Grid Short Term Load Forecasting and Real Time Demand Management

- Report By: Sushant Sonar

Table of Contents

Data Flow Diagram:

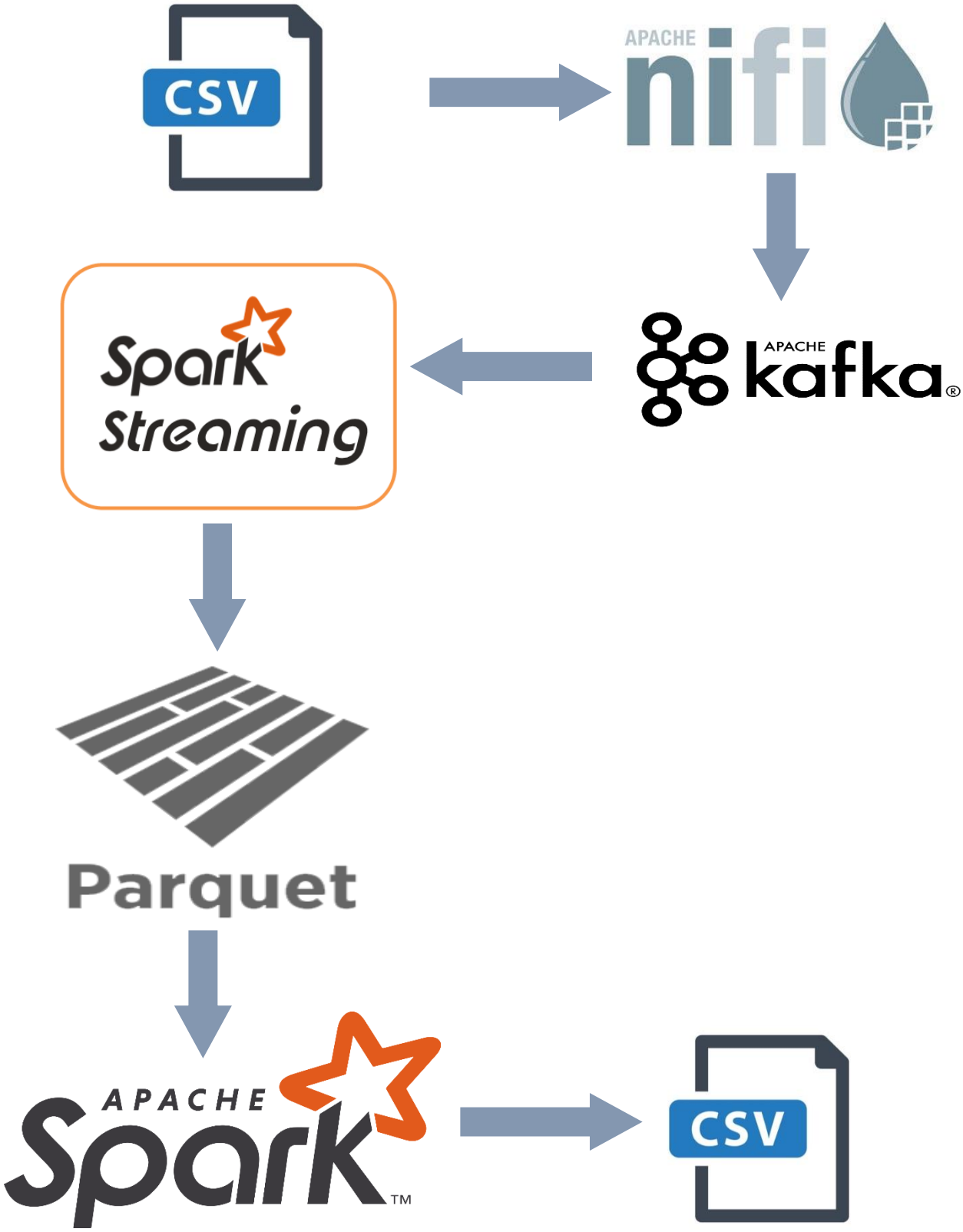

Figure 1: Data Flow Diagram

Data Flow:

I have divided this data flow into two tasks. One is streaming data flow while other runs in batch of one hour.

I. **Streaming Data Flow:**

CSV -> NiFi -> Kafka -> Spark Streaming -> Parquet

Here, pipeline starts by reading csv files using NiFi and sending them to spark streaming via kafka.

In spark streaming, I have done hourly aggregation on data. I have aggregated for hourly mean per house_id per household_id for that date-hour. This aggregation will be done on streaming data received per minute or second.

Further, I am storing these aggregations in a parquet file partitioned by date. I have partitioned parquet file by date as I have filtered data by date in the next step of pipeline.

II. **Batch Data Flow:**

Parquet -> Spark -> CSV

Here, we are reading required data from parquet file and processing it in spark. After pre-processing is done, spark does aggregations to generate both alerts and store them in csv file in 'append' mode. So, after each hour, new alerts get appended to existing csv file.

Cluster Setup:

I have setup all required software on my laptop. I am not dealing with any remote server here.

I. **NiFi:**

- I have used NiFi to read files from directory and send them to kafka.
- I have created a small pipeline consisting of two processors GetFile and PublishKafka_1_0.
- GetFile will read files from a directory as soon as files is arrives in directory. It will erase the file once read.
- PublishKafka_1_0 will send flowfile content to a kafka topic as a message.

See images below showing NiFi flow and configuration

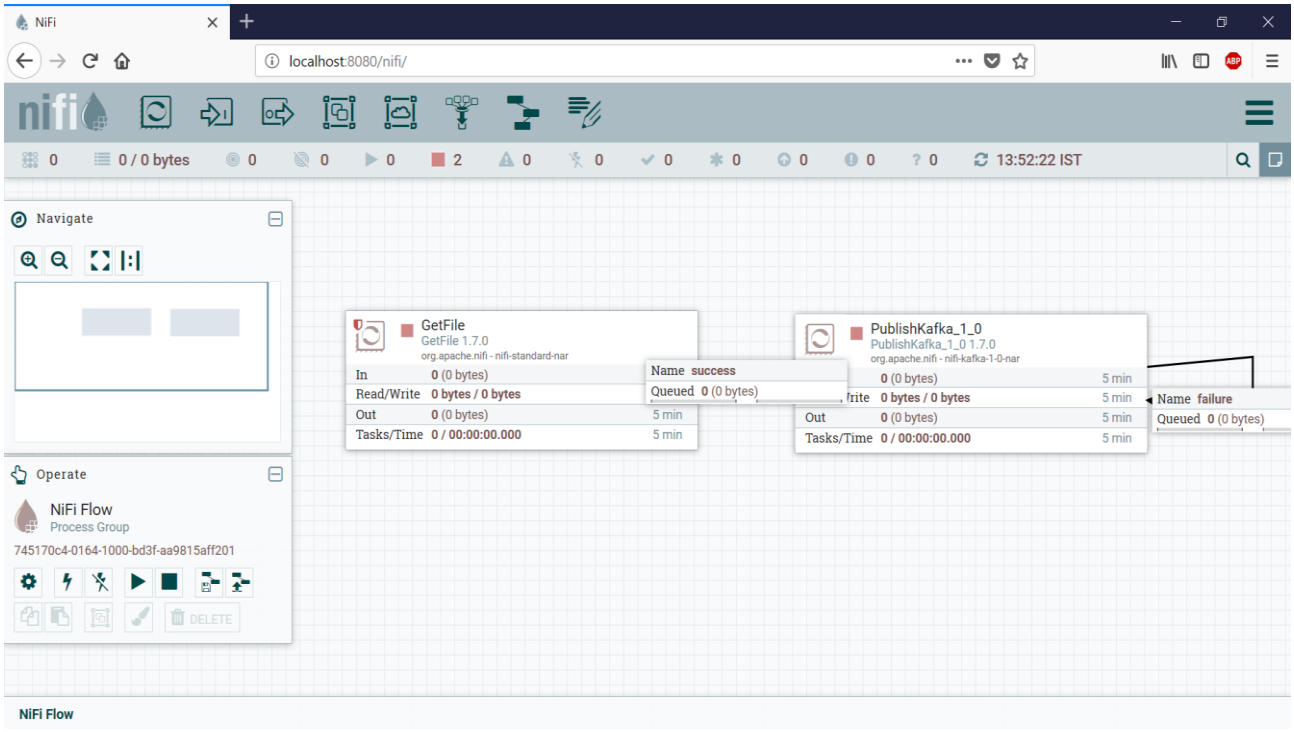

Figure 2: NiFi Flow

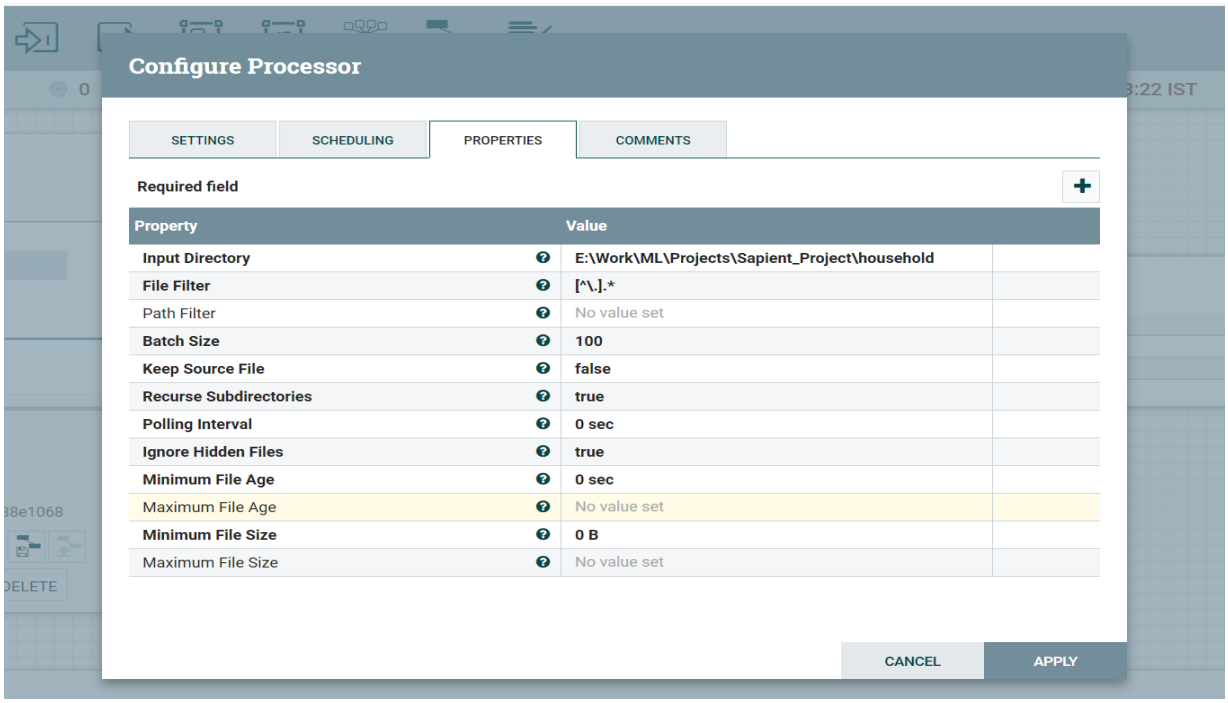

Figure 3: GetFile Configuration

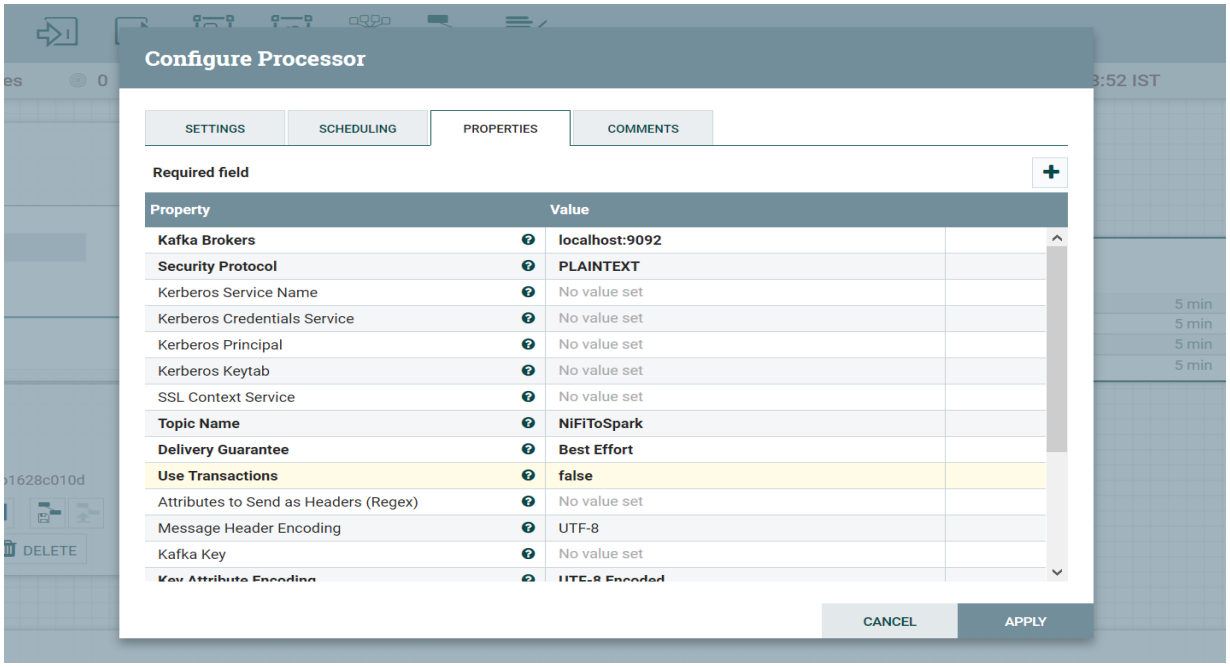

Figure 4: PublishKafka_1_0 Configuration

Why I chose NiFi:

1. NiFi is easy to use beacuase of UI based interface.
2. It is scalable. So in future if volume of data increases, we can add nodes to this NiFi cluster.
3. NiFi provides support for almost all types of files, databases. You can also do some preprocessing or validation if needed using NiFi.

II. Kafka:

- Created a topic named **'NiFiToSpark'**.
- kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic NiFiToSpark
- Considering my local machine, I kept **replication-factor and number of partitions 1**. For real scenario, we can decide their value based on cluster configuration.

Data Preprocessing:

A. In Spark Streaming

1. Dropping duplicates :
   Used 'dropDuplicates()' function
2. Converting timestamp to date_time variable as per UTC time zone.
3. Creating new features:
   - Converted timestamp value to ‘datetime’ value. It is combination of date and time.
   - Created a new column ‘date’ which is required for partitioning of parquet file.

B. In Batch Processing:

1. Created new column ‘hour’ which is required for alert_type_1.
2. Finding Nulls:
   -While doing a groupBy on streaming data, if data for a household for an hour is missing, that data is not included in result. So, such data is missing in the parquet file.
   - To detect such missing values, I have created a dataframe of all houses and households combination for that date_time. This dataframe contains 3 columns (date_time, house_id and household_id) and 289 rows. Further, I have joined data from parquet file with this dataframe to detect nulls.
3. Null Value Imputation:
   - The distribution of values of a household for an hour is skewed with 2 or 3 peaks. So, median will be a good measure to impute nulls rather than average.
   - I have replaced null value by the **median** of historical values of that household for that particular hour.
   - Spark doesn’t provide any function to get median of data. So, I have created a UDF to calculate median.
   - I have used ‘**Window’** functionality provided by spark to partition required data.

w=Window.partitionBy('house_id','household_id','hour').orderBy('date_time').rowsBetween(float(‘-inf’),0)

- Nulls whose median was not calculated due to lack of historical data were imputed by 0.

Alert Generation:

To get alert, we need to calculate mean and standard deviation.
To find how many standard deviations away the current datapoint is, I have used following formula:

z= (datapoint – mean)/ standard_deviation

I. **Alert Type 1:**

Here, we want to generate alert if mean consumption of a household for an hour is greater than 1 standard deviation of household's historical mean consumption for that hour.

To get mean and standard deviation required to calculate z, I have used '**Window**' functionality of spark.

w=Window.partitionBy('house_id','household_id','hour').orderBy('date_time').rowsBetween(float('-inf'),0)

data.withColumn('result',(col('value')-avg('value').over(w))/stddev('value').over(w))

This will partition data based on house_id, household_id and hour. Further it will order data by 'date_time'. It will calculate historical mean and standard deviation from this partitioned data. Then it will create a new column 'result' which is nothing but the z score of that row.

II. **Alert Type 2:**

Here, we want to generate alert if hourly consumption for a household is higher than 1 standard deviation of mean consumption across all households within that particular hour on that day.

As we want mean and standard deviation for an hour on that day, I have used 'date_time' column to group the data and calculate mean and standard deviation.

Future Scope:

Because of time constraints, I mainly focused on data pipeline part and not on the analysis part. There is much scope to improve this model for better accuracy. Some points to mention:

- Null-value Imputation: Instead of using last known non-null value, historical mean or moving average of the household for that hour can be used.
- Instead of parquet file, MongoDB can be used to store results of spark streaming. We can utilize indexing facility of MongoDB so access data faster. Also IO delay for parquet file slows down the process.
- Currently to get historical mean and standard deviation, we are loading all previous data. We can do some streaming calculations for this purpose so that loading all previous data can be avoided.

  e.g.

  Mean can be calculated as:

  meanDiff= (newValue – currentMean) / currentCount

  newMean= currentMean + meanDiff