Final Project Report

Real-Time Data Pipeline with Apache Kafka and Spark

CSP554 – Big Data

May, 2020

Siri Chandrashekar	A20435389
Bhuvnesh Tejwani	A20444878
Anwesha Kakoty	A20433149
Gurunath Reddy	A20443036

1. Introduction & Background Information

A data pipeline is software that enables the smooth, automated flow of information from one point to another. This software prevents many of the common problems that the enterprise experienced: information corruption, bottlenecks, conflict between data sources, and the generation of duplicate entries.

Streaming data pipelines, by extension, are data pipelines that handle millions of events at scale, in real time. As a result, you can collect, analyze, and store large amounts of information. That capability allows for applications, analytics, and reporting in real time. The first step in a streaming data pipeline is that the information enters the pipeline. Next, software decouples applications, which creates information from the applications using it. That allows for the development of low-latency streams of data.

1.1. Apache Kafka:

Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies. Apache Kafka is *a* distributed streaming platform.

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

1.2. Apache Spark:

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance

Spark Streaming is a Spark component that enables the processing of live streams of data. Live streams like Stock data, Weather data, Logs, and various others.

2. Literature Survey

2.1. Apache Kafka:

In the last few years, there has been significant growth in the adoption of Apache Kafka. Current users of Kafka include Uber, Twitter, Netflix, LinkedIn, Yahoo, Cisco, Goldman Sachs, etc.

Kafka is a scalable pub/sub system. Users can publish a large number of messages into the system as well as consume those messages through a subscription, in real time.

Most of the companies use Kafka as a central place to ingest all types of data in real time. The same data in Kafka is then fed to different specialized systems. We refer to this architecture as a stream data platform as depicted in the figure below. Adding additional specialized systems into this architecture is easy since the new system can get its data by simply making an extra subscription to Kafka.

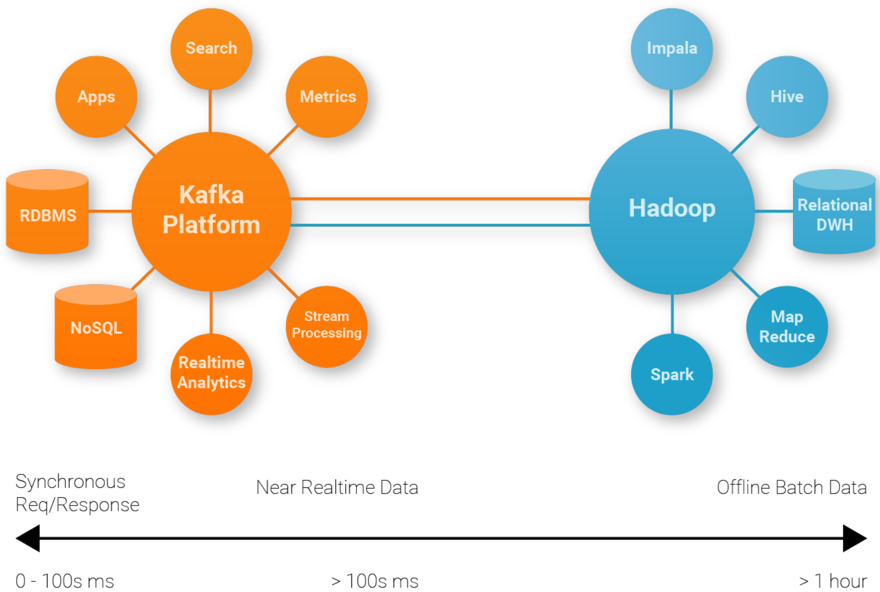


Fig:1 Stream data platform

Kafka can be used to feed fast lane systems (real-time and operational data systems) like Storm, Flink, Spark streaming, and your services and CEP systems. Kafka is also used to stream data for batch data analysis. Kafka feeds Hadoop. It streams data into your big data platform or into RDBMS, Cassandra, Spark, or even S3 for some future data analysis. These data stores often support data analysis, reporting, data science crunching, compliance auditing, and backups.

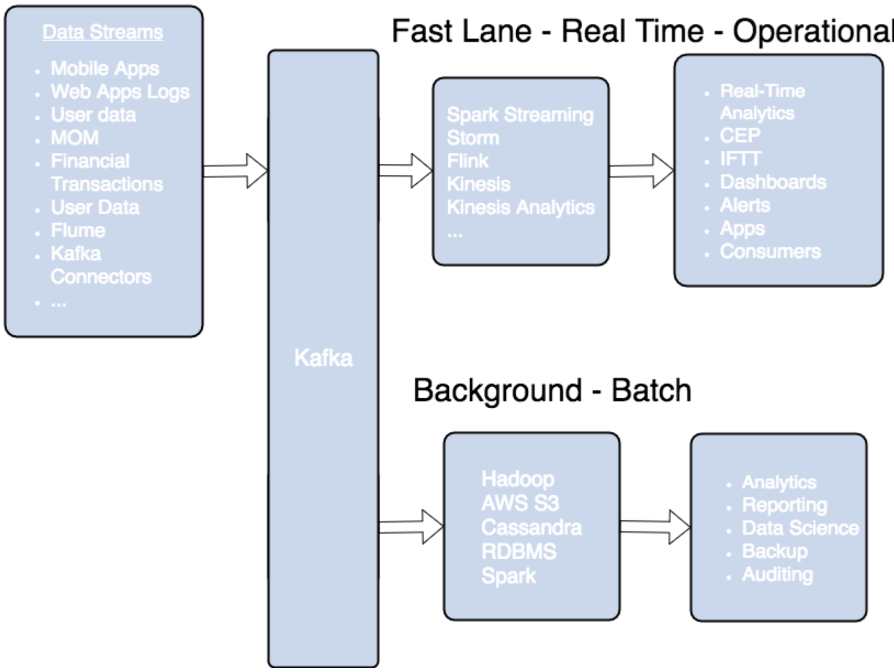


Fig:2 Kafka streaming architecture diagram

2.2. Spark Streaming:

A data stream is an unbounded sequence of continuously arriving data. Streaming split the input data flowing continuously into discrete units for further processing. Stream processing is collection and analysis of streaming data at low latency.

In 2013, Spark Streaming was introduced to Apache Spark, a core Spark API extension which provides scalable, high-throughput, and fault-tolerant stream processing of live data streams. Data ingestion here is done by Kafka but can be done by varieties of other sources such as Apache Flume, Amazon Kinesis, etc and processing can be done using complex algorithms expressed with high-level map, reduction, join and window functions. Finally, it is possible to transfer processed data into file systems and databases.

Batch processing can give great insights into things that happened in the past, but it is not possible to answer the question of "what is happening right now?" using batch processing. It has become important nowadays to process events as they arrive for real-time insights, but high performance at scale is necessary to do this.

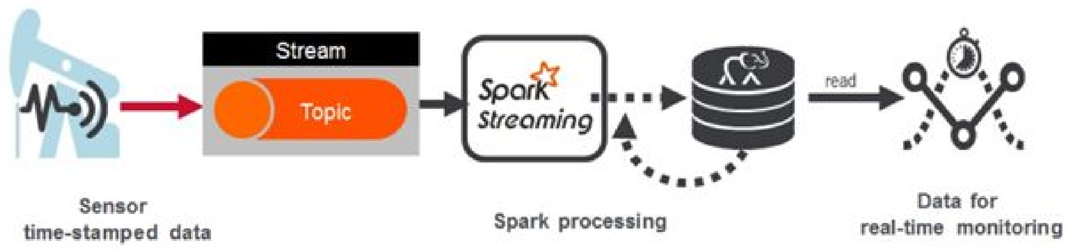


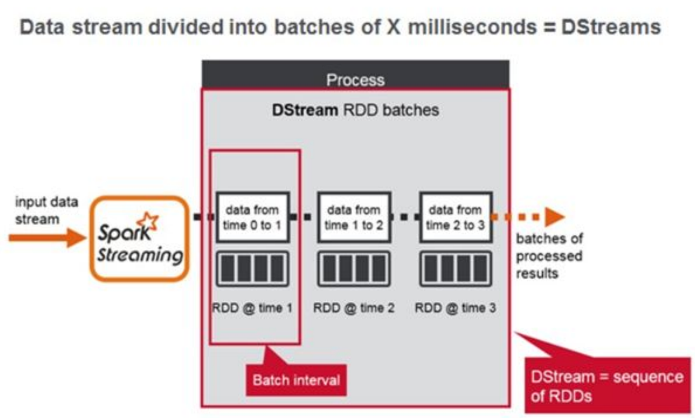
Fig:3 Spark Streaming

The real-time weather data that we have is the streaming data from the weather sensors. This data is processed by Spark and stored in HBase, for further analytics. We will store4 every single event in HBase as it streams in.

2.3. Real-time Data Processing using Spark Streaming

Spark APIs are used by Spark Streaming to stream processes and it lets us use the same APIs for streaming and batch processing. We can use Spark's core API to process data streams and we use HBase to store it.

Data stream is divided into batches called Dstreams, and each of these batches are of X seconds, which internally is a sequence of Resilient Distributed Datasets (RDD), one for each batch interval. The RDD is the primary abstraction in Spark. The records received during the batch interval are contained in the RDD.



The data stored in RDDs is partitioned, and concurrent operations are performed on the data cached in memory. Spark caches RDDs in memory, whereas MapReduce involves more reading and writing from disk. The key to high performance is partitioning and reducing the disk I/O. There are two types of operations that can be done on DStreams: Transformations and output operations.

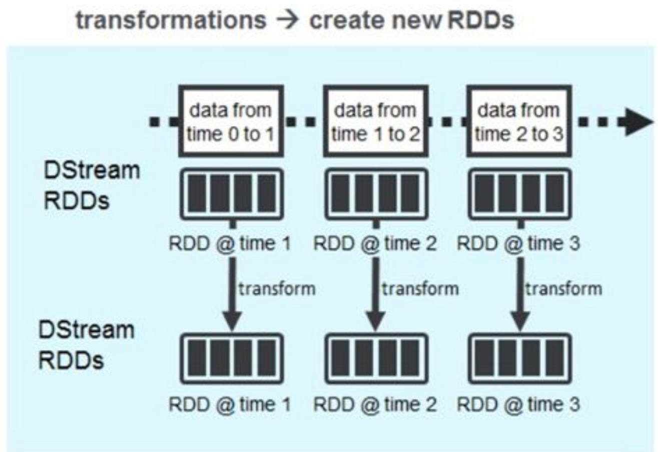


Fig:4 DStream Transformations operation

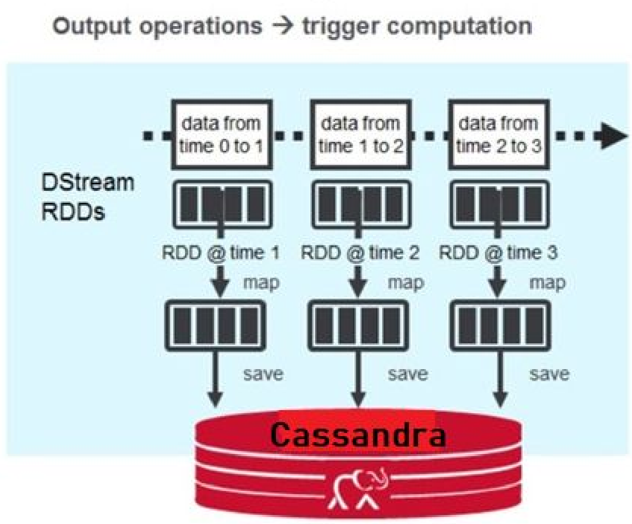


Fig:5 DStream Output operations

2.4. Data Storage in Cassandra

Originally developed by Facebook in 2007, Apache Cassandra is a free, open-source, distributed, wide column store, NoSQL database management system designed to handle large quantities of data across several commodity servers, providing high availability with no single point of failure. Cassandra uses a Dynamo architecture and a Bigtable-style data model to provide a high-accessibility, high-scalability NoSQL data store Cassandra provides robust support for multi-center clusters with asynchronous masterless replication that enables low latency operations for all clients.

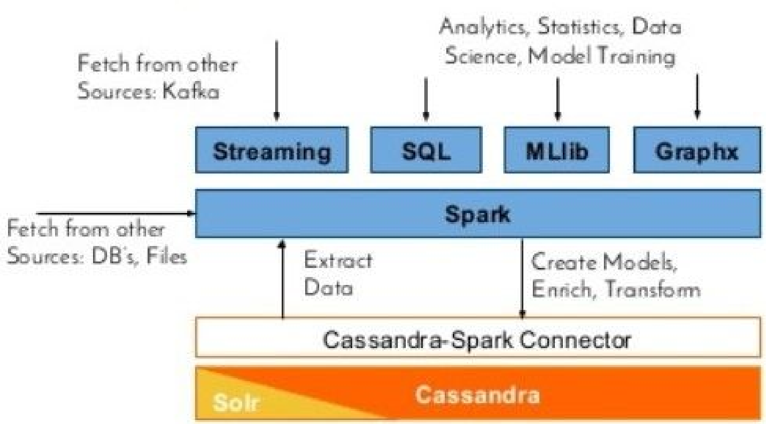
Why have we used Cassandra?

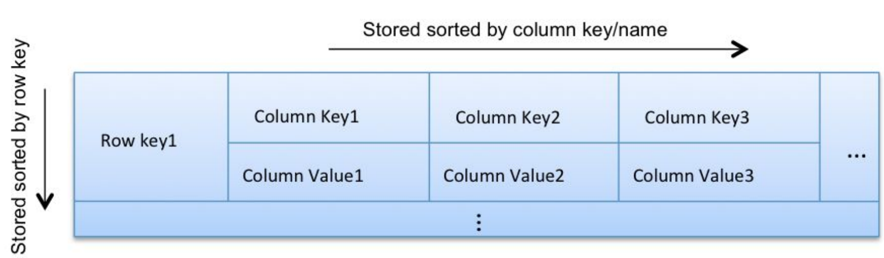
1. Highly Scalable NoSQL database:
    - Cassandra supplies linear scalability
    - Cassandra is a partitioned row store database
    - Automatic data distribution
    - Built-in and customizable replication
2. High Availability:
    - In a Cassandra cluster all nodes are equal
    - There are no masters or coordinators at the cluster level
3. Read/Write anywhere:

    Cassandra is a R/W anywhere architecture, so any user can connect to any node in any DC and read/write the data.
4. High Performance:
    - All disk writes are sequential, append-only operations
    - Ensure no reading before write

Spark Cassandra connector

- It allows us to expose Cassandra tables as Spark RDDs
- Write Spark RDDs to Cassandra tables
- It maps table rows to CassandraRow objects or tuples
- Do join with a subset of Cassandra data
- Partition RDDs according to Cassandra replication





Once the weather API data is moved from Kafka into Spark streaming, RDD transformation is then performed and the resulting values are then stored in Cassandra.

3. Problem Statement and Proposed Solution

To build a streaming data pipeline. To create a high-throughput, scalable, reliable and fault-tolerant data pipeline capable of fetching event-based data and streaming those events to Apache Spark which will parse their content, all of which will be done in near real-time. The data will be stored in Cassandra.

4. Implementation of the Solution Proposed

4.1 Intuit's application architecture

Before detailing Intuit's implementation, it is helpful to consider the application architecture and physical architecture in the AWS Cloud. The following application architecture can launch via a public subnet or within a private subnet.

4.2 Steps for project execution:

- Setting up the virtual machine
- Download the required Packages
- Installing Dependencies
- Creating Open Weather API key and Json Sample
- Running Kafka Producer-Consumer

· Running Spark

· Running Cassandra and Creating Table

· Steps for Execution and Results

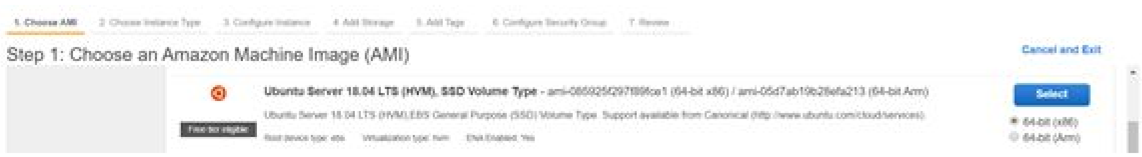
4.2.1 Setting Up a EC2 Machine on AWS and Create a EMR-KEY PAIR

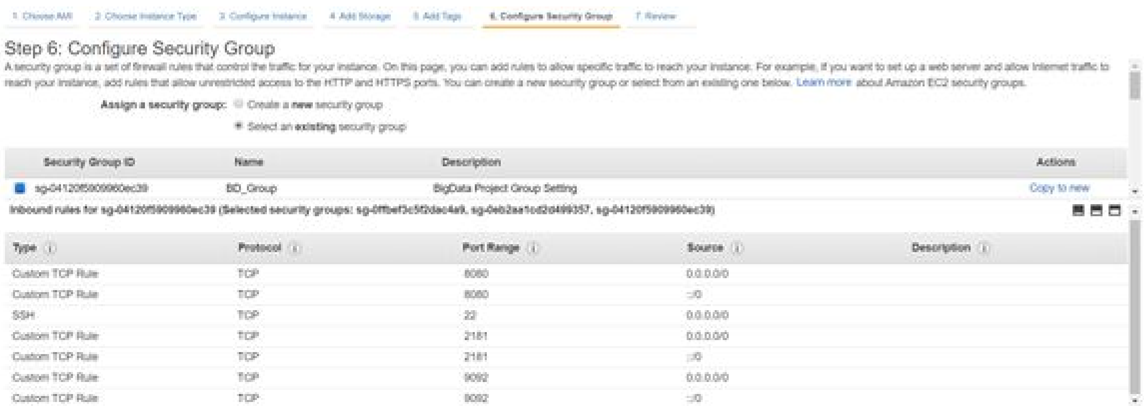
Amazon Elastic Compute Cloud (Amazon EC2) provides secure and resizable computing capacity in the AWS cloud. Using Amazon EC2 eliminates the need to invest in hardware up front, so you can develop and deploy applications faster.

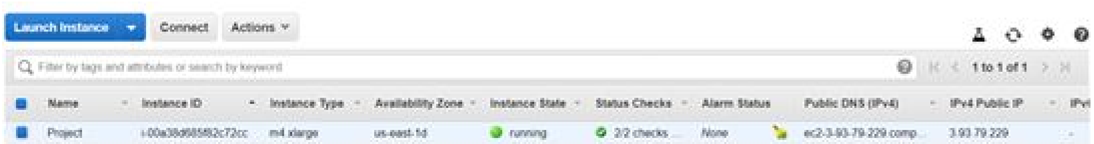
Given that Intuit had existing infrastructure leveraging Kafka on AWS, the first version was designed using Apache Kafka on Amazon EC2, EMR for Persistence.

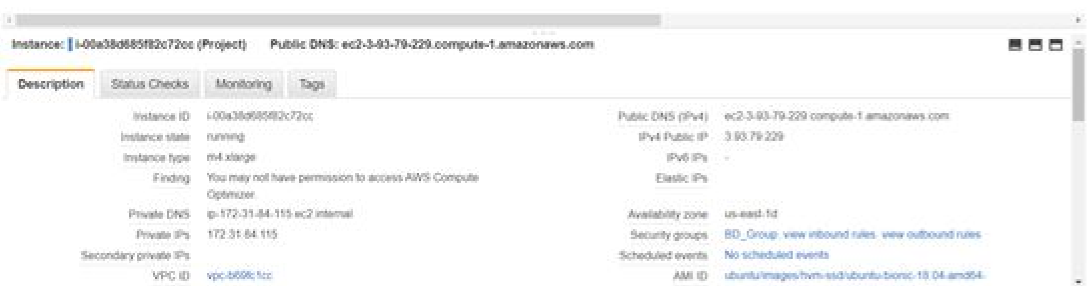
Steps for creating EC2 instance and EMR- Key pair

1. Login and access to **AWS** services.
2. Choose AMI.
3. Choose **EC2** Instance Types.
4. Configure Instance.
5. Add Storage.
6. Tag Instance.
7. Configure Security Groups.
8. Review Instances.



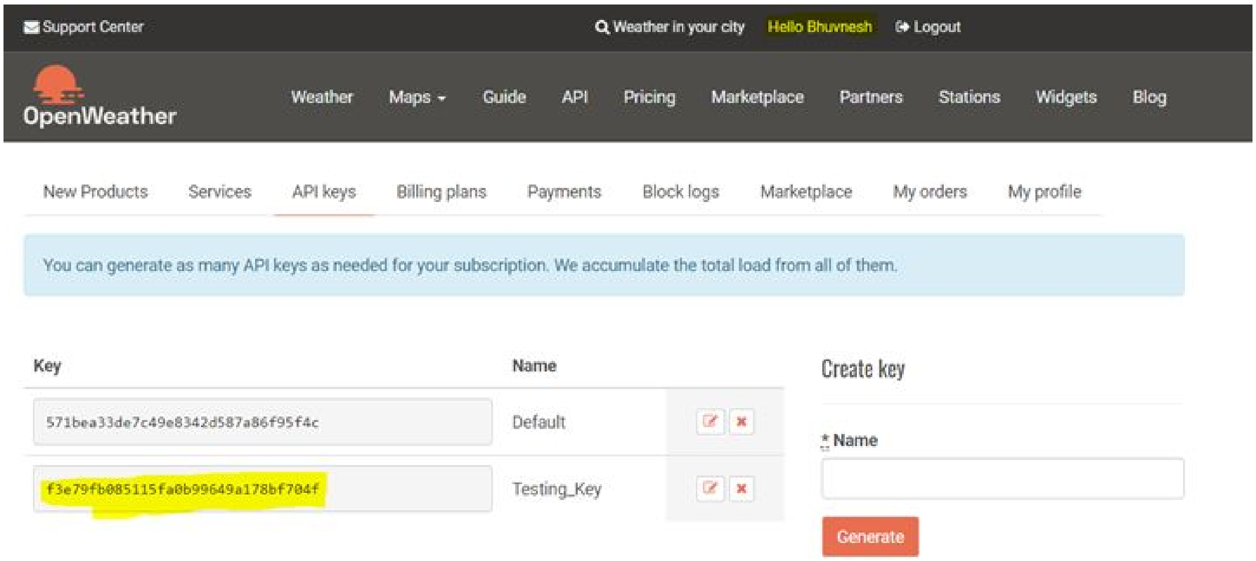






4.2.2. Creating Open Weather API key and Json Sample

1. Visit https://openweathermap.org/ and Sign Up. Use their fast and easy-to-work weather APIs for free.
2. Create an API Key which can be used in Kafka Producer to fetch data.



Sample Json

{"coord":{"lon":-0.13,"lat":51.51},"weather":[{"id":300,"main":"Drizzle","description":"light intensity drizzle","icon":"09d"}],"base":"stations","main":
{"temp":280.32,"pressure":1012,"humidity":81,"temp_min":279.15,"temp_max":281.15},"visibility":10000,"wind":{"speed":4.1,"deg":80},"clouds":{"all":90},"dt":1485789600,"sys":
{"type":1,"id":5091,"message":0.0103,"country":"GB","sunrise":1485762037,"sunset":1485794875},"id":2643743,"name":"London","cod":200}

4.2.3. Download Packages

Once EC2 machine is setup, now it's time to download required packages for the project:

1. KAFKA

   Go to https://kafka.apache.org/downloads and download Kafka_2.12-2.4.0.tgz on the local machine

2. SPARK

   Go to https://spark.apache.org/downloads.htm and download Spark-2.1.0-bin-hadoop2.7.tgz on the local machine

3. Get Cassandra 3.11.6

   Steps mentioned in next part (Installing Dependencies)

4.2.4 Install Dependencies and Setting up Kafka, Spark and Cassandra

First things first! We need to connect to our machine using SSH. We will start with Updating and Installing the required dependencies.

4.2.4.1. Execution of the following general commands required for all three:

1. *sudo apt update*
2. *sudo apt install openjdk-8-jdk openjdk-8-jre*
3. SCP Kafka_2.12-2.4.0.tgz and Spark-2.1.0-bin-hadoop2.7.tgz from your local machine to EC2 ubuntu machine.

   Execute command on local-

   Command: *scp -i emr-key-pair.pem Kafka_2.12-2.4.0.tgz Spark-2.1.0-bin-hadoop2.7.tgz ubuntu@<IPV4 address>:/home/ubuntu*

4.2.4.2. Following Commands will get Kafka installed:

1. Command: *sudo apt install python3-pip*
2. Command: *pip3 install kafka-python*
3. Make a directory ~/SERVER/KAFKA .This will be the directory to install Kafka
4. Move Kafka_2.12-2.4.0.tgz to SERVER/KAFKA and execute Command: *tar xvzf kafka_2.12-2.4.0.tgz*
5. Move to kafka_2.12-2.4.0 directory and create two python files: Kafka_Producer.py and Kafka_Consumer.py
6. The files will contain the python code for Producer and Consumer respectively.
7. Kafka is installed on our Machine.

4.2.4.3. Following Commands will get Spark installed:

1. Command: *export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/*
2. Make a directory ~/SERVER/SPARK . This will be your directory to install Spark
3. Move Spark-2.1.0-bin-hadoop2.7.tgz to SERVER/SPARK and execute command: *tar xvzf Spark-2.1.0-bin-hadoop2.7.tgz*
4. Move to Spark-2.1.0-bin-hadoop2.7.tgz directory and create a python files: KafkaSparkStreaming.py
5. The files will contain the python code for Real-time message consumption through Kafka-Spark Streaming and Storing into Cassandra.
6. Spark is installed on our Machine.

4.2.4.4. Following Commands will get Cassandra installed:

1. Install the apt-transport-https package that is necessary to access a repository over HTTPS: *sudo apt install apt-transport-https*
2. Add the Apache repository of Cassandra to

   /etc/apt/sources.list.d/cassandra.sources.list *wget -q -O - https://www.apache.org/dist/cassandra/KEYS | sudo apt-key add -*

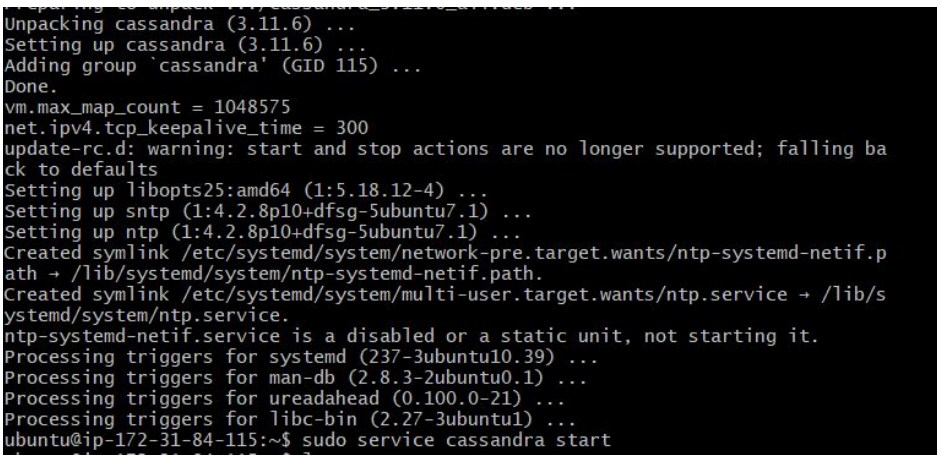
sudo sh -c 'echo "deb http://www.apache.org/dist/cassandra/debian 311x main" > /etc/apt/sources.list.d/cassandra.list'

3. Update the repositories

   sudo apt update

4. Install Cassandra

   sudo apt install cassandra



```
Unpacking cassandra (3.11.6) ...
Setting up cassandra (3.11.6) ...
Adding group `cassandra' (GID 115) ...
Done.
vm.max_map_count = 1048575
net.ipv4.tcp_keepalive_time = 300
update-rc.d: warning: start and stop actions are no longer supported; falling ba
ck to defaults
Setting up libopts25:amd64 (1:5.18.12-4) ...
Setting up sntp (1:4.2.8p10+dfsg-5ubuntu7.1) ...
Setting up ntp (1:4.2.8p10+dfsg-5ubuntu7.1) ...
Created symlink /etc/systemd/system/network-pre.target.wants/ntp-systemd-netif.p
ath → /lib/systemd/system/ntp-systemd-netif.path.
Created symlink /etc/systemd/system/multi-user.target.wants/ntp.service → /lib/s
ystemd/system/ntp.service.
ntp-systemd-netif.service is a disabled or a static unit, not starting it.
Processing triggers for systemd (237-3ubuntu10.39) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for ureadahead (0.100.0-21) ...
Processing triggers for libc-bin (2.27-3ubuntu1) ...
ubuntu@ip-172-31-84-115:~$ sudo service cassandra start
```

4.2.4. Running Kafka

1. We have already created a virtual machine and Installed **Kafka_2.12-2.4.0** (while installing all the dependencies)

2. Kafka uses Zookeeper so you need to first start a local ZooKeeper server in the window. You use the following command packaged with kafka to get a quick-and-dirty single-node Zookeeper instance. Notice the “&” at the end of the command. It instructs the command shell to run this command in the background

   Command: *bin/zookeeper-server-start.sh config/zookeeper.properties > ./zookeeper-logs &*

3. Now start the Kafka server itself in the window as follows:

   Command: *bin/kafka-server-start.sh config/server.properties > ./kafka-logs &*

4. Let's create a topic named "Weather" with a single partition and only one replica.

Command: *./bin/kafka-topics.sh --zookeeper ec2-3-93-79-229.compute-1.amazonaws.com:2181 --create --topic Weather --partitions 1 --replication-factor 1*

5. We can now see that topic if we run the list topic command.

   Command: *./bin/kafka-topics.sh --zookeeper ec2-3-93-79-229.compute-1.amazonaws.com:2181 –list*

6. Run the Producer
Command: *python3 Kafka_Producer.py*

7. Run the Consumer to see the result of real-time data coming from OpenWeatherApi. We can see the results for Chicago City.
Command: *python3 Kafka_Consumer.py*

4.2.4.1. Result:

We can see that Kafka is up and running. The messages are being produced by Kafka Producer and are also being consumed by Kafka Consumer.

ubuntu@ip-172-31-84-115: ~/SERVER/KAFKA

```python
#!/usr/bin/env python
# coding: utf-8

# In[12]:


import webbrowser
import json
import requests
import time
from kafka import KafkaProducer
from kafka.errors import KafkaError
from urllib.parse import urlencode
import pandas as pd
from pandas.io.json import json_normalize


API_KEY = 'f3e79fb085115fa0b99649a178bf704f'
Locations = [4887398]#,4164138,5391959]
weatherDF = pd.DataFrame(columns=("Name","Country","WindSpeed"))


#Setting up Kafka
KAFKA_TOPIC = 'Weather'
KAFKA_BROKERS = 'localhost:9092'
producer = KafkaProducer(bootstrap_servers=KAFKA_BROKERS,
                         value_serializer=lambda v:json.dumps(v).encode('utf-8'),
                         linger_ms=10)

for i in Locations:
    mydict = {'id': i, 'appid': API_KEY}
    WEATHER_URL = 'http://api.openweathermap.org/data/2.5/weather?'
    url = WEATHER_URL + urlencode(mydict, doseq=True)
    #webbrowser.open(url)
    response1 = requests.get(url)
    info_as_json = json.loads(response1.text)
    #print(info_as_json)
    weatherDF = weatherDF.append({"Name":info_as_json['name'],\
            "Country":info_as_json['sys']['country'],\
            "WindSpeed":info_as_json['wind']['speed']}, ignore_index=True)
    producer.flush()
weather= json.loads(weatherDF.to_json(orient='split',index=False))
producer.send(KAFKA_TOPIC, weather)
producer.flush()
print(weatherDF)
```

ubuntu@ip-172-31-84-115: ~/SERVER/KAFKA

```python
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import sys
from kafka import KafkaConsumer

KAFKA_TOPIC = 'Weather'
KAFKA_BROKERS = 'localhost:9092'

consumer = KafkaConsumer(bootstrap_servers=KAFKA_BROKERS,auto_offset_reset='latest')

consumer.subscribe([KAFKA_TOPIC])
try:
    for message in consumer:
        print(message.value)
except KeyboardInterrupt:
    sys.exit()
```

4.2.5. Running Spark

1. We have already created a virtual machine and Installed Spark-2.1.0-bin-hadoop2.7 (while installing all the dependencies)

2. Move to SPARK directory and start the Master process.

   Command: *./sbin/start-master.sh*

   Now you should be able to access the Spark Master status page on the URL *http://ec2-XX-XXX-XXX-XX.sa-east-1.compute.amazonaws.com:8080/*, remember to enable access to port 8080 on your EC2 machine.

3. Let's start the Slave service

   Command: *./sbin/start-slave.sh spark://ip-172-31-46-153.ec2.internal:7077*

4. Create a KafkaSparkStream.py file in your Spark directory which will connect the Kafka Server, Spark Master Server and Start receiving messages produced from Kafka Producer. It will also load the Cassandra Table.

5. Run the command with following packages:

   Command: *bin/spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.0.2,com.datastax.spark:spark-cassandra-connector_2.11:2.5.0 ../KafkaSparkStreaming.py*

4.2.5.1 Result:

We can see that Spark is up and running. The messages that are sent by Kafka Producer and are getting displayed and we are converting it to a dataframe.

MINGW64:/c/Users/bhuvn/OneDrive/Desktop/BigDataProject

```python
from pyspark.sql import SparkSession,SQLContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
import json
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *

# Task configuration.
topic = "weather"
brokerAddresses = "localhost:9092"
batchTime = 20

# Creating stream.
spark = SparkSession.builder.appName("Kakfa-SparkStreaming").getOrCreate()
sc = spark.sparkContext
sql = SQLContext(sc)
stream = StreamingContext(sc, batchTime)
kafka_stream = KafkaUtils.createDirectStream(stream, [topic], {"metadata.broker.list": brokerAddresses})

# Creating Schema
schema = StructType([StructField("columns", StringType(), True),StructField("data", StringType(), True)])

def main():

    lines = kafka_stream.map(lambda x: x[1])
    lines.pprint()
    lines.foreachRDD(process_batch)

    # Starting the task run.
    stream.start()
    stream.awaitTermination()

def process_batch(rdd):
    if not rdd.isEmpty():
        lines = rdd.map(str)
        df2 = sql.read.json(lines, schema)
        split_col = split(df2['data'], ',')
        df2 = df2.withColumn('city', split_col.getItem(0))
        df2 = df2.withColumn('country', split_col.getItem(1))
        df2 = df2.withColumn('windSpeed', split_col.getItem(2))
        df2 = df2.withColumn('city', regexp_replace(regexp_replace('city', '"', ''),'\\[\\[','')).withColumn('country', regexp_replace('country', '"',""))\
                .withColumn('windspeed', regexp_replace('windspeed', '\\]\\]',""))
        df2 = df2.drop('columns')
        df2 = df2.drop('data')
        df2.show()

        #Writing to Cassandra Table and Creating a csv file as backup
        df2.write.format("org.apache.spark.sql.cassandra").mode('append').options(table="weather",keyspace="a20444878").save()
        df2.write.format('csv').save('file:///home/ubuntu/output3.csv')


if __name__ == '__main__':
    main()
```

```
ubuntu@ip-172-31-84-115:~/SERVER/SPARK/spark-2.1.0-bin-hadoop2.7$ bin/spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.0.2,com.datastax.spark:spark-cassandra-connector_2.11:2.5.0 ../KafkaSparkStreaming.py
Ivy Default Cache set to: /home/ubuntu/.ivy2/cache
The jars for the packages stored in: /home/ubuntu/.ivy2/jars
:: loading settings :: url = jar:file:/home/ubuntu/SERVER/SPARK/spark-2.1.0-bin-hadoop2.7/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
org.apache.spark#spark-streaming-kafka-0-8_2.11 added as a dependency
com.datastax.spark#spark-cassandra-connector_2.11 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
        confs: [default]
```

```
20/05/06 19:07:01 INFO VerifiableProperties: Property zookeeper.connect is overridden to
20/05/06 19:07:01 INFO PythonRunner: Times: total = 224, boot = 170, init = 53, finish = 1
20/05/06 19:07:01 INFO PythonRunner: Times: total = 8, boot = 2, init = 6, finish = 0
20/05/06 19:07:01 INFO PythonRunner: Times: total = 47, boot = -2, init = 49, finish = 0
20/05/06 19:07:01 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1728 bytes result sent to driver
20/05/06 19:07:01 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 1310 ms on localhost (executor driver) (1/1)
20/05/06 19:07:01 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/05/06 19:07:01 INFO DAGScheduler: ResultStage 0 (runJob at PythonRDD.scala:441) finished in 1.330 s
20/05/06 19:07:01 INFO DAGScheduler: Job 0 finished: runJob at PythonRDD.scala:441, took 1.493895 s
-------------------------------------------
Time: 2020-05-06 19:07:00
-------------------------------------------
{"columns": ["Name", "Country", "WindSpeed"], "data": [["Chicago", "US", 3.1]]}

20/05/06 19:07:01 INFO JobScheduler: Finished job streaming job 1588792020000 ms.0 from job set of time 1588792020000 ms
20/05/06 19:07:01 INFO JobScheduler: Starting job streaming job 1588792020000 ms.1 from job set of time 1588792020000 ms
20/05/06 19:07:01 INFO SparkContext: Starting job: runJob at PythonRDD.scala:441
20/05/06 19:07:01 INFO DAGScheduler: Got job 1 (runJob at PythonRDD.scala:441) with 1 output partitions
20/05/06 19:07:01 INFO DAGScheduler: Final stage: ResultStage 1 (runJob at PythonRDD.scala:441)
```

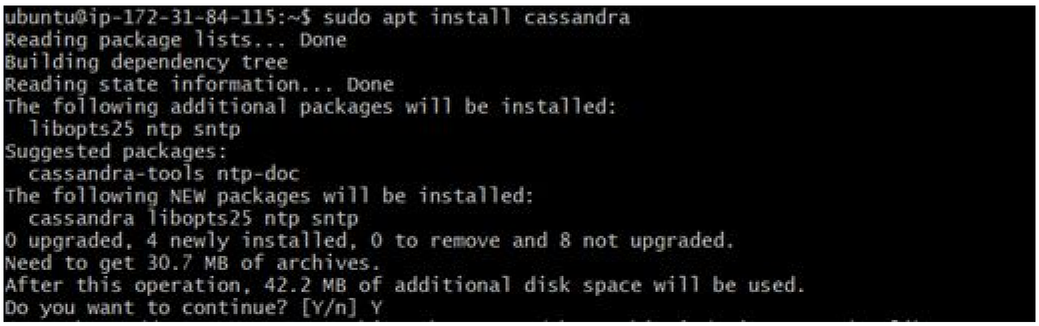
```
20/05/06 19:07:04 INFO PythonRunner: Times: total = 41, boot = -2, init = 43, finish = 0
20/05/06 19:07:04 INFO Executor: Finished task 0.0 in stage 2.0 (TID 2). 2032 bytes result sent to driver
20/05/06 19:07:04 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 2) in 154 ms on localhost (executor driver) (1/1)
20/05/06 19:07:04 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
20/05/06 19:07:04 INFO DAGScheduler: ResultStage 2 (showString at NativeMethodAccessorImpl.java:0) finished in 0.155 s
20/05/06 19:07:04 INFO DAGScheduler: Job 2 finished: showString at NativeMethodAccessorImpl.java:0, took 0.178798 s
20/05/06 19:07:04 INFO CodeGenerator: Code generated in 13.622237 ms
+-------+-------+---------+
|   city|country|windspeed|
+-------+-------+---------+
|Chicago|     US|      3.1|
+-------+-------+---------+

20/05/06 19:07:04 INFO BlockManagerInfo: Removed broadcast_2_piece0 on 172.31.84.115:44515 in memory (size: 12.3 KB, free: 366.3 MB)
20/05/06 19:07:04 INFO ContextCleaner: Cleaned accumulator 97
20/05/06 19:07:04 INFO ContextCleaner: Cleaned accumulator 98
20/05/06 19:07:04 INFO DefaultMavenCoordinates: DataStax Java driver for Apache Cassandra(R) (com.datastax.oss:java-driver-core-shaded) version 4.5.0
```

4.2.6. Running Cassandra and Creating Table

1. The installation proceeds in two phases: create a virtual machine and then install and start Cassandra.
2. We have already created a virtual machine and Installed Cassandra (while installing all the dependencies).

Command: *sudo apt install cassandra*



3. We will now proceed and start Cassandra

Command: *sudo service cassandra start*

4. Create a Keyspace by entering the following command and saving it in a file called init.cql

Command:

CREATE TABLE a20444878.weather (
City text,
Country text,
WindSpeed text,
PRIMARY KEY(City)
) ;

```
cqlsh> USE a20444878;
cqlsh:a20444878> source './ex2.cql'
cqlsh:a20444878> DESCRIBE TABLE weather

CREATE TABLE a20444878.weather (
    city text PRIMARY KEY,
    country text,
    windspeed int
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCom
pactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandr
a.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

5. Now create a file in your working directory called ex2.cql. In this file write the command to create a table named ‘weather’

Command: *SELECT * FROM weather;*

```
cqlsh:a20444878> SELECT * FROM weather;

 city | country | windspeed
------+---------+-----------

(0 rows)
```

We can see that an empty table is generated and waiting for data to get loaded.

5. Steps for Execution and Results

1. Once we have created a AWS EC2 instance of Ubuntu 18.04 and also created openweather api Key, we can then download and install all the required packages and dependencies.
2. Open three terminal windows such as EC2-1 (for Kafka), EC2-2 (for Spark) and EC2-3 (for Cassandra). SSH all three windows to connect to our Machine

Command: *ssh -i emr-key-pair.pem ubuntu@ec2-3-93-79-229.compute-1.amazonaws.com*

3. In your EC2-2 window, move to ~/SERVER/SPARK/spark-2.1.0-bin-hadoop2.7 directory and considering you have KafkaSparkStreaming.py in ~/SERVER/SPARK/ directory, execute the following command to make Spark Streaming Running.

Command: *bin/spark-submit --packages org.apache.spark:spark-streaming-kafka-0-8_2.11:2.0.2,com.datastax.spark:spark-cassandra-connector_2.11:2.5.0 ../KafkaSparkStreaming.py*

```
20/05/06 05:20:20 INFO KafkaRDD: Beginning offset 160 is the same as ending offset skipping Weather 0
20/05/06 05:20:20 INFO PythonRunner: Times: total = 41, boot = -19735, init = 19776, finish = 0
20/05/06 05:20:20 INFO PythonRunner: Times: total = 43, boot = -19733, init = 19776, finish = 0
20/05/06 05:20:20 INFO PythonRunner: Times: total = 43, boot = -41, init = 84, finish = 0
20/05/06 05:20:20 INFO Executor: Finished task 0.0 in stage 9.0 (TID 9). 1539 bytes result sent to driver
20/05/06 05:20:20 INFO TaskSetManager: Finished task 0.0 in stage 9.0 (TID 9) in 137 ms on localhost (execut
20/05/06 05:20:20 INFO TaskSchedulerImpl: Removed TaskSet 9.0, whose tasks have all completed, from pool
20/05/06 05:20:20 INFO DAGScheduler: ResultStage 9 (runJob at PythonRDD.scala:441) finished in 0.138 s
20/05/06 05:20:20 INFO DAGScheduler: Job 9 finished: runJob at PythonRDD.scala:441, took 0.143652 s
-------------------------------------------
Time: 2020-05-06 05:20:20
-------------------------------------------

20/05/06 05:20:20 INFO JobScheduler: Finished job streaming job 1588742420000 ms.0 from job set of time 1588
20/05/06 05:20:20 INFO JobScheduler: Starting job streaming job 1588742420000 ms.1 from job set of time 1588
20/05/06 05:20:20 INFO SparkContext: Starting job: runJob at PythonRDD.scala:441
20/05/06 05:20:20 INFO DAGScheduler: Got job 10 (runJob at PythonRDD.scala:441) with 1 output partitions
20/05/06 05:20:20 INFO DAGScheduler: Final stage: ResultStage 10 (runJob at PythonRDD.scala:441)
20/05/06 05:20:20 INFO DAGScheduler: Parents of final stage: List()
20/05/06 05:20:20 INFO DAGScheduler: Missing parents: List()
20/05/06 05:20:20 INFO DAGScheduler: Submitting ResultStage 10 (PythonRDD[33] at RDD at PythonRDD.scala:48),
```

Since Producer is not active hence, empty Spark Streaming.

4. Now, In your EC2-1 window, move to ~/SERVER/KAFKA/kafka_2.12-2.4.0 directory and considering you have Kafka_Producer.py and Kafka_Consumer.py in ~/SERVER/KAFKA/ directory, execute the following command to make Kafka Producer running

```
ubuntu@ip-172-31-84-115:~/SERVER/KAFKA$ python3 Kafka_Producer.py
      Name Country  WindSpeed
0  Chicago      US        3.1
ubuntu@ip-172-31-84-115:~/SERVER/KAFKA$ python3 Kafka_Consumer.py
b'{"columns": ["Name", "Country", "WindSpeed"], "data": [["Chicago", "US", 3.1]]}'
```

We can see Kafka Producer is active and sending messages which is also getting consumed by Kafka Producer.

```
20/05/06 19:07:01 INFO PythonRunner: Times: total = 224, boot = 170, init = 53, finish = 1
20/05/06 19:07:01 INFO PythonRunner: Times: total = 8, boot = 2, init = 6, finish = 0
20/05/06 19:07:01 INFO PythonRunner: Times: total = 47, boot = -2, init = 49, finish = 0
20/05/06 19:07:01 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1728 bytes result sent to driver
20/05/06 19:07:01 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 1310 ms on localhost (executor driver) (1/1)
20/05/06 19:07:01 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/05/06 19:07:01 INFO DAGScheduler: ResultStage 0 (runJob at PythonRDD.scala:441) finished in 1.330 s
20/05/06 19:07:01 INFO DAGScheduler: Job 0 finished: runJob at PythonRDD.scala:441, took 1.493895 s
-------------------------------------------
Time: 2020-05-06 19:07:00
-------------------------------------------
{"columns": ["Name", "Country", "WindSpeed"], "data": [["Chicago", "US", 3.1]]}

20/05/06 19:07:01 INFO JobScheduler: Finished job streaming job 1588792020000 ms.0 from job set of time 1588792020000 ms
20/05/06 19:07:01 INFO JobScheduler: Starting job streaming job 1588792020000 ms.1 from job set of time 1588792020000 ms
20/05/06 19:07:01 INFO SparkContext: Starting job: runJob at PythonRDD.scala:441
20/05/06 19:07:01 INFO DAGScheduler: Got job 1 (runJob at PythonRDD.scala:441) with 1 output partitions
20/05/06 19:07:01 INFO DAGScheduler: Final stage: ResultStage 1 (runJob at PythonRDD.scala:441)
20/05/06 19:07:01 INFO DAGScheduler: Parents of final stage: List()
20/05/06 19:07:01 INFO DAGScheduler: Missing parents: List()
20/05/06 19:07:01 INFO DAGScheduler: Submitting ResultStage 1 (PythonRDD[4] at RDD at PythonRDD.scala:48), which has no missing pare
```

We can parallelly see in EC2-2 Window that Spark is receiving the messages.

```
20/05/06 19:07:04 INFO PythonRunner: Times: total = 6, boot = 1, init = 5, finish = 0
20/05/06 19:07:04 INFO CodeGenerator: Code generated in 16.613029 ms
20/05/06 19:07:04 INFO PythonRunner: Times: total = 41, boot = -2, init = 43, finish = 0
20/05/06 19:07:04 INFO Executor: Finished task 0.0 in stage 2.0 (TID 2). 2032 bytes result sent to dr
20/05/06 19:07:04 INFO TaskSetManager: Finished task 0.0 in stage 2.0 (TID 2) in 154 ms on localhost
20/05/06 19:07:04 INFO TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from p
20/05/06 19:07:04 INFO DAGScheduler: ResultStage 2 (showString at NativeMethodAccessorImpl.java:0) fi
20/05/06 19:07:04 INFO DAGScheduler: Job 2 finished: showString at NativeMethodAccessorImpl.java:0, t
20/05/06 19:07:04 INFO CodeGenerator: Code generated in 13.622237 ms
+-------+-------+---------+
|   city|country|windspeed|
+-------+-------+---------+
|Chicago|     US|      3.1|
+-------+-------+---------+

20/05/06 19:07:04 INFO BlockManagerInfo: Removed broadcast_2_piece0 on 172.31.84.115:44515 in memory
20/05/06 19:07:04 INFO ContextCleaner: Cleaned accumulator 97
20/05/06 19:07:04 INFO ContextCleaner: Cleaned accumulator 98
20/05/06 19:07:04 INFO DefaultMavenCoordinates: DataStax Java driver for Apache Cassandra(R) (com.dat
20/05/06 19:07:05 INFO Clock: Could not access native clock (see debug logs for details), falling bac
20/05/06 19:07:05 INFO ChannelFactory: [s0] Failed to connect with protocol DSE_V2, retrying with DSE
20/05/06 19:07:05 INFO ChannelFactory: [s0] Failed to connect with protocol DSE_V1, retrying with V4
20/05/06 19:07:05 INFO Uuids: PID obtained through native call to getpid(): 0
20/05/06 19:07:06 INFO CassandraConnector: Connected to Cassandra cluster.
20/05/06 19:07:06 INFO SparkContext: Starting job: runJob at RDDFunctions.scala:36
20/05/06 19:07:06 INFO DAGScheduler: Got job 3 (runJob at RDDFunctions.scala:36) with 1 output partit
```

After transformation on rdd to make it a DataFrame, the output.

5. Now, in the EC2-3 window, enter cqlsh to start the Cassandra Query Language Shell. Enter the following command to see the loaded data.

   Command:

   USE a2044878;

   *SELECT * FROM weather;*

```
ubuntu@ip-172-31-84-115:~$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.6 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh> USE a20444878;
cqlsh:a20444878> SELECT * FROM weather;

 city    | country | windspeed
---------+---------+-----------
 Chicago |      US |       3.1

(1 rows)
cqlsh:a20444878> client_loop: send disconnect: Connection reset by peer
```

We observe that the data has been successfully loaded into the database through the pipeline. After this, we can use CQL to analyse the data and derive inferences.

6. Conclusions

In this project we have successfully written a python script for taking the real time weather data and sending it to Kafka Producer. Then we further move the data to Spark Streaming and create RDDs for the streamed data to be converted into dataframes. The data is then stored in the distributed, wide column store NOSQL database Cassandra. Finally, connect the streaming data pipeline to an analytics engine that lets us analyze the weather information on a real-time basis.

7. Github Link

Provided all the codes used for the application implementation.

https://github.com/BhuvneshT10/KafkaSparkCassandra.git

8. References

1. https://openweathermap.org/api
2. https://towardsdatascience.com/create-your-first-etl-pipeline-in-apache-spark-and-python-ec3d12e2c169
3. https://medium.com/forsk-labs/real-time-weather-analysis-using-kafka-and-elk-pipeline-a849eb27017a
4. https://mapr.com/blog/real-time-streaming-data-pipelines-apache-apis-kafka-spark-streaming-and-hbase/
5. https://data-flair.training/blogs/apache-spark-streaming-tutorial/
6. https://en.wikipedia.org/wiki/Apache_Cassandra
7. https://www.slideshare.net/DataStax/realtime-data-pipeline-with-spark-streaming-and-cassandra-with-mesos-rahul-kumar-sigmoid-c-summit-2016
8. https://www.duo.uio.no/bitstream/handle/10852/57141/Ibenholt-Master.pdf?sequence=11&isAllowed=y
9. https://aws.amazon.com/blogs/big-data/real-time-stream-processing-using-apache-spark-streaming-and-apache-kafka-on-aws/

9. Contribution of individual to the Project

Bhuvnesh Tejwani - A20444878
(i) Project Proposal and Project Description
(iI) EC2 Cluster Setup, Installing dependencies and sources
(iii) Python Scripts Kafka_Producer.py & Kafka_Consumer.py for Kafka
(iv) Python Script KafkaSparkStreaming.py for SparkStreaming
(v) Install and create table for Cassandra

Gurunath Reddy - A20443036

(i) Project Proposal and Project Description
(ii) Literature Survey - Kafka
(iii) EMR Cluster Setup, installing dependencies and sources required
(iv) Python script for Kafka spark Streaming Kafka_stream.py

Siri Chandrashekar - A20435389
(i) Project Proposal
(ii) Literature Survey - Spark Streaming & Cassandra
(iii) Background Check
(iv) Documentation

Anwesha Kakoty - A20433149
(i) Project Proposal
(ii) Background Check
(iii) Literature Survey - Hbase and Hive
(iv) Interpretation of Result