**Introduction:**

In this approach I focused on finding the Price behavior for taxi fare for every hour. We can use Uber API or Lift API to get the price details through their API. Collection price data from every location in United States for every 5 minutes is setup as a CRON job which ends up collecting large amount of data every day. We should use a message broker to queue the data because it’s a real-time data and at the same time we need the order of the messages collected so Kafka can be used as a message broker as it solves this problem working in a distributed environment. Apache Spark is used to process the streaming data by filtering and aggregating values and selecting all the required features for future processing. Apache Spark’s basic component is made up of RDD (Resilient Distributed Dataset) and it is processed in memory and based on lazy execution in a distributed environment, it is capable of handling streaming data. HBase is preferred to be used is this scenario where the datastore tables store the aggregate views for hour of day, day of week of each locations price and its surge in price is calculated from previous data. The table is designed with multiple partitions to support fast retrieval in an efficient way. Using a Web UI, we can show the locations which are hiked in price for every location.

Hadoop as a big data processing technology has been around for 10 years and has proven to be the solution of choice for processing large data sets Hadoop is one of the best possible choice to achieve these requirements since it can store bulk data with greater operational efficiency and cost reduction. In addition to this it also ensures other beneficial features like reliability, scalability and flexibility. Hadoop can store enormous data in HDFS and it allows this data to be analyzed using its MapReduce programming model. MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multi-pass computations and algorithms. Each step in the data processing workflow has one Map phase and one Reduce phase and you'll need to convert any use case into MapReduce pattern to leverage this solution. Spark working on top of Hadoop allows processing of structured and unstructured data ranging from terabytes to petabytes.

The Job output data between each step has to be stored in the distributed file system before the next step can begin. Hence, this approach tends to be slow due to replication & disk storage. Also, Hadoop solutions typically include clusters that are hard to set up and manage. Since we wanted to do something complicated, we would be needing to stick to a series of MapReduce jobs and execute them in sequence. Each of those jobs was high-latency, and none could start until the previous job had finished completely.

Spark allows us to develop complex, multi-step data pipelines using directed acyclic graph pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data.

Spark runs on top of existing Hadoop Distributed File System infrastructure to provide enhanced and additional functionality. It provides support for deploying Spark applications in an existing Hadoop v1 cluster (SIMR) Spark-Inside-MapReduce or Hadoop v2 YARN cluster or Apache Mesos.

In current scenario or current big data world, it is important to analyze data in real time stream. There are many business models which needs real time analysis to make their organization grow forward. To achieve better results the processing must be done faster. To demonstrate this kind of processing on big data we have many tools which helps us t do this with low latency. In olden days companies were following batch processing technique to process their data. To make insights and decisions from the data. But now because of high competition in every field the organizations must do the processing in real time to survive in this competitive pricing industry. To approach this problem, I have chosen Taxi Application like Lyft and Uber pricing analyzing in real time.

**Problem Statement**

Mobile Application Taxi companies are making profit from their customer whenever people use their application. Using the situation where demand is more, companies use this opportunity to increase the price and put up an added surge price based on the demand. Every company’s application is built on top of REST API Services. I am trying to analyze the price behavior of Uber based on which the other taxi application companies can provide a cheaper service than the competitor to attract more customers with profit.

**Technologies Used**

**Kafka**

Kafka is a distributed message streaming system that can stream data in real-time. It works based on a distributed environment, hence it can be easily scalable. It is based on publish subscribe model. In my application I used Kafka for streaming message in real-time.

**Kafka Topic** is specified as stream to which the messages are published into Kafka Topic that can be seen as a collection of messages which can be stored as a bucket. Kafka topics can be created and deleted as needed whenever required. Since it works in a distributed environment It is possible to set the replication factor for the topics to make sure the data is available in all situations.

**Kafka Producer** is the one who publishes the messages into Kafka topic. Producer first sends messages to Kafka Broker where the leader is located which is then published into the topic.

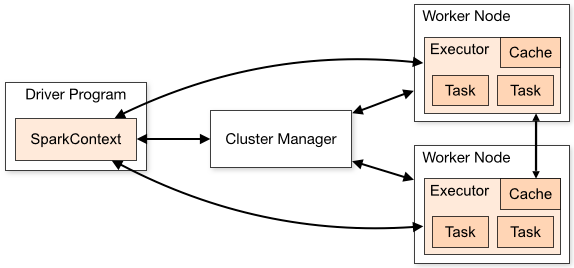
**Kafka Consumer** is the one who subscribes to the Topic for the messages in the topic. Consumer twitches the messages from the topic only when it is published by the producer and ready to consume the messages by producer. It also can pull messages corresponding to its current capacity.

**Kafka Broker** is stateless it does not have to maintain the number of messages consumed by the Kafka consumer basically this works in a distributed environment so every node in our cluster has the message. It is done by the Kafka consumer itself by maintaining offsets. Hence Kafka handles this by using time based retention policy where the broker keeps the message for certain amount of time after which it is not accessible

**Zookeeper** is used to maintain the coordination between the Kafka brokers. Zookeeper is used to notify the producers and the consumers about the availability or failure of the Kafka brokers.

**Spark** takes MapReduce to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, the performance can be several times faster than other big data technologies. Spark also supports lazy evaluation of big data queries, which helps with optimization of the steps in data processing workflows. It provides a higher-level API to improve developer productivity and a consistent architect model for big data solutions.

Spark holds intermediate results in memory rather than writing them to disk which is very useful especially when you need to work on the same dataset multiple times. It’s designed to be an execution engine that works both in-memory and on-disk. Spark operators perform external operations when data does not fit in memory. Spark can be used for processing datasets that larger than the aggregate memory in a cluster. Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours. Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines.

Spark contains multiple closely integrated components, at its core, Spark is a computational engine that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks on a computing cluster  


To run these operations, driver programs typically manage a number of nodes called executors.

Driver programs access Spark through a SparkContext, and SparkContext uses Py4J to launch a JVM and create a Java Spark Context. Py4J is only used on the driver for local communication between the Python and Java SparkContext objects. As mentioned earlier, Spark Core contains the basic functionality of Spark such components as task scheduling, memory management, fault recovery, interacting with storage systems.

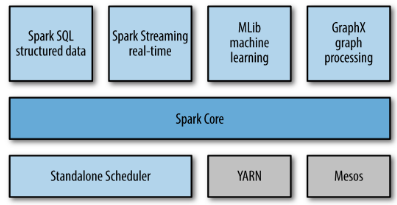
In other words, with Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. These collections / datasets are RDDs. RDDs are Spark's fundamental abstraction for distributed data and computation.

RDD transformations in Python are mapped to transformations on PythonRDD objects in Java. On remote worker machines, PythonRDD objects launch Python subprocesses and communicate with them using pipes, sending the user's code and the data to be processed.

**Spark Ecosystem**

Other than Spark Core API, there are additional libraries that are part of the Spark ecosystem and provide additional capabilities in Big Data analytics and Machine Learning areas.

1. **Spark Streaming -**  used for processing the real-time streaming data. This is based on micro batch style of computing and processing. It uses the DStream which is basically a series of RDDs, to process the real-time data.
2. **Spark SQL -** provides the capability to expose the Spark datasets over JDBC API and allow running the SQL like queries on Spark data using traditional BI.
3. **Spark MLlib -**  machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering.
4. **Spark GraphX -** GraphX is the new (alpha) Spark API for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph



Spark will attempt to store as much as data in memory and then will spill to disk. It can store part of a data set in memory and the remaining data on the disk. You have to look at your data and use cases to assess the memory requirements. With this in-memory data storage, Spark comes with performance advantage.

**INTEGRATION OF KAFKA & SPARK**

In this application data is streamed using Kafka and it is further handled using Spark. Both the data ingestion and data processing operations can be performed in real-time. Extracting the Pricing data from UBER API whenever it is collected by the producer program written in python. Once it is collected by the producer then immediately streaming it via Kafka makes the data available for real-time processing. It is one of the critical phase where the data has to be manipulated in the data processing phase, the resultant data streamed into Spark Streaming. Any defects in data will lead to improper productions.

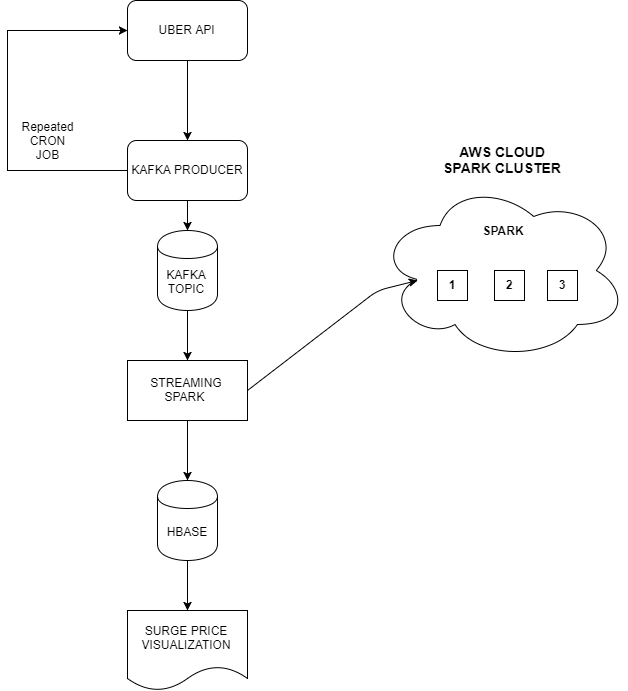


Fig 1 shows the flow chart for data ingestion, processing and then exporting and visualization.

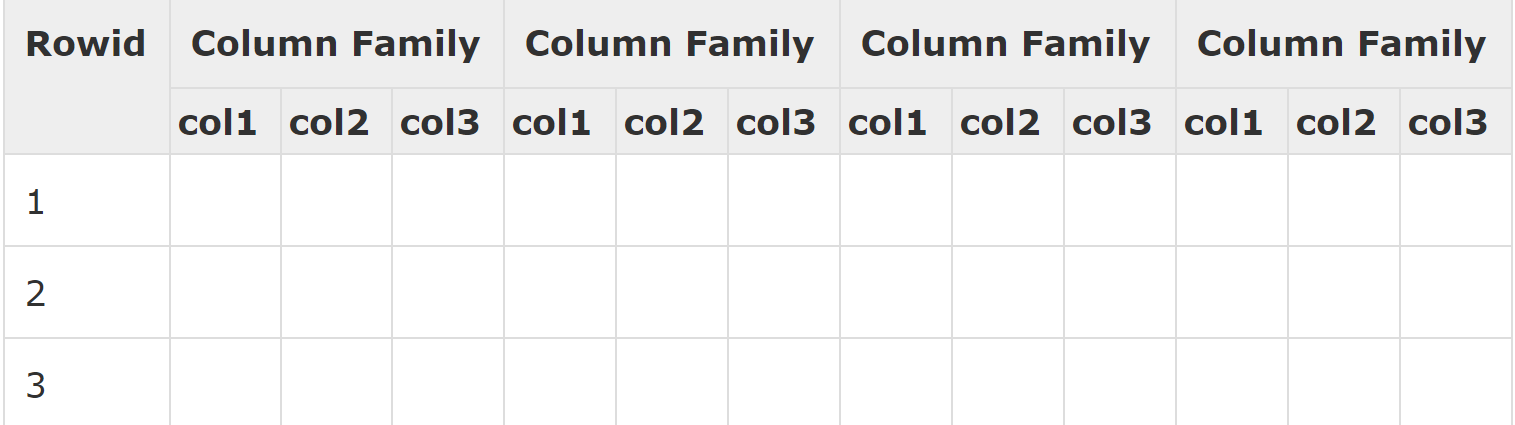
Here the pricing data is extracted from the data source and it is ingested via Kafka into Spark. In Kafka when the message is published to the Kafka topic zookeeper will be updated. This message will be consumed by the consumer based on its requests. The consumer sends the request along with an offset. This offset specifies the position of the message from where it wants to read. These offsets are maintained by the Zookeeper and every time there is communication between the Kafka producer, Kafka consumer and the Zookeeper.  
  
By this type of Kafka communication ensures that the message is ingested into Spark only once and hence it avoids redundancy. Since zookeeper keeps Kafka topic, producer and consumer coordinated with each other, Kafka consumer can be accurate about which messages are successfully ingested into Spark. This is done by keeping track of the message offsets. Hence it ensures that no message is delivered to Spark multiple times.

**HBASE**

The sudden increase in the volume of data from the order of gigabytes to zettabytes has created the need for a more organized file system for storage and processing of data. HDFS is fault-tolerant by design and supports rapid data transfer between nodes even during system failures. HBase is a non-relational and open source NO-SQL database that runs on top of Hadoop. HBase comes under CP type of CAP (Consistency, Availability, and Partition Tolerance) theorem. HBase is a Java based Not Only SQL database. HBase allows for dynamic changes and can be utilized for standalone applications. HBase is ideally suited for random write and read of data that is stored HDFS. HBase provides fast lookups for larger tables. HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups

The biggest drawbacks of Hadoop are its inability to perform real-time analysis, the trending requirement of the IT industry. HBase, on the other hand, can handle large data sets and is not appropriate for batch analytics. Instead, HBase use write/read data from Hadoop in real-time. HBase are capable of processing structured, semi-structured as well as un-structured data. HBase works as **an in-memory processing engine** that drastically increases the speed of read/write. HBase is a **column-oriented database** which is It is suitable for Online Analytical Processing (OLAP) and Column-oriented databases are designed for huge tables and the tables in it are sorted by row. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp.

* Table is a collection of rows.
* Row is a collection of column families.
* Column family is a collection of columns.
* Column is a collection of key value pairs.

  
  
Fig 2 shows the Columnar Schema of HBase.

Major features of HBase is linearly scalable and It has automatic failure support. It also provides consistent read and writes which integrates with Hadoop, both as a source and a destination. It has easy java API for clients and provides data replication across clusters.

**Related Work done**

I choose geolocation in USA using the latitude and longitude combination which acts as a data points in map. I have placed the few location points of a city with multiple area like downtown and location near Silicon Valley where it is always busy to check the price behavior in real time.

UBER API provides a RESTful service for requesting price estimate for the provided start and destination location. if we provide the start and end location we will get a JSON Response which has details for various category of cars available at that moment and the prize estimate for the distance and many other details. I have chosen a sample of 100 locations and permuting among them to produce different start and end location combination. Example say my data points are like (A, B, C, D, E) then the possible combinations of start and destination will (A, B), (A, C), (A, D), (A, E),(B, C).. and so on which would result me 20 combination of request.

I send this series of locations of start and end points of geolocation for estimating the price for the taxi. I get back the data in a form of JSON which usually all REST services do.

For my analysis I made this request and response to happen for every 2 minute which end up in adding tons of data every hour, every day and every month. To handle such kind of overwhelming data we need to decouple from data producers, to buffer unprocessed messages, etc. Kafka works well as a replacement for a more traditional message broker.

Our primary producer of data for will be from UBER. Separate topic must be created in kakfa and the consumers who is about to process the data and consume it can be added as a subscriber to topic.   
we can set up time for the data to live in the topic as 1 day in our Kafka cluster after which the data will be destroyed from the queue. By the time the next processing unit would have done the processing and stored the data into out storage model. Consumers here will be Spark streaming processor will label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be a separate machine which can be hosted in cloud high configuration. If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances. The reason to choose kafka here is that it keeps the order of the data constant within them and delivers to the consumer in the same order. As it is guaranteed in a high-level. Kafka has stronger ordering guarantees than a traditional messaging system.

* Messages given by a producer to a topic partition will be appended in the order they are given to Kafka topic. (i.e.) if a record M1 is sent by a producer and a record M2, then M1 is sent out first, then M1 will have a lower offset than M2 in the log.
* A consumer instance sees records in the order they are stored in the log.
* For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

**Setting Up Kafka**

1. Initially we need to start a zookeeper server - .***\bin\zkserver***
2. Next start the Kafka Server by using the command **- *.\bin\windows\kafka-server-start.bat .\config\server.properties***
3. Now create a Kafka topic - ***kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test***
4. To create a Kafka producer - ***kafka-console-producer.bat --broker-list localhost:9092 --topic test***
5. To create a Kafka Consumer - ***kafka-console-consumer.bat --zookeeper localhost:2181 --topic test***

**Glimpse on Data**

Sample Data Collected between the geolocations - 37.810604|-122.409856|37.775505|-122.446444 looks like this.

[{u'localized\_display\_name': u'SELECT', u'distance': 4.19, u'display\_name': u'SELECT', u'product\_id': u'57c0ff4e-1493-4ef9-a4df-6b961525cf92', u'high\_estimate': 29.0, u'low\_estimate': 23.0, u'duration': 1020, u'estimate': u'$23-29', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'uberXL', u'distance': 4.19, u'display\_name': u'uberXL', u'product\_id': u'821415d8-3bd5-4e27-9604-194e4359a449', u'high\_estimate': 20.0, u'low\_estimate': 16.0, u'duration': 1020, u'estimate': u'$16-20', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'BLACK', u'distance': 4.19, u'display\_name': u'BLACK', u'product\_id': u'd4abaae7-f4d6-4152-91cc-77523e8165a4', u'high\_estimate': 41.0, u'low\_estimate': 32.0, u'duration': 1020, u'estimate': u'$32-41', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'SUV', u'distance': 4.19, u'display\_name': u'SUV', u'product\_id': u'8920cb5e-51a4-4fa4-acdf-dd86c5e18ae0', u'high\_estimate': 53.0, u'low\_estimate': 42.0, u'duration': 1020, u'estimate': u'$42-53', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'ASSIST', u'distance': 4.19, u'display\_name': u'ASSIST', u'product\_id': u'ff5ed8fe-6585-4803-be13-3ca541235de3', u'high\_estimate': 17.0, u'low\_estimate': 13.0, u'duration': 1020, u'estimate': u'$13-17', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'WAV', u'distance': 4.19, u'display\_name': u'WAV', u'product\_id': u'2832a1f5-cfc0-48bb-ab76-7ea7a62060e7', u'high\_estimate': 17.0, u'low\_estimate': 13.0, u'duration': 1020, u'estimate': u'$13-17', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'POOL', u'distance': 4.19, u'display\_name': u'POOL', u'product\_id': u'26546650-e557-4a7b-86e7-6a3942445247', u'high\_estimate': 13.0, u'low\_estimate': 9.0, u'duration': 1020, u'estimate': u'$9-12', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'uberX', u'distance': 4.19, u'display\_name': u'uberX', u'product\_id': u'a1111c8c-c720-46c3-8534-2fcdd730040d', u'high\_estimate': 17.0, u'low\_estimate': 13.0, u'duration': 1020, u'estimate': u'$13-17', u'currency\_code': u'USD'}, {u'localized\_display\_name': u'TAXI', u'distance': 4.19, u'display\_name': u'TAXI', u'product\_id': u'3ab64887-4842-4c8e-9780-ccecd3a0391d', u'high\_estimate': None, u'low\_estimate': None, u'duration': 1020, u'estimate': u'Metered', u'currency\_code': None}] E-D|37.810604|-122.409856|37.775505|-122.446444

Spark supports text files, SequenceFiles, Avro, Parquet, and Hadoop InputFormat

Think of running this type of combination for all locations in US. How large data would be created. This scalable real world big data problem can be approached and solved using streaming Spark and HBase for making real time analysis.

**SPARK IMPLEMENTATION**

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Using Spark Streaming as a consumer to our topic**.** We use Spark Streaming’s high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. Internally, a DStream is represented as a sequence of [RDDs](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD)



**Resilient Distributed Datasets**

A Resilient Distributed Dataset (RDD), the basic element in Spark, which represents an immutable, collection of elements in partition that can be operated in parallel. We can think about RDD as a table in a database. It can hold any type of data. Spark stores data in RDD on different partitions. They help with rearranging the computations and optimizing the data processing. They are also fault tolerance because an RDD know how to recreate and recompted the datasets. In our case each batch of collected data which means each price estimate by a category or type of vehicle that is obscured by spark Context as per the configuration will be consumed as RDD. When we create the spark context we define out configuration for the SC where we can mention the batch time and app name.

By using the this feature we process the JSON data we get from Kafka. We create a data lake in Kafka and processes it in spark. We make all data wrangling to processes on the data.

You can modify an RDD with a transformation, and the transformation returns you a new RDD whereas the original RDD remains the same.

Spark allows us to create distributed datasets from any file stored including the Hadoop distributed filesystem (HDFS) or other storage systems supported by the Hadoop APIs such as local filesystem, Amazon S3, Cassandra, Hive, HBase, etc

RDD supports two types of operations:

* Transformation
* Action

**Transformation** don't return a single value, they return a new RDD. Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.

* Some functions are map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe, and coalesce.
* The in each JSON has upper and lower expected price we make a calculated field from those prices using a flatmap and store the RDD with mean price.
* We remove many unwanted keys from the JSON data using the filter transformation.

**Action** Operation evaluates and returns a new value. When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.

* Some of the Action operations are reduce, collect, count, first, take, countByKey, and foreach.

Using many available HBase connector library which are written in JAVA or SCALA can be used to put the data values into HBase which are provided as open source.

Spark is a distributed computing framework, there are two types of variables in spark. Broadcast and Accumulator variables which makes is possible for share a common variable among all nodes of the cluster. If a particular piece of data is accessed by all nodes of a cluster then we can use these Broadcast and Accumulator variables.

**Broadcast** is a read-only global variable, which all nodes of a cluster can read. Think of them more like as a lookup variable.

**Accumulators** as a global counter variable where each node of the cluster can write values in to. These are the variables that you want to keep updating as a part of your operation.

Spark framework manages the distribution, storage optimization and race or deadlock issues on these kinds of variables for achieving optimum performance. In my project I used the location dictionary as a broadcast variable at I will be often refereeing the values stored in it.

In HBase I have created two schemas where one is main schema which is used for storing the complete data for historical purpose and the second schema is processed table used for storing processed data which is used for visualization to identify price increase.

HBase is a Linear columnar storage. The schema is planned to be holding price and percentage of surge is calculated for every hour. So, the schema is visualized something like this for each column family.

**Column Family**: A-B eg: (A) downtown to (B) AMC theater

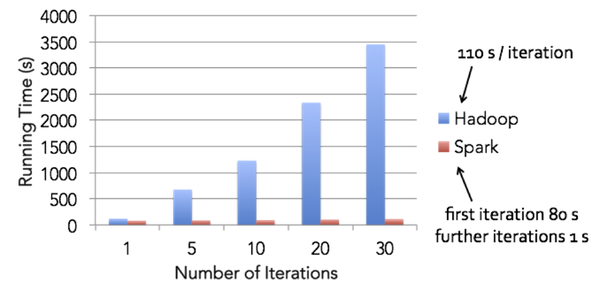
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Column-Family | A-B | | | | | | | B-C |
| TIME STAMP | C1 | C2 | C3 | …… | C22 | C23 | C24 | …. |
| 11-02-2015:8.50PM | 12.5 | 20.6 | 30 | …… | 0.1 | 56 | 35.23 | …. |

Here C denotes hour with each row denotes date and time. From Hbase using tableau or any other UI using google Map API to plot price surge locations which looks something like this.

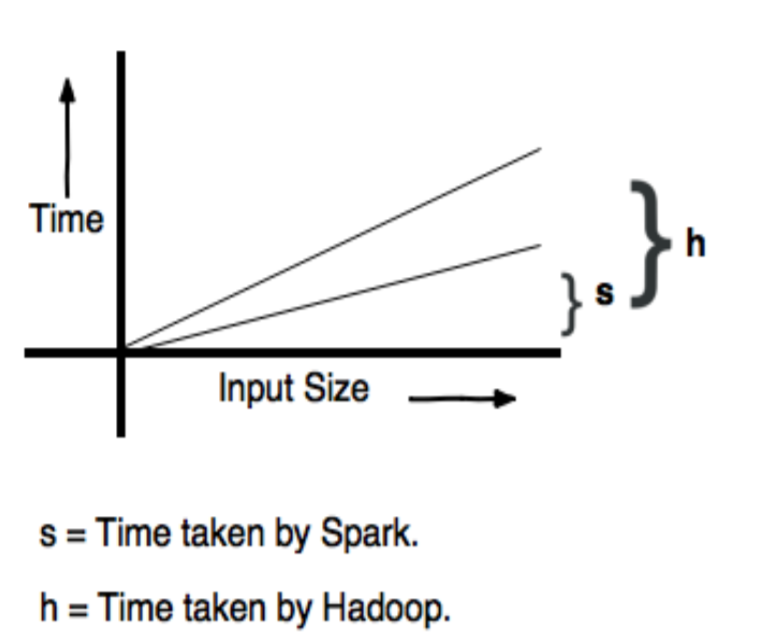


**PROCESSING TIME COMPARISION**

The map reduce model designed to work with Hadoop stores data into disk after every map and reduce task. Fetching the data from disk after every map task and reduce task is very much time consuming. This is a big disadvantageous to be associated with iterative algorithms.

****

**Fig: Shows the running time comparison between spark and Hadoop**



Time taken by Spark to complete the execution is approximately 10 times less than that of Hadoop.

**Execution Time for Hadoop:**

**T (n) = O (n2 )**

**Execution Time for Spark:**

**T(n) = O (n)**

**Conclusion**

**Reference**

Note that Spark does not require Hadoop, and it simply supports for storage systems that implement the Hadoop APIs.