Improving processing of real-time Big Data in Smart Grids using Apache Flink and Kafka

MSc Research Project
Cloud Computing

Supritha Shetty
Student ID: x18163009

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Supritha Shetty
Student ID:	x18163009
Programme:	Cloud Computing
Year:	2019
Module:	MSc Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	12/12/2019
Project Title:	Improving processing of real-time Big Data in Smart Grids using Apache Flink and Kafka
Word Count:	7080
Page Count:	22

Signature:	
Date:	12th December 2019

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Improving processing of real-time Big Data in Smart Grids using Apache Flink and Kafka

Supritha Shetty
x18163009

Abstract

As we inject more and more "smartness" into energy networks, it unfolds fine-grained data. Around 80 percent of electric meters determined to be replaced by smart meters by end of 2020 in European Union, evidently putting forward plethora of data. Processing and analyzing this Big Data would be complex, time consuming and would have more latency with lesser throughput. Data received by smart meters are at unprecedented speed at real-time, having said that a constant store of data received via smart meters also needs to be addressed, this paper aims to process the data at real-time as well as data received in batches or at store, with higher throughput and lower latency and much faster a execution time.There are different types of existing frameworks and techniques to process Smart grid Big Data being touched upon,this research focuses on using the two most relevant techniques. Apache Kafka being the best combination with Apache Flink using the in-built connector which helps to receive streaming data also being scalabe and fault-tolerant, With that making use of Apache Flink's own features in an optimized manner such as Windowing, its GlobalJobParameters as well as using Flink's Event time and Processing time with Java Programming language. The architecture is used to process Big Data from smart meters to achieve the end result help increasing the overall performance in real-time using cloud based services by deploying the JAR. A significant encouraging results of execution time being lesser when compared to existing approaches can be observed in the results.

1 Introduction

With cloud computing and energy networks tending towards digital technologies more and more data are induced through devices such as smart meters, thermostats, smart plugs, sensors(sensing the purity of air) and also data from two way communication initiated by the smart devices. All these systems are known as smart grids which depicts the consumption of energy at real-time and is controllable and adjustable. A precise reading on energy consumption can be achieved through smart meters unlike the traditional system mostly based on monthly visits and assumed on previous monthly bills. Smart Grid ongoing improvements and being adjustable with accurate results its a turning point for Big Data being generated, evidently needing to be processed at real-time.

1.1 Background

According to Horban (2018) the problems of capturing, analysing, managing the data and visualising is all resolved with help of big data. Big Data is allude to 5v's Li (2014), Acharjya and P (2014), Krüger1 and Teuteberg2 (2015), with respect to smart grid, the data has enclosed enclosed in Volume, Variety, Veracity, Velocity, Value. As claimed by Kamstrup (2019) the more data you have the value you create. When it comes to real-time data processing the instant need to process Big Data to get beneficial outcomes processing framework such as Apache Hadoop was put forth as with MapReduce as its default processing engine but it certainly did not support a real-time energy data.As shown in Figure 1 To being with lets understand why is it necessary to process data at

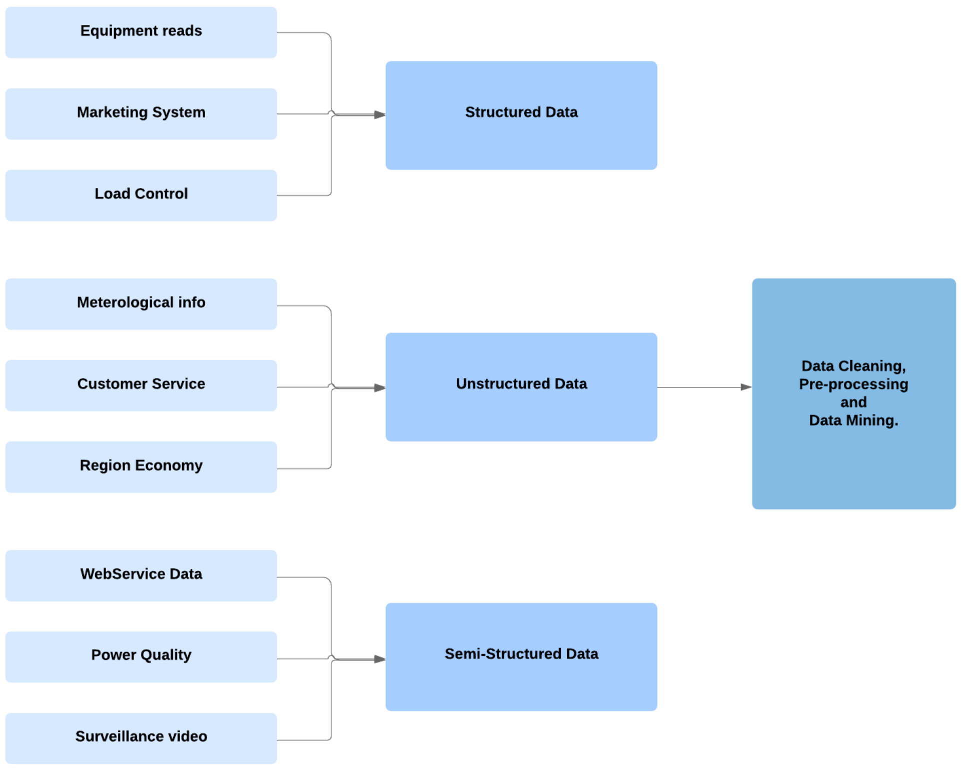


Figure 1: Smart Grid Data Distribution and process

real-time? Prathik et al. (2018) states that processing real time smart meter data does not only helps user understand the daily consumption but also weekend consumption if the user is away, helping to understand the habits and reduce consumption thereafter. The data includes switch stations, meters, and non-electrical information such as marketing, economic data are diverse in nature. Looking into this, Apache Spark with the fault tolerance of MapReduce working in streams and batches having its own integrations, libraries and tools was taken ahead Pérez-Chacón et al. (2018).

Lambda architecture which is also refereed in many papers, the original goals of this architecture were to improve the high result latency and have speed and batch processing performed Munshi and Mohamed (2018). However, it has a few notable limitations, first being two semantically equivalent implementations and the logic for two separate processing systems with different APIs are required. Second, finally the results achieved are only approximate and not accurate. Third it is difficult to set up and also to maintain having different resources such as databases and frameworks.

1.2 Research Gap and Contribution

Having said that Spark dominates when it comes to real-time streaming of Big Data however, is incapable of native iterations. To implement an iterative algorithm, a loop has to be repeated in order to to execute the step function and in addition check manually the termination steps, remarkably increasing overhead for large-scale iterative jobs. With such case this paper utilizes Flink's pipeline-based architecture with Windowing has one of the major feature. Unlike Spark, Flink will not split the stream data in micro-batches and with respect to iterative algorithms, the iterative step function needs to be scheduled once and the repetition will be handled by Flink engine.

Flink has the ability to process batch and streaming data without two separate processing systems as required by Lambda architecture, making it much easier to setup, hence both batch and stream processing with use of the relevant API's will be demonstrated in this project. Flink has the socketstreaming class having to process the data at real-time without use of external frameworks will be implemented in this paper, but for heavy data streaming Apache Kafka has been used,has many of the paper use API's or other 3rd party systems such as GitLabD for real-time streaming data. Apache Flink with best execution strategy having able to embracing Event-driven applications and Processing time application offering several benefits has been put to use in this research proposal. Help run consecutive operations on MAVEN without any need of manual tweaking hosted on cloud environment.

1.3 Research Question and Objective

How to improve processing of Real-time Big Data using Apache Flink and Kafka?

The primary objective of this study is to evaluate the performance of architecture designed with help of Apache Flink and Apache Kafka using cloud platform. The performance evaluation will be conducted taking terms of throughput and latency with making the architecture fault-tolerant. The evaluation result will be compared to common stream processing frameworks and related existing approaches as mentioned in Section II helping to answer the research question. Many authors despite of using the widely known approaches have mentioned some limitations and drawbacks with regards to processing methods, speed, efficiency, latency which are been tried to overcome using the implementations and environment setup.

Further paper is organized in following manner, Section 2 talks depicts related work on various Big Data processing frameworks with relation to smart grids and its processing time and efficiency. Section 3 and 4 describes the architecture methodology and design specification followed by section 5 containing the brief implementation. Section 6 presents a performance evaluation under Results and Critical analysis, a testing proof of the concept and, section 7 gives the concluding remarks.

2 Related Work

Based on research references, Big Data is significantly precise when it comes to smart grids. Processing the data at real-time having it stored in cloud is a appropriate solution for various properties such as elasticity scalablility. Numerous articles with various architectures are been proposed when it comes to processing data using data analytic technique and cloud computing.

2.1 Lambda Architecture for Big Data Energy Management

Generally a Lambda Architecture consists of three layers a batch layer which performs the batch processing, a speed layer performing the stream processing and a serving layer for data storage. Due to this 3 layers a batch data, and real-time data are taken care of as when data enters the system feds the data into this two layers as per computation to be processed.Different attempts have been made to merge the traditional batch processing approach analytics with popular stream processingHeidrich et al. (2016). The serving layer or also called as speed layer Heidrich et al. (2016) aggregates the outputs from batch and speed layers, storing the data in a datastore which uses different data storage systems. Author Kiran et al. (2015) states that this architecture is not only helpful for processing but is cost-effective at the same time. The uses are depicted in graphs and how kensis, aws resources cost monthly and how processing and correcting code errors having it tested before deploying help reduce the cost is mentioned. Following the experiments further with help of 32 core Linux machine with ram of 128GB,1594 count customers data Author Munshi and Mohamed (2018) took less than 100 seconds to calculate the above written function. As per authors experiment on dataset features such as robustness, fault-tolerance can be achieved by replicating the data and storing it in many nodes, which also helps achieve low-latency, scalablility and flexibility. However, the time taken to setup each resources and cost is high, making it difficult to setup and manage.

2.2 Apache Hadoop Based Framework for Energy Management

Hadoop based framework is an intial framework, the analysis were carried out by many authors referring to features using different approaches. Author Vaidya and Deshpande (2019) states that having a major advantage of legacy power Hadoop can be easily trusted on and is turned out to be a promising platform in the analysis performed. Author clearly explains on SCADA systems and how those system have MapReduce capability making 1000 terabyte of data to be easily scalable across 4000 nodes and handling failures are easier. However in this paper author compares the structure with database and hence stating the database takes 10-15 hours so therefore Hadoop does the better job comparatively. Author have not taken in consideration of any other frameworks or even Apache Spark. A similar comparsion is be stated by Shrivastava and Shrivastava (2018) comparing with SQL.

Hadoop is the core component in the data analysis process is notified by Merla and Liang (2018), which can support the processing of large datasets using distributed algorithms. Achieving the scalability and when it comes to latency there is a bit trade-off in accuracy. The complex process delays the functioning, impacting the output process time and accuracy Perez et al. (2017). Other challenges focus on MapReduce's fault tolerance implementation, the results of the Map phase to local files before sending them to the reducers Grolinger et al. (2014) which therefore adds high overloads to files adding latency to the processing pipelines. Author Fan et al. (2019) states that the data management layer can be entirely handled by an Hadoop distributed system including the storage , querying and resource management with that the use of algorithms such as XGBoost, decision tree, BP neural network help process massive data quickly.Author has experiment and drawn effectiveness on about 1500 substations from Hubei powergrid, the accuracy rate stated by author is about 93.98 percent and failure rate reduced by 75 percent. However author fairly contributes on Hadoop and its features with no consideration of any other architectures and real-time online streaming data.

2.3 Apache Spark Based Big Data Energy Management Architecture

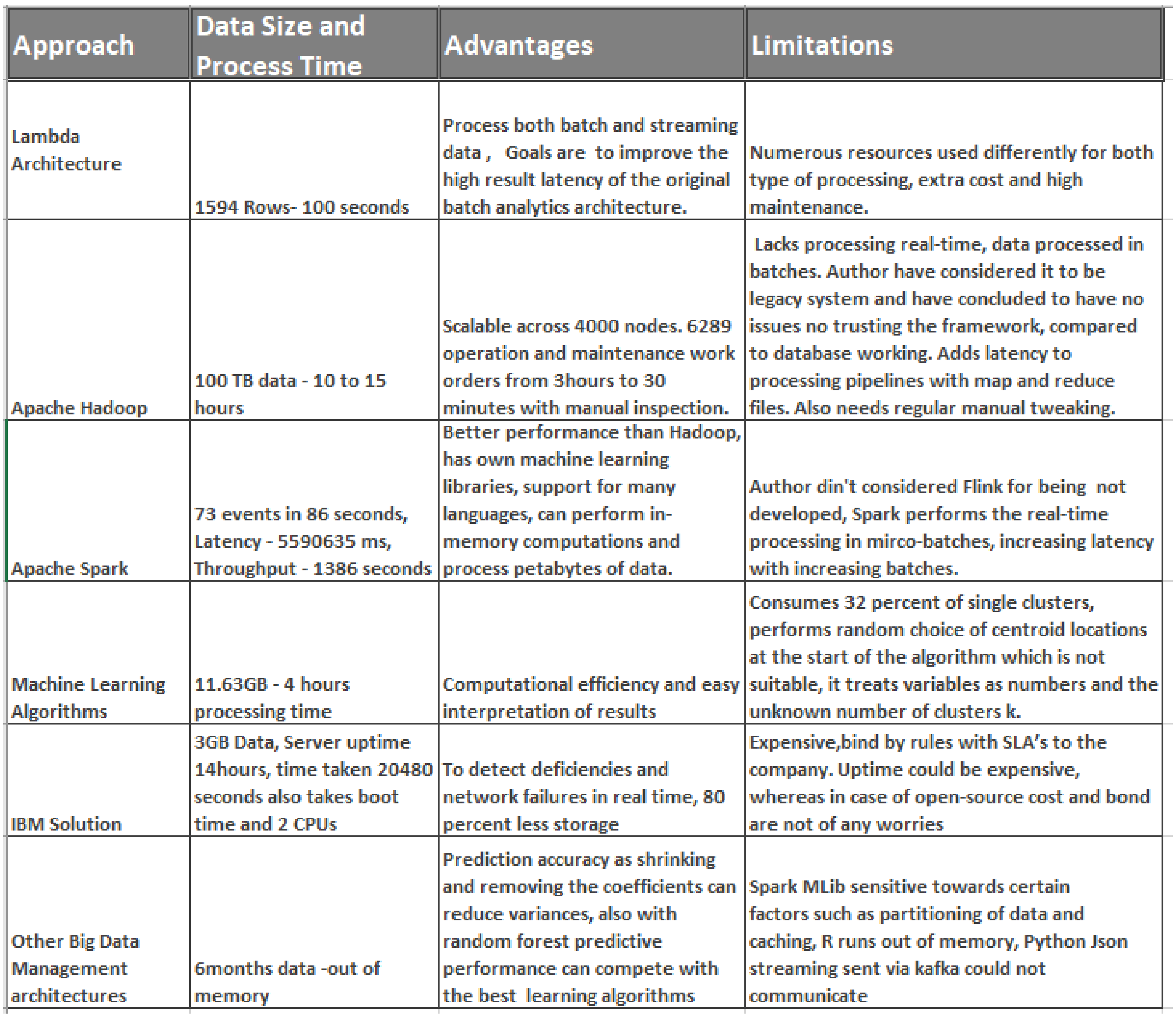
Widely known and highly chosen framework is Apache Spark Framework having larger community.Author R et al. (2018) briefs about the three different types of processing batch, stream and iterative processing.Apache Spark with data received from the smart grids, having an architecture of 16 GB RAM,2 TB hard disk, I7-4790K processor. Data was stored in Apache Cassandra database, which is best suited for time-series data and the streaming data was supplied to Apache Spark with help of Apache Kafka with windowed streaming.Nonetheless, author has not mentioned on the time and the latency been reduced but have concluded on the computation being fault-tolerant, peak-time load balancing and effective with lower latency.

Apache Spark the processing was performed using 8 nodes in comparison to Spark Streaming for 3.6 GB data. Author claims to have recorded throughput using only batch of data as when it comes to measuring the latency every time the batch of data is increased the latency would increase respectively Gibadullin et al. (2019), IBM (2016). The main goal with regards to author was to achieve high throughput, having said that, author have not considered Apache Flink and Apache storm, as incase of Apache Storm it lacked on comparable through-puts, incase of Apache Flink was then under development and lacked some important features.

Author Aziz et al. (2019) has performed a clear comparison of Hadoop and Spark framework and how Hadoop is not capable of processing data in real-time. Its mentioned whole process is performed in batches which puts in attention that despite of data being real-time spark performs them in batches and not real-time making it not a true streaming framework. Approach by Carvalho et al. (2017) is taken into account with having Redis as the storage.Input is taken with Apache Kafka and sent over to Apache Spark and stored in Redis in form of key-value in memory data store. Redis help perform simple and fast distributed I/O. Author Curtis (2018) describes with using different experiments, for word count the latency using SPARK YARN was observed to 5590635 millisecongds and throughput being 1363 seconds only. However author have pointed out few limitations of the working of comparison and the processing performed.The velocity problems of big data, when these data comes from a stream of events are not been considered. Zhang et al. (2018) states this can be solved with help of CEP's sliding window approach, which ensures only a portion of data actually passes into the main memory and old events can be removed or archived yet be that as it may, after an extent the main concern latency for detectingthe complex events is user or system can be observed and the communication cost for communicating with the coordinator ?.

2.4 Big Data with Machine Learning Algorithms

For processing the big data from smart grids author Pérez-Chacón et al. (2018) depicts a study of four Cluster Validity Indices Analysis works parallelly the DB-Dunn, DB-Silhouette, Davies-Bouldin and WSSSE indices.With help of k-means algorithm an op-



Approach	Data Size and Process Time	Advantages	Limitations
Lambda Architecture	1594 Rows- 100 seconds	Process both batch and streaming data , Goals are to improve the high result latency of the original batch analytics architecture.	Numerous resources used differently for both type of processing, extra cost and high maintenance.
Apache Hadoop	100 TB data - 10 to 15 hours	Scalable across 4000 nodes. 6289 operation and maintenance work orders from 3hours to 30 minutes with manual inspection.	Lacks processing real-time, data processed in batches. Author have considered it to be legacy system and have concluded to have no issues no trusting the framework, compared to database working. Adds latency to processing pipelines with map and reduce files. Also needs regular manual tweaking.
Apache Spark	73 events in 86 seconds, Latency - 5590635 ms, Throughput - 1386 seconds	Better performance than Hadoop, has own machine learning libraries, support for many languages, can perform in-memory computations and process petabytes of data.	Author din't considered Flink for being not developed, Spark performs the real-time processing in mirco-batches, increasing latency with increasing batches.
Machine Learning Algorithms	11.63GB - 4 hours processing time	Computational efficiency and easy interpretation of results	Consumes 32 percent of single clusters, performs random choice of centroid locations at the start of the algorithm which is not suitable, it treats variables as numbers and the unknown number of clusters k.
IBM Solution	3GB Data, Server uptime 14hours, time taken 20480 seconds also takes boot time and 2 CPUs	To detect deficiencies and network failures in real time, 80 percent less storage	Expensive,bind by rules with SLA's to the company. Uptime could be expensive, whereas in case of open-source cost and bond are not of any worries
Other Big Data Management architectures	6months data -out of memory	Prediction accuracy as shrinking and removing the coefficients can reduce variances, also with random forest predictive performance can compete with the best learning algorithms	Spark MLib sensitive towards certain factors such as partitioning of data and caching, R runs out of memory, Python Json streaming sent via kafka could not communicate

Figure 2: Summarized Approaches

timal number of clusters are chosen in terms of these indices with help of voting strategy. The above analysis creates 8 with major cluster consuming 39 percent of instances. The execution time for the largest dataset considering big data is less than 4 h.The dataset considered was about 11.63 GB.

Futhermore another paper illustrates the data analytic ARIMA model to process the 5-minutes smart meter data sets for 100 commercial buildings which will help exploring time series as well as the creation of different forecast models. A combination of these forecast models are used to compare performance, author states that ARIMA can produce more accurate forecast than exponential smoothing. Author Xiufeng et al. (2015) puts forth bench marking of data with regards to 5 different platforms which are System C, Hive, Matlab and PostgreSQL. Benchmarking is done on basis of consumption of energy, consumption by buildings, daily consumption of individual houses and private sectors. Daily consumption's are measured with PAR algorithm taking consideration of previous hours and data is been estimated. The benchmarked data was further put to comparison POSTgreSQL and MADLIB being the slowest , the data was about 10GB running on a Intel core processor. According to author with no effort to put having a single node cluster System C is the winner.

The Figure 2 points out the limitations and advantages of existing architecture highlighting the execution time taken to process the data.

3 Methodology

Big data approach on smart grids plays a significant role, various forms require an elastic self-service approach which would not be handled by traditional MapReduce approach. An algorithm such as random forest and LASSO needs to meet formats and just the huge chunk of data passed through would not provide a efficient outcome. Big Data and Cloud Computing inevitably work together, with the scalable nature of cloud and managing spikes without hindrance, and possible disaster recovery makes it a finest for smart meters Big Data.

3.1 Requirement Gathering

This methodology as shown in Figure 3, depicts the steps followed with equipment used for research. The architecture will be deployed on Amazon Elastic Compute cloud(EC2) and the processed data will be stored in Simple Storage Service. Amazon Web service (AWS) instance is selected, as its the "dominant player" in cloud computing states Serrano et al. (2018)adding on to be the very first company to have served cloud services since 2006,considering the growth in the market it makes total sense to go with Amazon Services.

With Smart grids having humongous amount of data, naturally requires an approach to be rapid at real-time and with that a need of batch processing for all the data which would not require real-time streaming therefore helping to avoid overload. Lambda architecture as shown in Section 2 is one of these architecture which performs both but Apache Flink performing both makes it obsolete. Since Apache Flink has a network stack supporting, both low-latency, high-throughput streaming data can perform runtime operators and even batch operations that can work for datasets that can be fetched from the location. As shown in Figure 4 the eco-system was built in Ubuntu environment using version 18.04.02 with 64 bit. The latest version of Apache Flink was used that

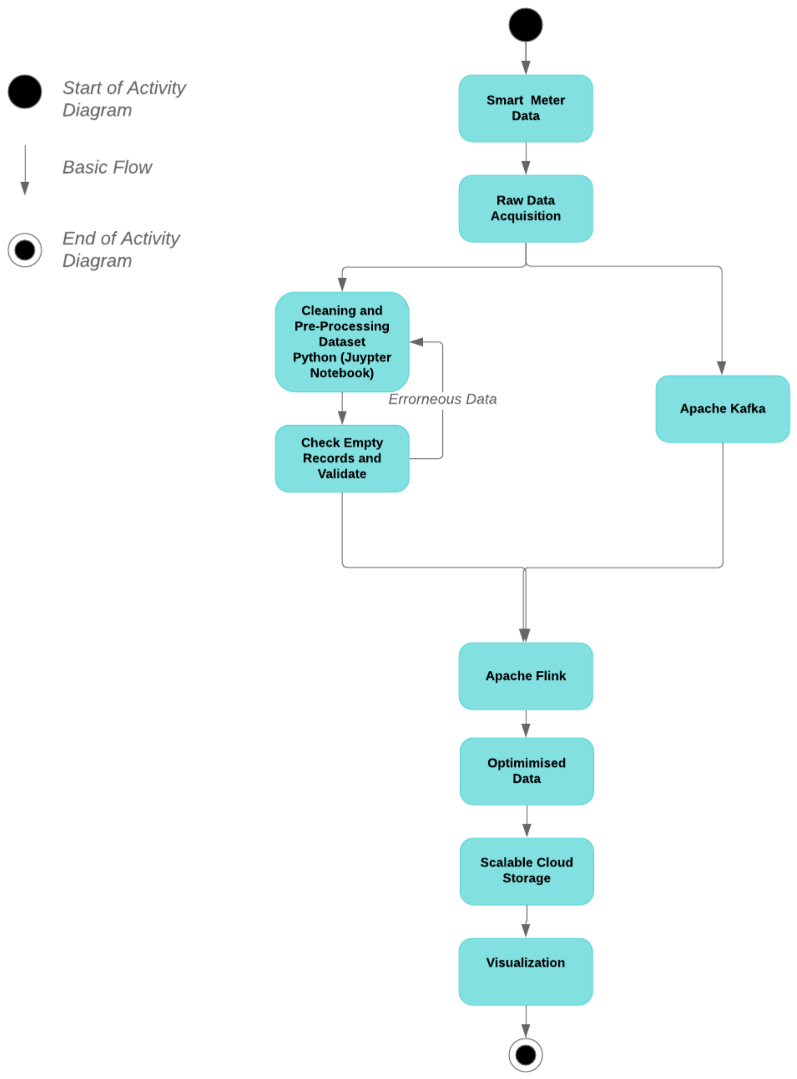


Figure 3: Activity diagram on the flow of proposed architecture

is 1.9.1. The RAM used was about 5GB including 2 processors in an virtual machine that is VMware Workstation. The cloud instance will contain both Apache Kafka and Apache Flink. When it comes to smart grid data, why does it require batch processing and real-time processing both? Historical data could be processed in batches for various such reasons such as forecasting which is important for decision making, wide-range planning on how customers will use energy and then plan utilities. With huge amount of data generated every second certainly needs speedy processing to have an communication built on real-time consumption also displaying it on smart devices. This streaming data needs to be consumed by Apache Flink, having its in-built connector offering Apache Kafka makes it a suitable choice. Kafka is capable of handling high-velocity and high-volume data providing high-throughput, handles messages in millseconds providing lower latency. Making it efficiently fit into the architecture.

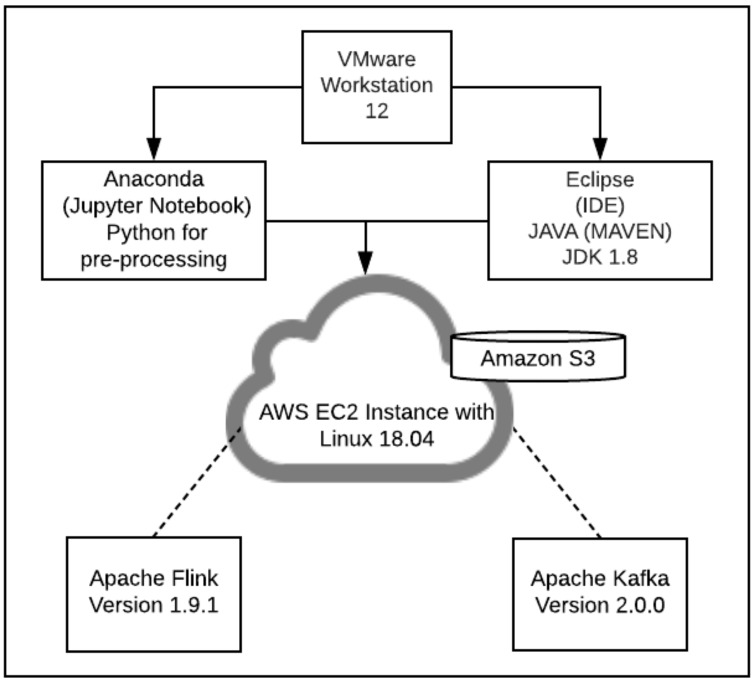
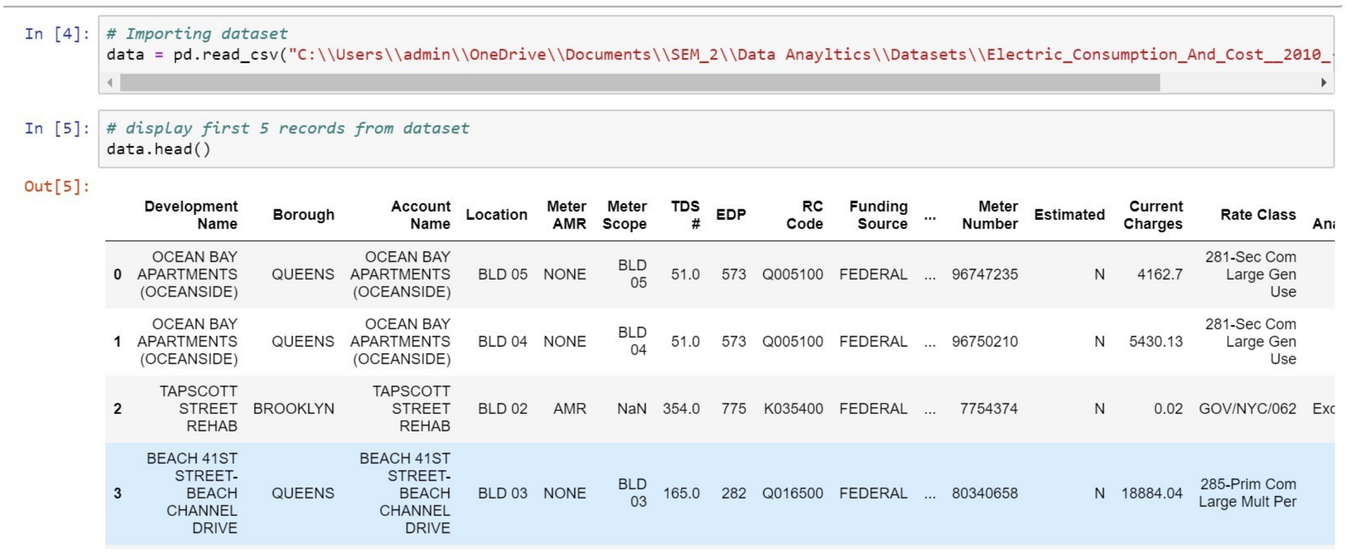

VMware Workstation 12
Anaconda (Jupyter Notebook) Python for pre-processing
Eclipse (IDE) JAVA (MAVEN) JDK 1.8
Amazon S3
AWS EC2 Instance with Linux 18.04
Apache Flink Version 1.9.1
Apache Kafka Version 2.0.0

Figure 4: Requirement Gathered with versions

3.2 Data gathering and Cleaning

Activity diagram on Figure 3 shows the approach with flow of architecture,having to process the data in batches and in order to achieve precision is always better pre-processed Daki et al. (2017). The dataset utilized for the approach is a CSV file ”Electric Consumption and Cost” having data lodged from 2010 to March 2019 about electricity consumption through smart meters from New York City. This data consists of 316579 rows and about 27 rows and provides a brief about the account name, location, Area and Building numbers, the automatic meter reading, funding source and current charges.In case of smart meter failures whether the bill is estimated or no is also mentioned in this dataset. Talking about failures, a disorder of data is a common sight for many such reasons. Hence, de-duping and cleaning is performed to remove the data errors and duplication. Cleaning and de-duping looks out for empty or null data rows and organizes it. The erroneous data will be sent back to be pre-processing again and later discarded by the system in case as no way out. Considering an example of Smart meter data having meter numbers which is unique has data being disorganized, in such cases a random or average value cannot be inserted, the row entirely needs to be discarded. As shown in Figure 6, Figure 6 Python code in Juypter Notebook is been used to perform the same, Sample code below helps drop the column where meter number unique number is not present. Also certain missing values of column Meter AMR are replaced with AMR by taking out the mean of the column, hence replacing with maximum occurred.

```
data.dropna(subset=['MeterNumber'], inplace= True)
data['MeterAMR'].fillna('AMR', inplace=True)
```



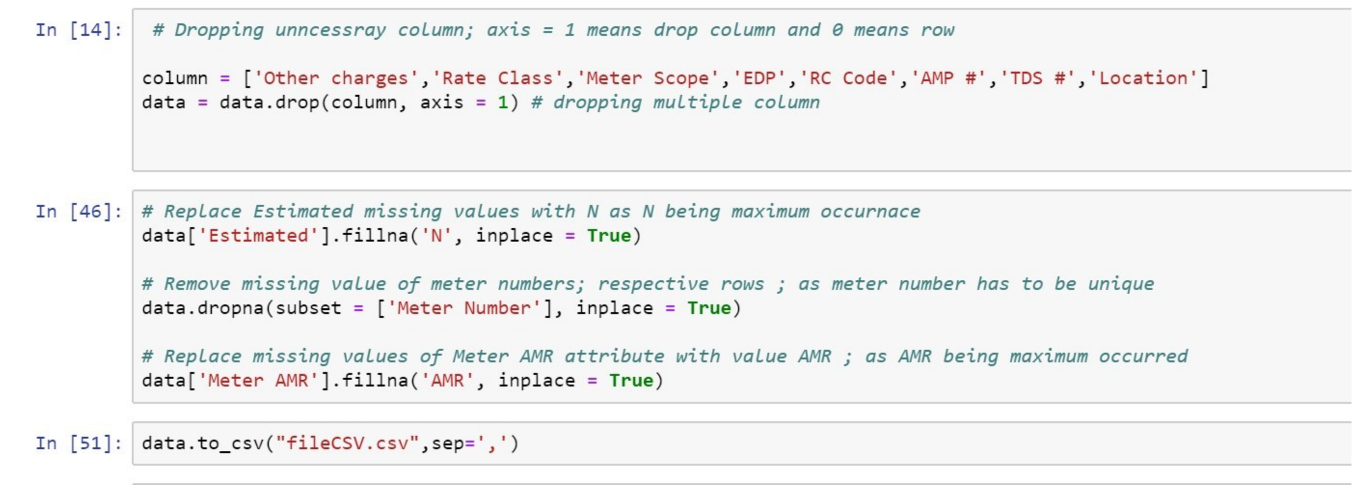
```
In [4]: # Importing dataset
        data = pd.read_csv("C:\\Users\\admin\\OneDrive\\Documents\\SEM_2\\Data Anayltics\\Datasets\\Electric_Consumption_And_Cost__2010_

In [5]: # display first 5 records from dataset
        data.head()

Out[5]:
```

	Development Name	Borough	Account Name	Location	Meter AMR	Meter Scope	TDS #	EDP	RC Code	Funding Source	...	Meter Number	Estimated	Current Charges	Rate Class	Ana
0	OCEAN BAY APARTMENTS (OCEANSIDE)	QUEENS	OCEAN BAY APARTMENTS (OCEANSIDE)	BLD 05	NONE	BLD 05	51.0	573	Q005100	FEDERAL	...	96747235	N	4162.7	281-Sec Com Large Gen Use	
1	OCEAN BAY APARTMENTS (OCEANSIDE)	QUEENS	OCEAN BAY APARTMENTS (OCEANSIDE)	BLD 04	NONE	BLD 04	51.0	573	Q005100	FEDERAL	...	96750210	N	5430.13	281-Sec Com Large Gen Use	
2	TAPSCOTT STREET REHAB	BROOKLYN	TAPSCOTT STREET REHAB	BLD 02	AMR	NaN	354.0	775	K035400	FEDERAL	...	7754374	N	0.02	GOV/NYC/062	Exc
3	BEACH 41ST STREET-BEACH CHANNEL DRIVE	QUEENS	BEACH 41ST STREET-BEACH CHANNEL DRIVE	BLD 03	NONE	BLD 03	165.0	282	Q016500	FEDERAL	...	80340658	N	18884.04	285-Prim Com Large Mult Per	

Figure 5: Raw data upload for transformation



```
In [14]: # Dropping unncessray column; axis = 1 means drop column and 0 means row

         column = ['Other charges','Rate Class','Meter Scope','EDP','RC Code','AMP #','TDS #','Location']
         data = data.drop(column, axis = 1) # dropping multiple column

In [46]: # Replace Estimated missing values with N as N being maximum occurnace
         data['Estimated'].fillna('N', inplace = True)

         # Remove missing value of meter numbers; respective rows ; as meter number has to be unique
         data.dropna(subset = ['Meter Number'], inplace = True)

         # Replace missing values of Meter AMR attribute with value AMR ; as AMR being maximum occurred
         data['Meter AMR'].fillna('AMR', inplace = True)

In [51]: data.to_csv("fileCSV.csv",sep=',')
```

Figure 6: Data Cleaning code performed

3.3 Model Analysis

Having cleaned and pre-processed the data, its further used in the materials gathered. Apache Flink is been installed with help of Ubuntu terminal using Virtual machine VMare Workstation with Ubuntu installed version 18.04. Apache Flink cluster is started written in a shell file inside the bin folder with help of below line of code the cluster can be started.

```
bin/start-cluster.sh
```

Since Flink supports JAVA the most recent version and Scala, a Maven project is created to run under Flink Cluster having the transformation logic as mentioned earlier. Apache Flink needs to support the maven version and its necessary to have it setup with dependencies. Several dependencies from core to compile having streaming, plugins are provided by apache.org that is added into the file called pom.xml. Below is the example of one such dependency which helps it understand the streaming environment

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>
```

To connect Apache Kafka, its installed downloading from the Apache official site and Zookeeper is executed. For Apache Kafka to run its must to execute Zookeeper with help of its properties file present in config folder. Apache Flink and Kafka requires connectors Hesse and Lorenz (2016), hence having version Kafka 2.11 fink-kafka connector is used and added into pom.xml file which is been setup in MAVEN. The KafkaFlinkConsumer help the flow to be started before its necessary to run Apache Kafka with its server properties and be passed with topics to be received from the Flink. Its important to specify the bootstrap server properties in the code, which helps understand Flink on which port is kafka running. In code port "127.0.0.1:9092" is added to pointing to Kafka port at localhost:9092.

```
bin/kafka-server-start.sh config/server.properties
```

Having this JAR file can also be executed in Apache Flink,with help of browser entered with url localhost:8081, the entire running if the file with failures and standard output can be observed. To make the process run much better scalable use of Cloud service is done and EC2 instance is setup using the AWS console. For the storage into cloud API is used to have the processed data to be stored in S3. The bucket name is mentioned in the S3 path of the API and is called through the JAVA code.

4 Design Specification

The architecture as shown in Figure 7 underlie the design and the associated requirements for efficient ways to analyse the Big Data from smart grid at real-time in an improved and efficient manner. The aim of these architecture is to capture the latency and throughput of the Smart Meter data and compare to the frameworks outlined in Section 2, which will answer the research question which forms the centre of this study. To being with batch processing, it will contain data which does not require to be processed at real-time such as data logs, non-electrical marketing data or electrical data sent over in csv format to be billed accordingly, backup data and others. This data can be de-duped or cleaned with

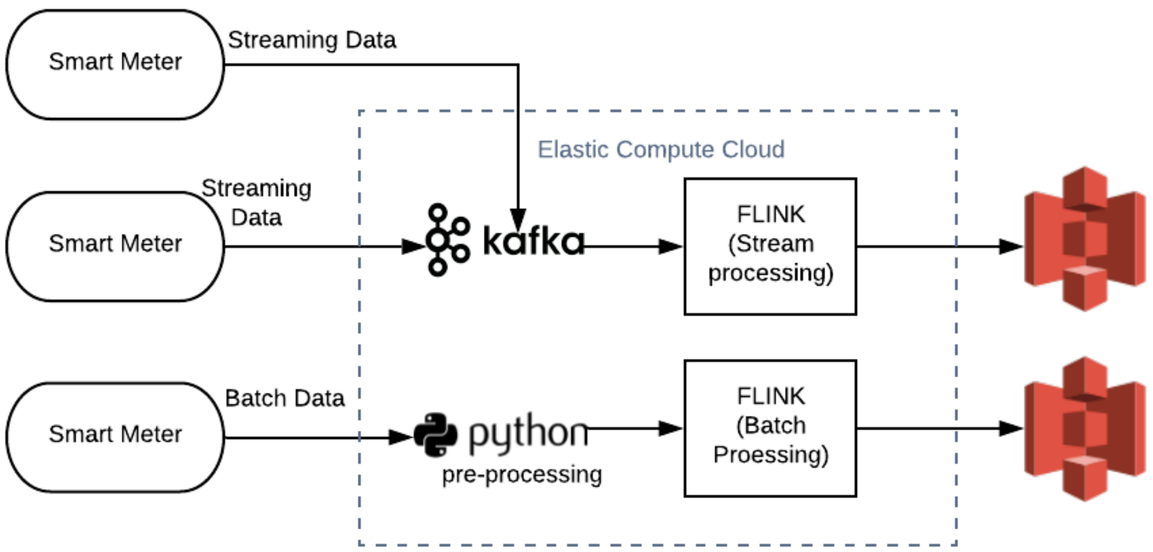


Figure 7: Smart Grid Data Distribution and process

help of machine-learning techniques, a similar developer friendly to do this without having to complicate it is by using Python programming language and performing the same on the environment.Another benefit of batch processing DataSet API is, it retries failed executions it can be configured numerous times, before retiring it as a failed job. The JAR containing the processing code will help look for the file as per the file maintained without having to store it in specific location. Its one of the reasons why Flink has benefits of processing Jar faster, which will be deployed in Amazon EC2, range using 2 virtual CPUs and 500MB ram with t2.medium instance. Once being SSH into the instance ,the jar file can run and help perform using the Apache Flink pre-installed in Amazon EC2 instance. Data once processed will be sent to Amazon S3 for storage creating private bucket, since this data needs to be highly available, and with Amazon S3 having four key factors retrieve data at any time, reliable, cheap and scalable it best to take advantage of the same. The code can be run into the MAVEN JAR file and the data will be sent over to S3 buckets. This JAR file will have special classes Dataset, a class from Apache Flink present to process the data as required.

With real-time streaming, Apache Flink being the true real-time streamer, provides data distribution, communication and fault tolerance for distributed computations over data streams. For this Apache Flink uses DataStream class present in Flink's structure. But first to achieve the real time streaming data Flink uses Apache Kafka. Class KafkaFlinkConsumer records from a topic and periodically checkpoint all its Kafka offsets. This class is attained with help of Flink Kafka connector dependency added. Once the connection is established the data streaming begins and is received by Apache Flink. Flink uses Master-Slave architecture, Job Manager acts as Master,responsible for scheduling multiple Task Managers which are slaves as well as monitoring and deploying them. Once processed returns via sink, the processed data is sent to Amazon S3 similar as performed in batch processing.

5 Implementation

5.1 Supporting Lanaguage and Dependencies

Having MAVEN JAR file created in the local environment which consists Java programming language JDK(1.8), the JAR is later deployed to Amazon EC2 instance cloud environment for better processing than local environment. To elaborate more, Apache Flink only supports JAVA(1.8) the most recent versions of JAVA, therefore, dependency related to JAVA needs to be used. With help of this dependency classes and objects such as Dataset, StreamingEnvrionemnt can be created and batch processing can be used. Core dependencies are in the flink-dist jar which are part of Flink's library folder and container images,which contains the classes like String and List.. With help Flink-Kafka-Connector dependency been added to the pom file, FlinkKafkaConsumer can be used, this consumer helps consuming the data. The FlinkKafkaConsumer version changes according to the Kafka version installed, in case of version 11, the number 011 will be appended to the same.The below code shows the kafka dependency added in pom.xml file

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.11</artifactId>
    <version>1.9.1</version>
</dependency>
```

Other streaming dependencies related to streaming environment are been used , with Maven plugins and assembly plugins with flink-client dependencies. With data common loggins dependencies used to log during runtime.

5.2 Development

5.2.1 Understand Classes and Parameters

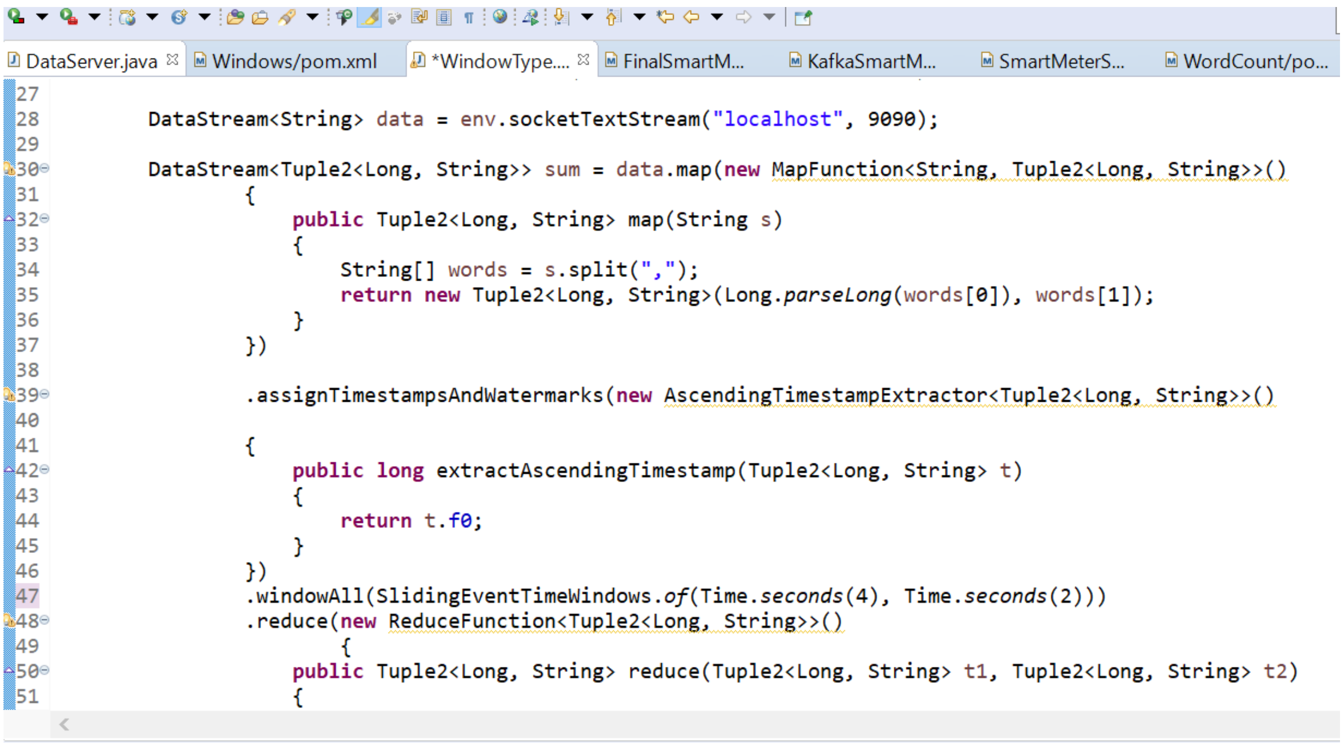
The transformed dataset having null values and duplication's are been put in the folder to be accessed by Apache Flink code with path mentioned in the jar file. This unquestionably saves time, than having it accessed by other framework or databases. The lines of code from CSV flink or text file one line at a time can be read by readTextFile or readCsvFile. With help of ExecutionEnvironment Class, it helps Flink to know where the program needs to be executed. When real-time streams of data coming in Class StreamExecutionEnvironment needs to be used. With help of these both classes, if a program is running on a IDE its executed in local environment, if its running on a cluster its executed on remote environment. The passing of arguments and configurations file will be read with help of ParameterTool. Another important parameter is GlobalJob-Parameter, every node in a cluster is known about the execution been running globally, as in case of failure the node helps resuming from where its stopped. Writing the output out to a file can be done with help of writeAsText.

5.2.2 To achieve latency and throughput

Besides Apache Kafka, Flink has its own in-built way to accept the data with help socket stream, the data can be sent via the port with writing the query on it. The class is SocketTextStream which helps read data from the socket and elements in the data can be separated using the delimiter.

```
DataStream<String> data = env.socketTextStream("localhost", 9090);
```

For this the system socket must be open using the terminal, or even using the java code.This helps have better throughput and lesser latency which can be evaluated using the dataset. There are different types of windowing such as Tumbling Windows, Sliding Windows, Session Windows and Global Windows Flink (2015-2019). Windows starts and ends when the condition referred is met. The Tumbling and Sliding windows are time based windows. To evaluate on basis on time in this implementation will be using Sliding Window. With sliding Window, the next window overlaps the first window with the time mentioned in the code whereas tumbling windows only starts once the 1st window is finished. Taking advantage of this and having the best to utilize the time, Figure 8 shows



```

        DataStream<String> data = env.socketTextStream("localhost", 9090);

        DataStream<Tuple2<Long, String>> sum = data.map(new MapFunction<String, Tuple2<Long, String>>()
                {
                    public Tuple2<Long, String> map(String s)
                    {
                        String[] words = s.split(",");
                        return new Tuple2<Long, String>(Long.parseLong(words[0]), words[1]);
                    }
                })

                .assignTimestampsAndWatermarks(new AscendingTimestampExtractor<Tuple2<Long, String>>()

                {
                    public long extractAscendingTimestamp(Tuple2<Long, String> t)
                    {
                        return t.f0;
                    }
                })
                .windowAll(SlidingEventTimeWindows.of(Time.seconds(4), Time.seconds(2)))
                .reduce(new ReduceFunction<Tuple2<Long, String>>()
                        {
                    public Tuple2<Long, String> reduce(Tuple2<Long, String> t1, Tuple2<Long, String> t2)
                    {
```

Figure 8: Sliding Window Code

the code written for sliding window,with help of sliding window the time mentioned as seconds to slide helps collect the data at that time and process it. The on way of having the latency at minimal is by utilizing the Tuples, Windowing, Apache Kafka and localhost stream connectors with utilizing Event and Time processing.

5.2.3 To acheive fault-tolerance

To achieve an improved processing architecture, this project also focuses on checkpointing. Checkpointing as stated earlier, helps save the snapshot to recover in time of failure. Its not default enabled and hence will be enabled with code, the time of 100seconds is passed for checkpointing, which will help save the state of the project at 100th second and can be observed in the local cluster using the browser. One way of achieving the checkpointing is through states, which helps to manage historic data allows efficient access to past events. With all this one of the major key factor of state is to convert stateless transformation to stateful transformation.The evaluation is performed with help of FlatMAp operations for the same to understand how a state can perform statless operations. Every state can be saved by passing the time in milliseconds and a checkpointing timeout can be added to limit to a certain time. Here in code, the timeout time is taken to be 10000 milliseconds. At cases where the data is huge, concurrent checkpointing can be set multiple numbers, where the number mentioned will help running the checkpointing at the same time. Certain snapshots can also be retained, with Flink having the functionality to delete the snapshot once the job is cancelled. Any snapshot to be retained on cancel can be persisted with help of ExternalizedCheckpointCleanup.

This approach also shows the restart strategy which will allow a job execution for a certain time and number of times, a fixedDelayStartegy is used in this approach which will help to take delay time that will be mentioned as the arguments. once the job exceeds that time with help restart strategy checkpointing it can be restarted and the attempts can be mentioned for how many times will the job be executed. Due to this case , efficiency can be obtained having to perform a certain job only for specific amount of time and incase of failures will be re-attempted.

5.3 Flink Interface and Analysis

The architecture consumes data on Kafka topics from the JAR file, app runs the data through several operations. The execute() call is when Flink starts processing, with help of run command from the terminal. The local cluster can be visible with help of browser at localhost:8081, shows everything up and running and failures logged. It helps understand the running jobs and time take to execute the job. Every case study experimented to understand the execution time and throughput will be helpful with the local cluster. With windowing performed the data is transformed and reduced having grouped into pairs, the 1st window has processed about 5 data counts in 0 to 5 seconds, the next window having the same processing speed processes about 7 data counts, with extra being the 2 seconds of sliding window time. Figure 9 shows the overview of test batch processing performed

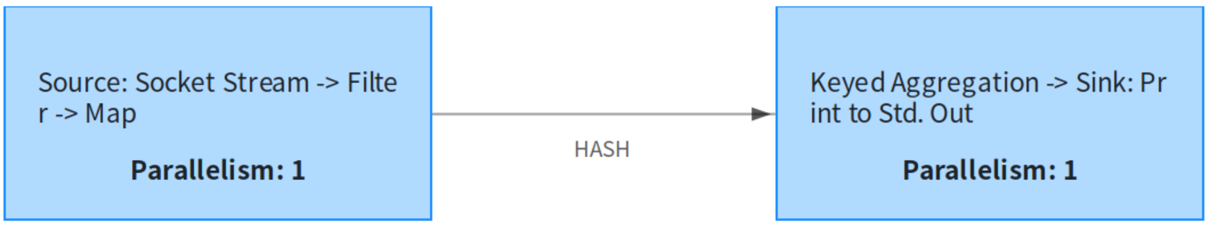


Figure 9: Overview of batch processing

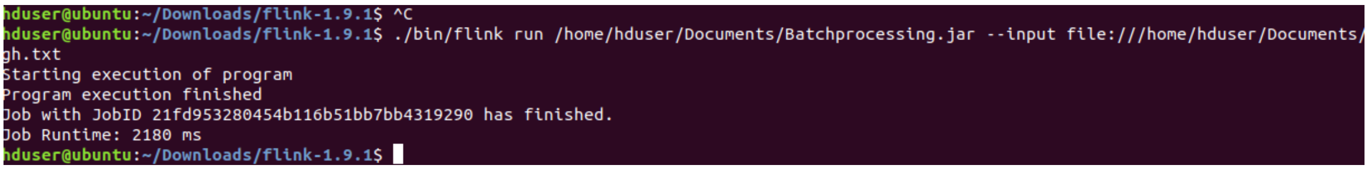
at intial stage to understand the working of web browser client request via localhost.It explains the stream received via socket and HashMap is performed on the data, forming keyed data finally to sink the output to standard output file. The overall study shows the 2 CPU cores been utilized, the physical memory of 5.55GB was present, JVM Heap Size 992MB and Flink Managed Memory about 642MB. The

6 Results and Critical Analysis

To evaluate the performance of the architecture the following case studies are taken up which will be briefly discussed. The experiments are performed using the dataset which is transformed, as mentioned is the above section 3.2

6.1 Case Study 1 : Filter Operator

Check the total number of Borough named QUEENS overall.
To achieve the above from the dataset, the Filter operator is applied to look out for QUEENS in the borough list. The output of this code provides the count of 36987 total Queens present in the dataset selected. Filter operator takes an interface called filterFuntion as argument and the parameters of datatype being string in this case will be added.The execution time to evaluate the same is an average of 2180ms7 after being run the jar for total 4-5 times and take average of all the 5 readings. The time taken at intial execution was more, but the time for execution was not taken up by flink to process, processing was in less than 1 second, the time to read the data from the system took about 2 seconds in all. The moment Flink recognizes the system files, the next execution was carried out in much lesser time. Figure 10 shows the execution time taken by the



```
hduser@ubuntu:~/Downloads/flink-1.9.1$ ^C
hduser@ubuntu:~/Downloads/flink-1.9.1$ ./bin/flink run /home/hduser/Documents/Batchprocessing.jar --input file:///home/hduser/Documents/
gh.txt
Starting execution of program
Program execution finished
Job with JobID 21fd953280454b116b51bb7bb4319290 has finished.
Job Runtime: 2180 ms
hduser@ubuntu:~/Downloads/flink-1.9.1$
```

Figure 10: Execution time of Filter operator for Batch Processing

JAR file. As shown in Figure 12 the very first row stating word count depicts latency and throughput for both Apache Spark from existing approaches and Apache Flink carried out now.

6.2 Case Study 2 : Aggregate Function using Streaming Environment

Check Maximum and Minimum consumption done by which Smart Meter and is from which Borough. The dataset contains 4 tuples required and the path of the file is placed as an argument into the jar file. The aggregate function is applied on the batch file data which acts as streaming data when received and works under the streaming execution environment. The execution time taken to process this code is around 18306ms and having sent about 53MB data the throughput was observed to be 17586.77 which is quite higher to that of Spark showing a positive outcome. Below chart Figure 11 shows the execution of Apache Flink code when ran under Flink Cluster.

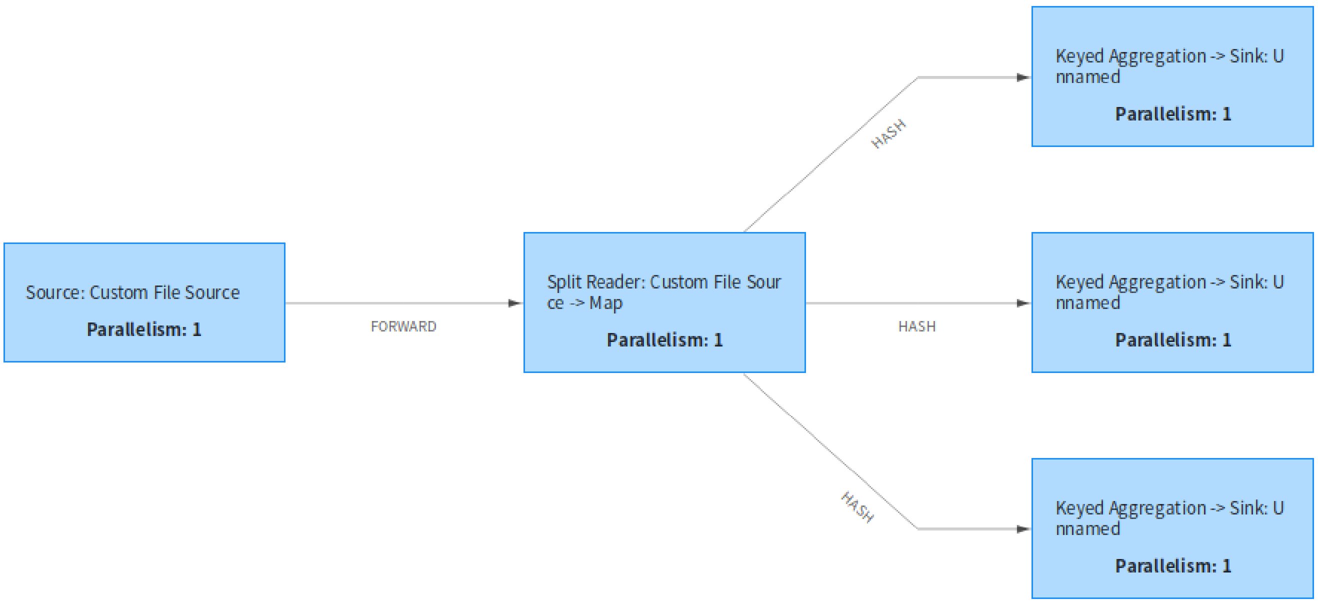
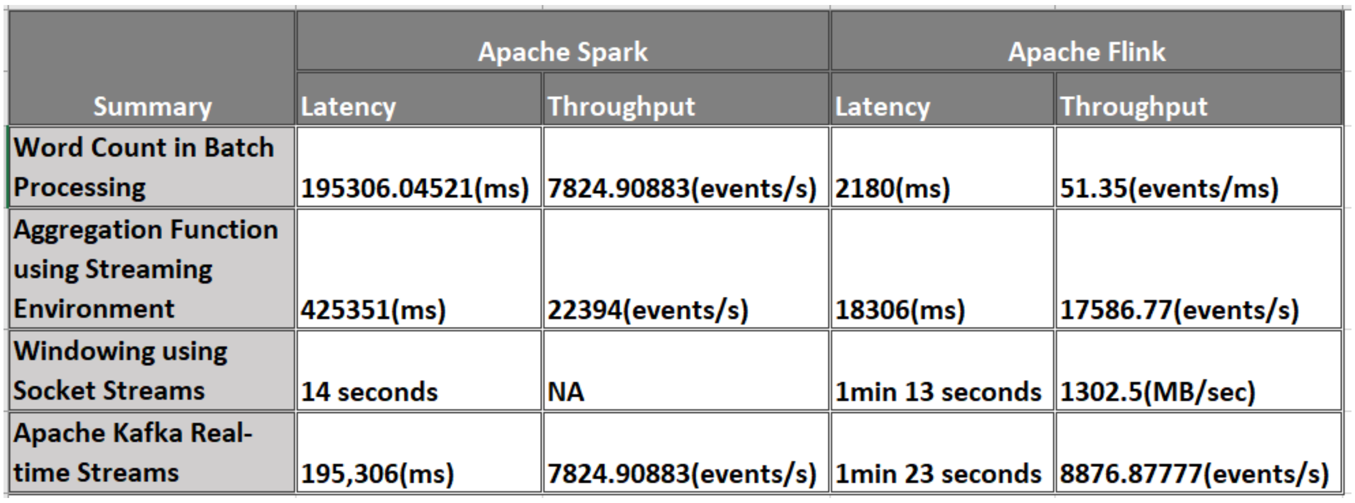

Source: Custom File Source
Parallelism: 1
FORWARD
Split Reader: Custom File Source -> Map
Parallelism: 1
HASH
HASH
HASH
Keyed Aggregation -> Sink: Unnamed
Parallelism: 1
Keyed Aggregation -> Sink: Unnamed
Parallelism: 1
Keyed Aggregation -> Sink: Unnamed
Parallelism: 1

Figure 11: Overview of aggregate function under streaming environment

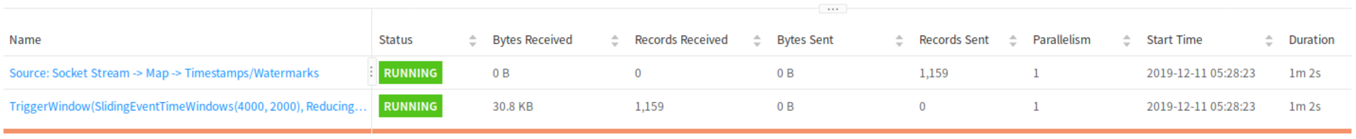


Summary	Apache Spark		Apache Flink	
	Latency	Throughput	Latency	Throughput
Word Count in Batch Processing	195306.04521(ms)	7824.90883(events/s)	2180(ms)	51.35(events/ms)
Aggregation Function using Streaming Environment	425351(ms)	22394(events/s)	18306(ms)	17586.77(events/s)
Windowing using Socket Streams	14 seconds	NA	1min 13 seconds	1302.5(MB/sec)
Apache Kafka Real-time Streams	195,306(ms)	7824.90883(events/s)	1min 23 seconds	8876.87777(events/s)

Figure 12: The Summary of the observed latencies and throughput

6.3 Case Study 3 : Sliding Window using SocketStream

As explained earlier,the sliding windows utilizes the time mentioned in the SlidingProcessingTimeWindows as time of two seconds to slide to even process the data. The reduce function takes in the tuples of data as per the dataset provided. the output of the data shows the groped up cities and counts having the consumption from the dataset. Here , the major factor is socketstream , where the streams of data are sent via the port opened in the system. In this case port 9090 is opened and the JAVA code is written to send the file data into the socket, which will act as the streaming data once received by the Apache Flink. The reduce function is later mapped using the map function and the output is written in the text file using writeToText command. The output clearly present, every window having processed data count is increased on going further below. And the execution time taken to process is 1 minute and 13 seconds and the throughput observed was 17586.77 events per second. Here the throughput is calculate as the data record sent per second.As Figure 13 depicts the records sent were about 1159 and the time taken was 1min 2sec , the job was still running and every record increased within the window the latency increased. The final time for whole execution was 2mins 13 seconds. With



Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism	Start Time	Duration
Source: Socket Stream -> Map -> Timestamps/Watermarks	RUNNING	0 B	0	0 B	1,159	1	2019-12-11 05:28:23	1m 2s
TriggerWindow(SlidingEventTimeWindows(4000, 2000), Reducing...	RUNNING	30.8 KB	1,159	0 B	0	1	2019-12-11 05:28:23	1m 2s

Figure 13: Sliding window code status

that referring to Figure 12 its observed that the throughput with regards to windowing in Apache Spark is higher than that of Apache Flink, and its vice versa for latency being higher for Apache Spark.

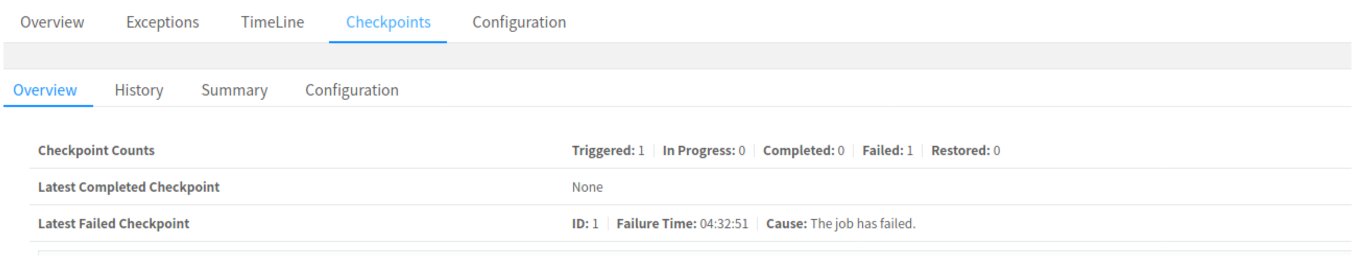
6.4 Case Study 4 : Apache Kafka real-time streaming

Apache Kafka is added to the JAR file using addsource method present in Apache Flink and with help of Flink-Kafka-connectors dependency.This helps Apache Flink to use using the producer and consumer, where the consumer helps to consume data from KafKa by Flink. the Before running the JAR file under Flink in Ubuntu terminal, its necessary to run the zookeeper for Apache Kafka. After which Kafka is executed using the server properties file present in config folder. The bootstrap server property present specifies on which port is kafka running. Once both of them are set up, the jar can be be executed which requires execute command in the code to be present. Time taken for execution is around 1 min 23 seconds and throughputs observed as per records processed per second. About 8876.8777 events/second can be observed with Spark being lower to 7824.90883 events/sec and latency 195.306ms.

6.5 Case Study 5: Maintain state and FaultTolerance

With help of Apache Flink code, a stateless transformations can be converted to stateful transformation due to which a previous value can be maintained even being the statless operator, wherein only present operations are performed. Here , the state of FlatMapFunction being statesless operator will be maintained. For this class has to be extended to richFlatMapFunction class helps accessing 4 methods getRuntimecontext , open ,close and setRuntimeContext. Being the value state, valueStateDescriptor is setup, which will be provided with name of the state, datatype maintaining the state and default initialized value. Every time a program is run a logic is presented which will always have the current number firstly intialized as 0, a randomly generated number comes in it checks for the existing number state and adds to it the new generated number, thereby maintaining the current value each time a new number is coming in and adding to it.

Below Figure 14 shows the number of checkpoints trigerred and its state of success or failure. The whole Checkpointing logic as mentioned in 5.2.3 to achieve the fault-tolerance is enabled in the code, which will create a snapshot every 100 milliseconds, any checkpoint failure will be only running until 10000 milliseconds. Its made sure that only one Checkpointing runs at a time and will be retained on cancellation. Also a checkpoint on failure can be restarted for 3 attempts and can have a delayed restart of 10 milli-seconds.



Checkpoint Counts	Triggered: 1 | In Progress: 0 | Completed: 0 | Failed: 1 | Restored: 0
Latest Completed Checkpoint	None
Latest Failed Checkpoint	ID: 1 | Failure Time: 04:32:51 | Cause: The job has failed.

Figure 14: Total Checkpoints triggered and failed

A comparison of test was performed between the architecture proposed in this paper using Flink and approaches mentioned in section 2. Apache Hadoop and Lambda Architecture becomes obsolete when it comes to Apache Flink features of executing real-time big data. The platform which dominates and as a cut throat competetion with Apache Flink is Apache Spark. Considering the mean latency values for each case study executed 4-5 times and have taken up the average value mentioned in the table Figure 12. With these case study is performed to understand the variance. The study shows Flink out-performs Spark in terms of having higher throughput , lower execution time and also being fault-tolerant at the same. The real-time streaming data from sockets or Apache Kafka is better working in Flink as compared to Spark However the comparison had to be done with Flink limited resources such as without using YARN cluster, had Flink worked the same way being on YARN cluster as SPARK could have been a possibility. However its quite clear that Flink running on the local environment and cloud environment performs significantly better than Spark and the execution time showing the difference certainly cannot be ignored. A stateless transformations can be maintained and incase of failure having Checkpointing features in Flink, with that every snapshot is maintained as per time mentioned and incase of failure it can be restarted making the architecture fault-tolerant.

6.6 Discussion

The above experiments help understand the three various designs Apache Flink have in store and there execution time in an efficient manner. It has its own modelling and transformation of data which proved to be an important factor. The first experiment is based on simple batch processing , later Windowing with help of local stream input which is compared with micro-batching of Apache Spark and finally taking in real-time data with Apache Kafka. The study clearly shows the execution time being alot lesser than when compared to existing approaches and Apache Spark.

However there are few limitations with this study such as running the experiment on many clusters with huge, Terabytes of data was not performed. In many frameworks as observed the efficiency varies with increasing clusters and size of the data, the throughput decreases, this could not be recorded in this approach.A couple of features with respect to Kafka and Flink packages like metrics and machine learning libraries for data pre-processing and unsupervised learning have also been tried but were unfortunately not applicable on these data due to its limitations and incompatibility with the execution environment. Amazon cloud provides EMR(Elastic MapReduce) as a service, with which the need to install the Flink on an EC2 instance is eradicated and provides a clear environment, inducing a large amount of data for both Apache Spark and Apache Flink, having access to Amazon EMR would be also one of the ways to compare the framework's with there efficiency in cloud using the following code but was beyond the scope of this work.Finally, two of the framework configuration study were not tested using Flink YARN cluster as with the results acquired from Spark were on a YARN cluster. Having Flink on YARN would have increased the latency, however Apache Flink shows a significant difference when its run on a local and cloud environment, having outperformed Spark in latency and throughput at majority and also being largely fault-tolerant at the same time.

7 Conclusion and Future Work

The proposed architecture is based on the analysis of Smart Grid Big Data with Apache Flink and also an comparison with massively known frameworks Apache Spark and Hadoop in order to achieve better and improved processing efficiency. The first section points out how Big Data relates to smart grids and further showing the existing approaches having there execution time there limitations and advantages. Although, these are promising solutions a noticeably delay can be seen with outputs processed and data loss can be observed. This paper tries to evaluate how Apache Flink can be used for both batch and real-time processing using Aggregator functions, Apache Kafka and Windowing using the socket streaming. A better execution out of that is noted and compared to existing frameworks. Having achieved the execution time for batch processing is equal to 1432 milliseconds. For socket streaming it was observed to be less than 40 seconds. An optimized way using the reduce function, event and time processing and also Checkpointing is carried under this project making sure there is no data loss and failure can be recovered with time mentioned in the code. Furthermore the processed data is collected and stored in cloud scalable services Amazon S3 using S3 application programming interface. The eco-system presented was setup on Cloud Computing platform that is Elastic cloud Compute. Further making the test validation for the solution, which observes a rise in processing efficiency, with lesser execution time and being fault-tolerant with minimum resources and no such lags in processing the data.

Therefore future work can focus more on using other features of Apache Flink such Watermarks and Lateness helping data to be more precise and efficient also having to work with larger data streams through smart meters which can be a use to business and government dealing with smart grids and meter data wanting results at real-time. With that the architecture have used combination of data analytic tools to perform the desired pre-processing, de-duping, visualization. A single developed JAR file which can perform all of these operations as a whole in addition to using Apache Flink and Apache Kafka its own connectors and in-built functions will be also looked into as future work. Big Data and Smart Grid are one reasons for evolution, certainly going hand-to-hand being a powerful combination and its is necessary to have developer friendly, better processing efficiency, low latency software for real-time streaming data from smart meters.

References

Acharjya, D. P. and P, P. A. (2014). A survey on big data analytics: Challenges, open research issues and tools, *International Journal of Advanced Computer Science and Applications 7(2):511-518* **3**(37): 9.

Aziz, K., Zaidouni, D. and Bellafkih, M. (2019). Real-time data analysis using spark and hadoop, *2018 4th International Conference on Optimization and Applications (ICOA)* **18**(4): 6.

Carvalho, O., Roloff, E. and Navaux, P. O. A. (2017). A distributed stream processing based architecture for iot smart grids monitoring, *2017 IEEE* **6**(25): 6.

Curtis, J. (2018). A comparison of real time stream processing frameworks, *Dublin Institute of Technology, Ireland* **2**: 107.

Daki, H., Hannani, A. E., Aqqal, A., Haidine, A. and Dahbi, A. (2017). Big data management in smart grid: concepts, requirements and implementation, *Journal of Big Data* **4**(28): 19–22.

Fan, L., Li, J., Pan, Y., Wang, S., Yan, C. and Yao, D. (2019). Research and application of smart grid early warning decision platform based on big data analysis, *2019 4th International Conference on Intelligent Green Building and Smart Grid* **3**: 19.

Flink (2015-2019). Apache flink® — stateful computations over data streams, *Official Website* **1**.

Gibadullin, R., Baimukhametova, G. and Perukhin, M. (2019). Service-oriented distributed energy data management using big data technologies, *2019 International Conference on Industrial Engineering* **1.5**(15): 7.

Grolinger, K., Hayes, M., Higashino, W. A., L'Heureux, A., Allison, D. S. and Capretz, M. A. (2014). Challenges for mapreduce in big data, *2014 IEEE World Congress on Services* **6**(24): 8.

Heidrich, J., Trendowicz, A. and Ebert, C. (2016). Exploiting big data's benefits, *IEEE Software;2016* **4**(38): 6.

Hesse, G. and Lorenz, M. (2016). Conceptual survey on data stream processing systems, *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS);2015* **1**: 6.

Horban, V. (2018). A multifaceted approach to smart energy city concept through using big data analytics, *IEEE First International Conference on Data Stream Mining Processing (DSMP)* **7**(2): 4–5.

IBM (2016). Managing big data for smart grids and smart meters, *White Paper* **1**(26): 19.

Kamstrup (2019). The value of data, *Blog* **1**(4): 5.

Kiran, M., Murphy, P., andJon Dugan, I. M. and Baveja, S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing, *2015 IEEE International Conference on Big Data* **4**(19): 8.

Krüger1, N. and Teuteberg2, F. (2015). From smart meters to smart products: Reviewing big data driven product innovation in the european electricity retail market, *Lecture Notes in Informatics* **1**(3): 14–15.

Li, H. H. Y. W. T.-S. C. X. (2014). Toward scalable systems for big data analytics: A technology tutorial, *IEEE Access;2014* **3**: 36.

Merla, P. and Liang, Y. (2018). Data analysis using hadoop mapreduce environment, *2018 IEEE International Conference on Big Data* **6**(22): 5.

Munshi, A. A. and Mohamed, Y. A.-R. I. (2018). Data lake lambda architecture for smart grids big data analytics, *IEEE Access 2018* **6**(20): 9.

Perez, D. J. F., Birke, D. R. and Chen, D. L. Y. (2017). On the latency-accuracy tradeoff in approximate mapreduce jobs, *IEEE INFOCOM 2017* **6**(23): 9.

Prathik, M., Anitha, K. and Anitha, V. (2018). Smart energy meter surveillance using iot, *2018 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)2018* **2**(37): 4.

Pérez-Chacón, R., Luna-Romera, J. M., Troncoso, A., Martínez-Álvarez, F. and Riquelme, J. C. (2018). Big data analytics for discovering electricity consumption patterns in smart cities, *Concurrency and Computation: Practice and Experience* **2**(1): 19.

R, S., Ganesh, B., Kumar, S., Poornachandran, P. and K.P., S. (2018). Apache spark a big data analytics platform for smart grid, *Procedia Technology, 21 (2015)* **6**(13): 8.

Serrano, N., Gallardo, G. and Hernantes, J. (2018). Infrastructure as a service and cloud technologies, *IEEE Software;2018;32* **1**(1): 7.

Shrivastava, G. and Shrivastava, S. (2018). Analysis of customer behavior in online retail marketplace using hadoop, *International Journal of Innovative Research 2017* **6**(21): 9.

Vaidya, M. and Deshpande, S. (2019). Distributed data management in energy sector using hadoop, *2015 IEEE Bombay Section Symposium (IBSS);2015; ; ;10.1109/IBSS.2015.7456653* **1**(17): 6.

Xiufeng, L., Lukasz, G., Wojciech, G. and F., I. I. (2015). Benchmarking smart meter data analytics, *Open Proceedings)* **3**: 19.

Zhang, Y., Huang, T. and Bompard, E. F. (2018). Big data analytics in smart grids: a review, *Energy Inform* **46**(5): 23–24.

Configuration Manual

MSc Research Project
Cloud Computing

Supritha Shetty
Student ID: X18163009

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Supritha Shetty
Student ID:	X18163009
Programme:	Cloud Computing
Year:	2019
Module:	MSc Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	12/12/2019
Project Title:	Configuration Manual
Word Count:	858
Page Count:	4

Signature:	
Date:	12th December 2019

Configuration Manual

Supritha Shetty
X18163009

1 Introduction

The configuration manual refers to the process of understanding a architecture in a systematic manner and helps maintain integrity over time. Manual helps understand the the flow, the working and steps followed. Provides detailed descriptions about the delivered features including use cases, implementation, and configuration specifics. (Hlova; 2017)

Contents

1.1 Purpose of this manual

Purpose of this manual is to understand the configurations and setups required with respect to research Project. The structure followed and the environments used. Provides an overview of all the framework used and its installation process. Further running the program to achieve the objective.

1.2 Evaluation Performed

This research project is majorly focused on improving the efficiency of streaming data received from Smart grids at real-time. The paper touches both real-time processing and batch processing, and with that provides an architecture with minimum resources being tolerant and performs processing at minimal execution time. To understand the efficiency and to compare with existing approaches certain experiments are performed as stated below

- Filter Operator
- Aggregate Function using Streaming Environment
- Sliding Window using SocketStream
- Apache Kafka real-time streaming

To perform the experiments requires to follow installation as shown in Section 3.

2 Environment Setup

2.1 Local Environment

To follow the installation of Apache Flink and Apache Kafka in a local environment requires a VMware Workstation installed with 64bit Ubuntu 18.04

VMware Workstation Setup	
Memory	5.8 GB
Processor	2 CPUs
HardDisk	30 GB
JDK	1.8

Once the setup is done, Eclipse IDE is installed from the official site supporting Ubuntu 64bit to perform the coding.

2.2 Cloud Environment

Create an EC2 instance using AWS console. Launch your instance using the "Launch" button. Create key-value pair, provides you a pem file which will help to access into cloud environment using SSH Amazon (2017a). Once the instance is launched the jar file can be sent over using the Ubuntu terminal via SSH providing the pem file as the key and the password set. Similar steps and configurations have been followed in EC2 instance as that of local environment.

For storage , Amazon service S3 is used. Create a bucket, provide the name and the region required which will be later used to setup API and the object will be sent via the code Amazon (2017b)

3 Installing required Frameworks

3.1 Apache Flink

For Apache Flink to be installed, a tar file needs to be downloaded of most recent version 1.9.1, using the url below

```
https://flink.apache.org/downloads.html#apache-flink-191=-
```

Once the tar file is downloaded , it needs to be unzipped using the following command

```
tar -xvzf flink-1.9.1-bin-scala_2.11.tgz
```

Go inside flink-1.9.1 folder after its unzipped and start the cluster using below command before executing the code. Once the code is executed you can run the stop command. Also to run the a jar file command run needs to be used as shown below.

```
bin/start-cluster.sh

bin/stop-cluster.sh

./bin/flink run /*path of the jar*/
```

3.2 Apache Kafka and Zookeeper

For Apache Kafka to be installed, a tar file needs to be downloaded of version 2.0.0 having scala version 2.11, using the url below and unzip the file. The 2.0.0 is a stable version with Apache Zookeeper supported Apache (2017).

```
https://kafka.apache.org/downloads
tar -xvzf kafka_2.11-2.2.0
```

After being unzipped start the zookeeper inside the kafka folder using below command and run the Kafka server. The topic needs to be sent via terminal using the topic name listed in the code only when there is consumer sending the file data after running the jar.

```
cd kafka_2.11-2.0.0/
bin/zookeeper-server-start.sh config/zookeeper.properties
bin/kafka-server-start.sh config/server.properties
```

4 Dataset and Code Repository

Dataset used belong to smart meters and from the main csv file is divided into txt files for each of the evaluation performed.Both dataset and Code Repository used to evaluate the performance of the architecture is attached in the zip file submitted to the ICT solution link in moodle.

5 Getting Started

The JAR files inside the folder submitted has the code present for each type of testing. It can be ran one by one to under Flink server using the run command. To run the batch processing JAR requires input file and output file, below is the command for the same. Also before running the jar file, the path mentioned in the code will need to be changed.

```
.bin/fink run /*path of the jar*/batchprocessing.jar -input file:///*path
to file*/boroughpresent.txt -output file:///*path to output*/
```

Rest of two jar can be ran using the below run command.

```
.bin/fink run /*path of the jar*/
```

For Apache Kafka jar to run, once the flink, zookeeper and kafka server are started, it must send the broker list using topic name below is the command to hit in the terminal.

```
 bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

Opening localhost:8081 in browser will help understand the Apache Flink running progress of each jar file which is been run under the server. Giving out exact accuracy of execution time for each function performed with heap Size, Flink memory size consumed and other such information about completed and running jobs.

References

Amazon (2017a). Amazon ec2.
URL: *https://aws.amazon.com/s3/*

Amazon (2017b). Amazon s3.
URL: *https://aws.amazon.com/ec2/*

Apache (2017). Apache kafka and downloads.
URL: *https://kafka.apache.org*

Hlova, M. (2017). A user manual or user guide.
URL: *https://stepshot.net/a-user-manual-or-user-guide-how-to-name-a-document-with-instructions/*