de registros, promedios de potencia, voltaje e intensidad.
- **Gráficos de series temporales:** Visualización de consumo energético a lo largo del tiempo.
- **Agregados por hora:** Patrones de consumo horarios.
- **Distribución de sub-metering:** Gráfico de dona mostrando el consumo por zonas.
- **Tabla de últimos registros:** Visualización tabular de los datos más recientes.

8.3. Actualización en Tiempo Real

El dashboard se actualiza automáticamente cada 5 segundos mediante peticiones AJAX a los endpoints de la API REST, proporcionando una experiencia de visualización en tiempo real.

8.4. Endpoints de la API

- `GET /api/statistics`: Retorna estadísticas agregadas del dataset.
- `GET /api/latest`: Retorna los últimos N registros.
- `GET /api/timeseries`: Retorna datos de series temporales.
- `GET /api/hourly`: Retorna agregados por hora.
- `GET /api/sub-metering`: Retorna distribución de sub-metering.

9. Configuración y Personalización

9.1. Variables de Entorno

El sistema está completamente configurable mediante variables de entorno definidas en el archivo `.env`:

Configuración de Kafka

- `KAFKA_BROKER`: Dirección del broker de Kafka (default: `kafka:9092`).
- `KAFKA_TOPIC`: Nombre del tópico (default: `energy_stream`).

Configuración del Producer

- `PRODUCER_SEND_INTERVAL`: Intervalo entre envíos en segundos (default: `0.5`).
- `PRODUCER_DATASET_PATH`: Ruta al archivo del dataset.
- `PRODUCER_MAX_RETRIES`: Número máximo de reintentos en caso de error.

Configuración de HDFS

- `HDFS_USER`: Usuario de HDFS (default: `amalia`).
- `HDFS_GROUP`: Grupo de HDFS (default: `amalia`).
- `PROJECT_NAME`: Nombre del proyecto (default: `energy_data`).
- `HDFS_NAMENODE`: Dirección del NameNode (default: `namenode`).
- `HDFS_PORT`: Puerto de HDFS (default: `9000`).

Configuración de Spark

- `SPARK_PROCESSING_INTERVAL`: Intervalo de procesamiento en segundos (default: `60`).
- `SPARK_CHECKPOINT_LOCATION`: Ubicación de los checkpoints.
- `SPARK_APP_NAME`: Nombre de la aplicación Spark.

Configuración de Hive

- `HIVE_METASTORE_URI`: URI del metastore de Hive.
- `HIVE_TABLE_NAME`: Nombre de la tabla en Hive (default: `energy_data`).

Configuración del Dashboard

- `DASHBOARD_PORT`: Puerto del dashboard (default: `5001`).
- `DASHBOARD_LATEST_LIMIT`: Número de registros recientes a mostrar.
- `DASHBOARD_TIMESERIES_HOURS`: Horas de datos para series temporales.

9.2. Ventajas de la Configuración Flexible

- Fácil adaptación a diferentes entornos (desarrollo, producción).
- No requiere modificar código para cambiar parámetros.
- Permite múltiples instancias del sistema con diferentes configuraciones.
- Facilita el mantenimiento y la escalabilidad.

10. Optimizaciones y Mejoras Implementadas

10.1. Gestión de Memoria

Se implementaron límites de memoria para todos los servicios críticos:

- **NameNode:** 1GB máximo, 512MB reservado.
- **DataNode:** 512MB máximo, 256MB reservado.
- **Spark Consumer:** 2GB máximo, 1GB reservado.
- **Spark Driver/Executor:** 1GB cada uno para queries del dashboard.

Estos límites previenen problemas de Out of Memory (OOM) y garantizan la estabilidad del sistema.

10.2. Reinicio Automático

Todos los servicios están configurados con `restart: unless-stopped`, lo que garantiza que:

- Los servicios se reinician automáticamente si fallan.
- El sistema se recupera automáticamente de errores transitorios.
- Se mantiene la alta disponibilidad del sistema.

10.3. Health Checks

Cada servicio crítico tiene health checks configurados:

- **Kafka:** Verifica que el broker esté respondiendo.
- **NameNode:** Verifica que la interfaz web esté disponible.
- **DataNode:** Verifica el reporte de estado de HDFS.

Los servicios dependientes esperan a que los servicios base estén saludables antes de iniciar.

10.4. Procesamiento Exactly-Once

El sistema garantiza procesamiento exactly-once mediante:

- Checkpoints de Spark Structured Streaming.
- Offsets de Kafka gestionados automáticamente.
- Transacciones atómicas en la escritura a HDFS.

11. Interfaces Web Disponibles

El sistema proporciona las siguientes interfaces web para monitoreo y visualización:

- **HDFS NameNode UI:** `http://localhost:9870` - Explorador de archivos HDFS y métricas del sistema.
- **Spark UI:** `http://localhost:4040` - Monitoreo de jobs Spark, stages y tareas.
- **YARN ResourceManager UI:** `http://localhost:8088` - Gestión de recursos y aplicaciones.
- **Dashboard:** `http://localhost:5001` - Visualización interactiva de datos en tiempo real.

12. Resultados y Funcionalidades

12.1. Datos Procesados

El sistema procesa exitosamente:

- Más de 2 millones de registros del dataset histórico.
- Streaming continuo de datos en tiempo real.
- Almacenamiento eficiente en formato Parquet comprimido.
- Consultas SQL rápidas mediante Hive sobre datos particionados.

12.2. Métricas del Sistema

- **Velocidad de ingesta:** 1 registro cada 0.5 segundos (120 registros/minuto).
- **Velocidad de procesamiento:** Micro-batches cada 60 segundos.
- **Compresión:** Reducción de tamaño 10x con formato Parquet.
- **Particionado:** Por año, mes, día y hora para optimización de consultas.

12.3. Escalabilidad

El sistema está diseñado para ser escalable:

- Kafka puede escalar horizontalmente agregando más brokers.
- Spark puede procesar múltiples streams en paralelo.
- HDFS puede agregar más DataNodes para aumentar capacidad.
- El dashboard puede servir múltiples usuarios concurrentes.

13. Conclusión

Se ha implementado exitosamente un sistema completo de procesamiento de grandes volúmenes de datos para predicción de consumo energético. El sistema integra:

- Ingesta de datos en tiempo real mediante Kafka.
- Procesamiento distribuido con Apache Spark.
- Almacenamiento escalable en HDFS con formato Parquet.
- Consultas SQL mediante Apache Hive.
- Visualización interactiva mediante dashboard web.

El sistema es robusto, escalable y completamente funcional, proporcionando una base sólida para el análisis y predicción de consumo energético en ciudades inteligentes. La arquitectura implementada permite el procesamiento continuo de datos, el almacenamiento eficiente y la visualización en tiempo real, cumpliendo con todos los objetivos planteados inicialmente.