Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

A Near Real-Time Big Data Processing Architecture

Xuming Xiu
[redacted]

April 30, 2020

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Latency has been an issue that many trading firms making efforts to solve. When trading data coming in the form of the streams, a system that can help to make decisions in a relatively short time before the next data point arrives would benefit a lot. Concerning Big Data, the characteristics challenge the industry from Velocity, Volume, and Variety. This paper prosed an architecture of a system that can quickly retrieve a large volume of the historical dataset, and performing near real-time aggregation as new data coming. This project targets the velocity and volume aspects of big data.

The paper explores the possible technical solutions to reduce the latency for trading firms. The performance of such a system is directly related to profit. The motivation of this project is to present a solution to help with decision making and keep the latency as low as possible. During the investigation of the technologies and the problem. Apache Spark and Kafka proved to have good performance in terms of real-time processing. There is also a methodology of using Geometric Brownian Motion to generate synthetic data to enrich the sample space. This project also uses a time-series database for real-time monitoring of the volatility. Finally, the system can aggregate 1 single column of 20 million rows with up 10 seconds in average in the Hadoop cluster.

Acknowledgements

I would first like to acknowledge my supervisor, Dr. Khurshid Ahmad, who has been support me during the entire project. He led to discover my interest in big data and supported me when I met difficulties. Sometimes, he explained more than once to me about the concepts that I had trouble understanding.

I also would like to thank my parents who supported me to study abroad. Without them, I wouldn't have such a fantastic opportunity to discover my interests. Finally, I also want to thank my girlfriend who has been taking good care of me during the busy time.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The Internet generates billions of Data every minute around the world. These data usually have the properties of high velocity, large volume, and multi-variant. These data always have valuable information along with them and waiting to be discovered. Time series is one important category of Big Data. Apart from the general characteristics of big data, time series Big Data is a sequence of discrete-time data that taken successively equally spaced points in time. It is observable that potential correlations can be discovered as the data points are collected at adjacent periods. When there is a large volume for such data in high-frequency streams, analysis can be challenging. Such time-series data can be collected from many different places. Some typical ones are like trading data, service logs, and traffic data. Many applications have the capability to support time series analysis.

1.1 Problem Statement

The characteristic of Big Data includes Velocity, Volume, and Variety [30]. These are also the major challenges of Big Data analysis. Streamed data can be even more challenging as it requires a system to be able to continuously handle the incoming data and perform real-time analysis. Trading data is typical streaming data. Besides, the system has to be able to retrieve the large volume of the historical dataset in a short time. The latency is the critical problem to streaming analysis. When the data arriving at a particular frequency, the result of the analysis has to be finished before the next data coming in. Especially in terms of high-frequency trading (HFT), the reaction time is critical. Many trading firms place the server next to the trading house to reduce the latency. The journal [9] mentioned a new measure to capture the reaction time which is called Decision Latency. The quickest HFT firm in their models first responds to profitable trading openings, taking all of the profits, while the slower participants arrive slightly too late. As a consequence, minor variations in trade pace are correlated with significant changes in company-wide trade revenues, with trading clustered among the fastest HFT industry. Another journal [15], the authors presented a conclusion that relative latency is important for success in trading on short-lived information. The study also suggested that lower latency can help with improving traditional market quality measures. Latency becomes a critical issue that can significantly benefit

market behaviours.

In summary, latency is the problem that this project aims to solve. When trading data are coming in as stream, a low latency decision-making mechanism is important. The decision has to be made before the next data point arrives.

1.2 Motivation and Objectives

The motivation is to reduce Decision Latency to achieve a near-real-time analysis system. In the previous section, we have introduced the latency in high-frequency trading and why it is important to keep as low as possible. However, from a technical point of view, Decision Latency can be optimized from the software level. The most direct cause of the Decision Latency is brought by data analysis. Typically, analyzing real-time data in real-time requires historical knowledge. Thus, the first objective is to collect the historical dataset for a given stock price. Next, since the historical dataset will be used to perform real-time aggregation. Therefore, the second objective is to quickly retrieve a large volume of the historical dataset into the system. When making a decision, specifically for trading data, volatility is an important metrics to reference. It indicates the potential profit or loss when making a transaction for the next trading. Therefore, the third objective of this project is computing statistical moments of the trading records with the latency with as low as possible. Most importantly, the computation of these statistical moments has to be finished before the arrival of the next data point. To simulate the trading process, there has to be a service to collect trading data in real-time. This is an important step to measure the Decision Latency as it will continue sending new data as a deadline of statistical moments computation.

1.3 Contribution

This project makes contributions in reviewing the related journals in low latency trading and trying to reduce Decision Latency as smaller as possible. While researching on other similar systems provides inspirations, a system architecture was designed and implemented to address the problem. The system has been proven to work as expected. However, it still has room to improve in terms of the automation process and resource allocations. These are pointed out in the discussion section. This is a working example for trading firms to have a near real-time time processing system to help making decisions based on a large volume of historical data within a relatively short time.

1.4 Content Structure

In the second chapter, the Big Data related technologies and literature are reviewed. Different technologies in terms of data storage (HDFS, NoSQL, and RDBMS) and big data processing tools (Spark and Hadoop MapReduce) are compared. These technologies and tools are fundamentals to achieve near real-time processing. There are also Benchmarking results included illustrating the most suitable technologies for this project. There are 2 similar projects introduced in this chapter as well. The real-time traffic data analysis project has demonstrated the benefits of using HDFS as storage technology. Another project High-speed log streams analytics provided the guidelines of the simulation real-time trading records using Kafka and SparkStreaming. This project is inspired by the above 2 similar works.

The third chapter describes the design and implementation details of the architecture. There are 2 major components in this project which are the Ingestion Platform and the Spark App. The purpose of the Ingestion Platform is to simulate an environment of real-time trading data stream where the velocity of data flow can be controlled. The Spark App uses Apache Spark [31] to perform real-time analysis. There is also an extra step to generate synthetic data because the size of the collected data is not large enough. The synthetic data is a simulation of the large historical datasets.

The fourth chapter is an actual case study where it uses the stock price from J.P. Morgan Chase & Co. to test the system. It indicated that the system running in a cluster that contains 3 nodes can aggregate a single column with 10 seconds. The evaluation part contains the system setup speed. It discussed the advantage and disadvantages of such a design. It also evaluates the system performance in terms of the latency. There is also a discussion regarding the challenges that this project resolved, the reasons to use these technologies that have been chosen for this project, and possible future work.

Finally, the conclusion chapter concludes that the Decision Latency could be reduced down to 10 seconds for a single column using such system and configuration. It also concludes the drawbacks of the system and suggests some possible working solutions in future.

2 Literature Review and Similar Work

This chapter includes some of similar work and review of the related literature about the Big Data technologies. There are great amount of previous work and papers to reference about big data. Some techniques and methodologies provide very insightful guidelines for this project. The most critical part of this project is the issue brought by properties of big data. There are some of previous work that provide similar uses cases for addressing these challenges in related to High Frequency Trading data.

2.1 Big Data Challenges

The definition of Big Data is given as massive data sets having large, more varied and complex structure with the difficulties of storing, analyzing and visualizing for further processes or results [30]. This definition provides perspectives of potential problems that are needed to be addressed when working with big data. The following three introduce the challenges of Big Data in terms of Velocity, Volume, and Variety.

2.1.1 Velocity

The first problem facing is that the issue of velocity. The velocity is directly related to latency when data coming as stream. The limitation of velocity is not only in terms of the data flow but also is required for all processing unit. A typical question is how fast we can process the incoming data [30]. Latency is a critical challenge that needs to be solved. There are some components that may slow down the computation from different aspects. The latency comes from hardware level mostly related to the network. As the the nodes within the cluster are connected via local network, a decrease in time taken for a packet traverses switches would result in faster communication [32]. From operating system point of view, a well configured scheduler and resource allocator will significantly increase the speed [32]. Hadoop Yarn is a perfect choice as a role of scheduler no matter in cluster or a single machine.

2.1.2 Volume

Other than velocity, volume is also a critical feature of big data [30]. Volume generally means the size of the big data. With that being introduced, the challenge facing with volume is the storage. Since the data tend to be very large, traditional file systems or databases hosting on single node is not able to handle such amount of data. The enterprise level big data could be in tarabytes or even petabytes. These data are even more complex rather than just big in file size. They could cover multi dimensional data which makes it more difficult to perform analysis against these data [25].

2.1.3 Variety

Variety refers to the data coming from multiple sources and the data itself consists multiple types both structured and unstructured [30]. To analysis big data that has such characteristic, the Dimensional Data Analysis (DDA) technique is recommended [14]. The algorithm basically works as follows:

```
for dataset in each schema:
    if number of rows > 1:
      get number of rows
      get number of unique columns
      get number of values
    else:
      continue;
```

Once the metrics are collected, they are used to compare with ideal and vestigial values to determine approximate structural model. The performance benchmarks indicated that using such algorithm, analyst can finish data ingestion 2 million data points within 400 seconds [14].

With more study in Big Data, there are more challenges discovered in Big Data. There are extra dimensions added to the Big Data concept [5]. *Variability* refers to inconsistent of data flow. It can be hard to manager particular for unstructured data. *Value* indicates data cannot be meaningless. There are also valuable information can be concluded from big data. *Veracity* refers to the quality of data. With the size of of data growing large, the data quality can be challenging. *Validity* can help with making correct and accurate decisions. *Visualization* is a task to present the data with the most intuitive approach. Despite new studies keep adding more characteristics to Big Data concept, this project will focus only on Velocity, Volume, and Variety.

2.2 Big Data Technologies

2.2.1 Databases vs Distributed File Systems

The database has always been playing an important role in data storage technologies. A relational database can store data in table format. The great advantage of relational databases is that it supports schema which makes data modelling easier. One of the aims of RDBMS is to enforcing data integrity. To achieve this, a proper design of tables is always required by RDBMS. This may be achievable when the data is relatively small. However, when dealing with big data, this can be significantly expensive as the normalization process requires table joins and searching for keys throughout the entire dataset. NoSQL is expected to be out-performed than RDBMS since there are no requirements of normalization. Besides, the variety also challenges the RDBMS since the database schema of the dataset can be difficult to design.

NoSQL uses a new way of storage. It supports key-value storage, document-oriented storage, wide-column storage, and graph database. MongoDB as one of the most popular document-oriented database has grown very fast with its scalability and reliability [23]. A comparison between RDBMS and NoSQL database indicates that the difference in average time CRUD operations of NoSQL is increasing as the size of data increases [12]. Therefore, a valid conclusion can be drawn that in NoSQL database is more suitable than RDBMS for big data applications.

With the development of the hardware, this challenge can also be addressed by using a cluster. Instead of using a single machine to store everything, a group of computers that are physically distributed and connected by a local area network (LAN) are used to share data and storage resources. This is also known as the Distributed File System (DFS). Larger main memories with less expensive price enable the exponential increment in caching performance. Optical disk and optical fibre networks make it faster to access the resources and to communicate among the nodes. Battery-backed memory can enhance the reliability of main memories caches as well [20]. As the hardware development reaches the bottleneck, the software design level has been put into consideration.
Other new technologies were invented to deal with volume issue for the past years. Google developed Google File System (GFS) which is believed to be outperformed than Hadoop file system to address their challenge [10]. GFS integrates MapReduce and offers the capability of byte-stream-based file view of big data that is partitioned over several hundreds of nodes within a cluster.

2.2.2 MapReduce

MapReduce is introduced as one of the approaches to process big data efficiently. Notably, MapReduce as the programing model works independently with storage layers. MapReduce can be considered as a monument in the history of Big Data technologies. MapReduce is a programming model for processing a large volume of data. There are 2 phases in this programing model namely Map and Reduce. Both phases allow programmers to customize the function to achieve the goal. Map phase takes inputs and transforms inputs to the key-value pair format. Then the key-value pairs received by the reducer and performed aggregations by the reduc. Before the key-value pairs received by the reducer, there is also a step in between whose task is to shuffle the key-value pairs to consolidate records from the mapper. MapReduce is expected to have a good performance under the parallel environment as mapper and reducer are separately doing tasks. A survey [19] investigated the MapReduce programming model from various dimensions include usability, flexibility, fault tolerance, and efficiency. The survey took the implementation of MapReduce, Hadoop, to examine the performance under the parallel processing environment. It concludes that as Hadoop uses checkpoints frequently, the efficiency could be dropped by I/O operations. However, the pros of using checkpoints can significantly increase the ability of fault tolerance and scalability. When using MapReduce programming model to process big data, the clear drawback comparing to DBMS is that it is schema-less [26]. A MapReduce job can be triggered immediately when the data is loaded. At this stage, there is no data modeling involved and hence the data is not indexed by the MapReduce job. This faster in processing large volume of data but insufficient for modeling.

2.2.3 The Hadoop Architecture

The combination of MapReduce and DFS has made huge progress in processing a large volume of data. Hadoop is an excellent implementation of MapReduce has connected MapReduce and distributed file system successfully. The ecosystem even supports other big data technologies like Cassandra and Hive [6]. The core components of Hadoop includes Hadoop MapReduce, Hadoop Distributed File System (HDFS), and Hadoop Yarn for resource management and job scheduling. Hadoop makes MapReduce running more efficiently on distributed nodes with the help of Yarn. Yarn can dynamically allocate resources to applications on demands. Comparing to the pure combination of MapReduce application and DFS, it can utilize the resources and application performance. Yarn default scheduler processes jobs in First In First Out (FIFO) policy. The global ResourceManager is responsible for taking jobs submitted by users, scheduling these jobs, and allocating resources to them. In each node, there is a NodeManager installed monitoring and reporting the current resources available to the global ResourceManager to assign the resources to

each application. NodeManager also takes control of Resource Containers The ApplicationManager is responsible for negotiating resources for each submitted tasks to ensure utilizing the tasks.

While Hadoop MapReduce focusing on the processing of big data, HDFS takes care of the data storage. A diagram of the HDFS architecture has been shown in Figure 2.1. HDFS consists of NameNode and DataNode where NameNode manages the name, location, the permission of each block of a dataset and DataNode stores replication of data blocks in memory and process I/O operations [36]. HDFS provides reliable fault tolerance ability by 2 policies, replication and checkpoint recovery. Since HDFS works on multiple nodes, the data could be replicated anywhere on DataNode within the cluster. With the help of the NameNode, retrieving the data could be done efficiently. Using checkpoint recovery help to improve the fault tolerance by rolling back the last saved synchronization point and restart all transactions if a failure occurs.

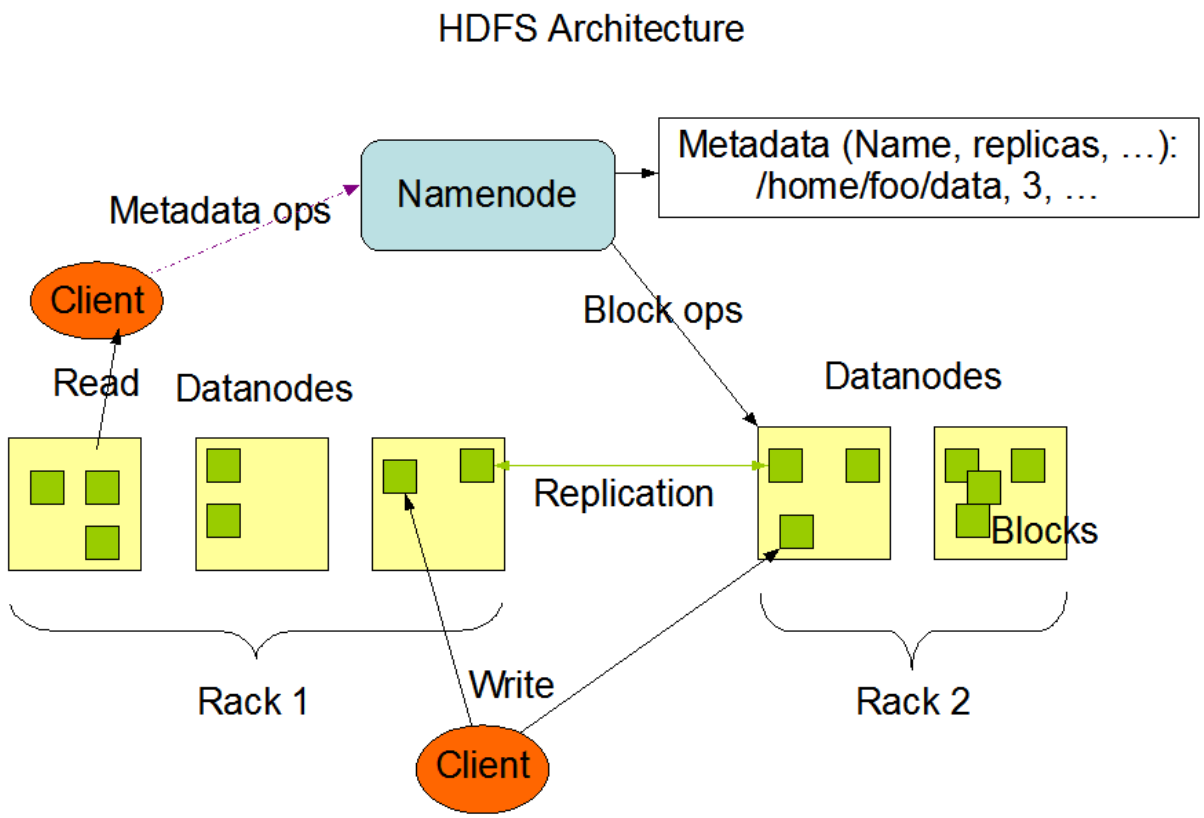

Figure 2.1: The HDFS Architecture

In summary, the optimized hardware and choice of frameworks can significantly reduce some of the challenges of Big Data. The Hadoop ecosystem provides the ability to integrate the distributed computation resources and storage resources to allow users processing and storing a large volume of data. The combination of the MapReduce programming model and Distributed File System (DFS) will make significantly huge improvements in terms of performance. An analytical study [33] compared different major DFSs and concludes that MapReduce is the perfect framework to apply to DFSs to maintain the performance and satisfy the requirements.

2.2.4 Spark

Hadoop MapReduce is not the only MapReduce paradigm, there are many implementations of MapReduce in the market holding different purposes. Among all of them, Spark is famous for its processing speed. While the Hadoop MapReduce reads in files as stream and stores each record in the result file after reduce, Spark uses in-memory processing to increase the speed. A very important concept in Spark is Resilient Distributed Dataset (RDD) which is an immutable and fault-tolerant collection of elements [29]. These RDDs are created when loading the files from the DFS. Users have options to persist them in memory. Since the RDDs are immutable, in the later releases, Spark introduced DataFrame API which allows users to perform aggregations, joins, and other relational operations. The best feature of DataFrame is that it supports schema inference. The data lives within the DataFrame is almost equivalent to a relational table. Thanks to this feature, there is no need to use Java/Scala serialization when writing the data to disk (or distribute on clusters) as Spark in nature knows the schema. Moreover, when the user defines the mapper and reducer functions, no execution occurs until the user calls an output operation. This is known as the logical plan.

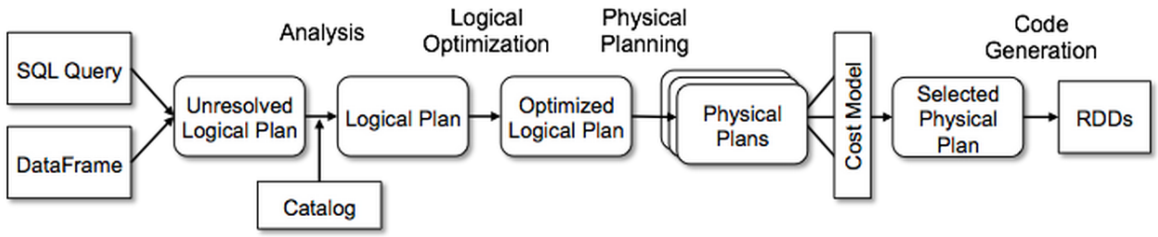

Figure 2.2: The Workflow of Catalyst Optimizer

SparkSQL is a framework associated with DataFrame API. It supports multiple data sources as input. It also supports user-defined types as semi-structured data. Therefore, when

loading the data into memory, it will automatically deserialize to Java/Scala objects. A very important feature in SparkSQL is the Catalyst Optimizer [8]. The Catalyst Optimizer is designed for tackling different optimization tasks in Big Data. The implementation of Catalyst Optimizer is based on Tree data structure. The nodes in the tree can be manipulated by rules which are the rule. Rules are nothing but functions to transform the input tree to another output tree and finally to an RDD. As Figure 2.2, the Optimizer starts to perform logical optimization first after resolving the logical plan. This is done by analysis rules to determine which operations to perform first when having multiple transforms/actions. After this phase is done, it takes the optimized logical plan to generate one or more physical plans. Then Spark Engine will evaluate these physical plans against the cost model. finally, the most optimized plan is selected and used for execution [8]. SparkSQL reduces latency by using in-memory caching and query optimization.

SparkStreaming is a scalable fault-tolerant streaming processing tools also works for batch workloads. SparkStreaming will receive real-time data from different sources and transfer the processed data into different storage technologies such as database, or live dashboard. The high-level abstraction of SparkStreaming is Discretized Stream which is known as Dstream object in Scala [31]. The major aspects of SparkStreaming are fast recovery from failures, better load balancing and resource usage. More importantly, it supports interactive queries between streaming data and static datasets. The use of SparkStreaming will help to achieve near-real-time analysis for big data.

While both Hadoop MapReduce and spark are designed for big data processing. However, their use cases are different in many ways. Spark supports in-memory processing and linear processing which means there is no need to load the data in every time to perform a query. It is especially good for near real-time processing when more hardware is available. However, Hadoop MapReduce is designed for more general purpose. A comparison conducted came out a conclusion that Spark is the framework for processing a large number of data after considering various algorithms[35] because of in-memory processing. When processing large scale of data from a different resource, the performance of Hadoop MapReduce is observed for inefficient in the analysis.

2.3 Previous Work

There are several applications which focusing on streaming data analysis. Although they have applied in different areas, some design techniques worth referencing. When designing the streaming data analysis architecture, there are two major challenges need to put into consideration. The main challenges in the storage layer are the capability of storing a large volume of data, data loss, and consistency. A proper designed DFS like HDFS could resolve the above challenges together with distributed infrastructure. From the processing layer perspective, latency, scalability and fault-tolerance are primary requirements.

2.3.1 A real-time traffic data analysis

A similar application of big data real-time streaming analysis in traffic data area [21]. In these projects, the authors emphasized the velocity of Big Data. The data source is collected from various traffic sensors and are preprocessed. There are several phases involved in this architecture.

- **Ingestion and aggregations** Apache Flume is chosen as the technology to ingest the data from different sources. The data source is from GPS data, from taxi and road weather data, from weather stations and then ingested and forwarded to Spark for preprocessing. At the same time, data are stored as a raw file in HDFS.
- **Real-time analysis** In this phase, the traffic flow indicator values will be calculated in real-time with the input given as GPS event. The main operations are based on Sparks's map, reduce, and filter. The results are also stored in HDFS.
- **Periodic batch analysis** There are also batch analysis carried out periodically. This is executed separately in a child process so that the error resiliency could be improved. The latency results higher when using Hive over Impala.
- **Archiving data** The data then is archived to Impala data tables.

The main latencies in this project are accumulated in preprocessing, Impala table updating, query execution, and Ingestion. The main reasons for these latencies including the configuration of the Messaging system, I/O performance, and network resource. Our project can inspire from this project by optimizing the I/O performance. This can be archived by choosing the different size of partitions of the raw file according to the requirements. Since our project will be running on the cloud environment, more enhanced computing resources will be available. Moreover, this project has proved that HDFS is a perfect choice of storage technologies for our project. However, it remains doubtful if it is suitable in terms of methodologies of the ingestion phase because we don't have a large volume of streaming data.

2.3.2 High speed log streams generated from web

As the development of web technologies, there are always valuable insights in logs of web-based applications. These logs will help to find usage patterns, potential failures, and so on. This project proposed architecture for high-speed web log analysis [3]. The overall architecture consists of a Kafka Server for accessing weblogs, SparkStreaming for processing the incoming logs in memory and then output the results to storage places. The most critical part of this project is the usage of Kafka. As the velocity can be fully controlled by using a message queue, which is very similar to our project goal, Kafka is an ideal message tooling. Moreover, Kafka provides large scalability by distributing messages on different machines. It also has a great mechanism for preventing data loss by using replicated and persistent storage [7]. The results of the projects also proved Kafka is the most suitable tool for message queue and satisfy our requirements. The details about the usage of message queue will be introduced in the next chapter.

2.4 Summary

This chapter reviews the possible technologies to build a real-time analysis system to reduce Decision Latency. The comparison between Spark and Hadoop MapReduce suggests that Spark is more suitable for this project because of the in-memory processing. HDFS have better performance in terms of storing a large volume of data. However, it is schema-less. It can work with Spark and make use of Spark-SQL to set the schema of the data manually while maintaining the efficiency of processing. The review of the 2 examples of similar architectures inspired the design of this project. Kafka works as a messaging system can be used to simulate the real-time trading records and HDFS can be used to store the historical data with large volumes.

3 Design and Implementation

This chapter will present the methodologies of solution design, a detailed introduction of the proposed architecture, and workflow process. There are many types of streaming data. The data source could be from sensors on IoT devices, service logs, or even financial data. The common nature of the streaming data is that they are time-series data. With this being introduced, the proposed architecture will focus on financial trading data and perform a real-time calculation of volatility. As it has discussed in the previous chapter, the main challenges in for streaming data analytics occurs to the storage layer and processing layer. Therefore, this project aims to design an architecture using available technologies to exam the latencies and where they might happen. This project also highlighted how cloud environments can impact on latency.

3.1 Data Source

3.1.1 Data Collection

Historical stock prices are extremely valuable. It reflects the performance of the company in the past. Such data usually not available for free or only available for a limited period of times. The maximize the data volume, this project uses the intraday stock prices with 1-minute bar. The historical stock price provider is IEXCloud [16]. This platform offers a free plan with a limited number of request. The APIs available for intraday is designed to request the minute file on the current day. However, it is possible to configure to get the historical data if parsing a specific date in the request body. Therefore, the logic is for a particular stock, keep requesting for the records on previous day until there is no record available. The more historical data we have, the more synthesis data we can generate. IEXCloud also provides other APIs for market indexes, cryptocurrencies, options and so on. However, due to the limited number of requests, another open-source platform needs to be used for this project.

Alpha Vintage [34], is a provider for real-time and historical data on stocks, forex, and cryptocurrencies. However, this platform does not support historical data in 1 minute for a

relatively long time. Some tests conducted using this platform indicates that Alpha Vintage will provide historical data for 1-minute stock prices up to a week. This will not satisfy the requirement of using big data. Therefore, the combined usage of the above 2 is essential.

3.1.2 Data Fusion

Using IEXCloud allows collecting historical data for the past 1 years. However, the data collected can not satisfy the requirement of Big Data because the volume is not large enough. The nature of the stock trading data is time-series data, data is in series of particular periods or intervals. Trading data as such series has the property of discreteness. The Efficient Market Hypothesis indicates that the history of a stock price can be fully reflected in current prices and if any new information about the stock, the market will always respond [11]. Given these two properties, a valid assumption can be made that current price can also have an impact on the expected future prices. However, there remain uncertainties brought by unexpected events.

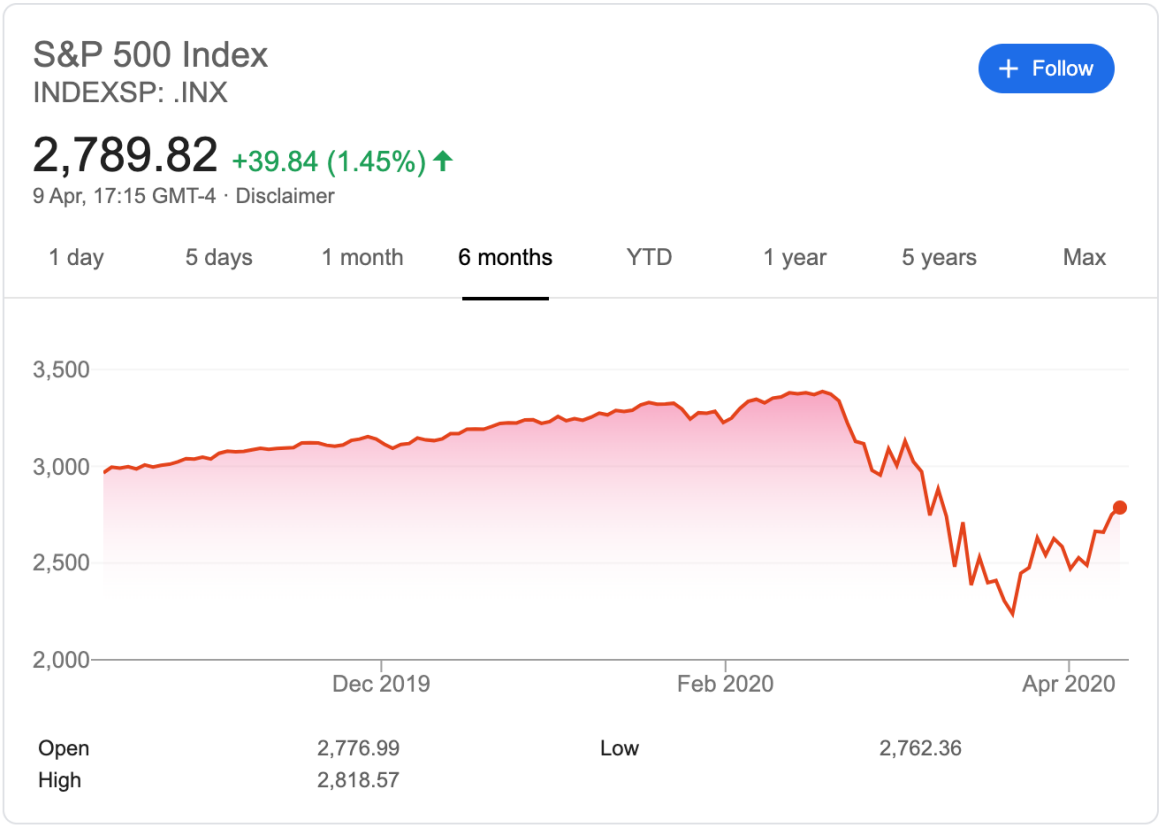

Figure 3.1: S&P 500 index for the past 6 months

The above diagram 3.1 shows the index prices changing over the past 6 months where the y-axis is the price and x-axis is the time. S&P 500 indicates the stock performance of the large 500 companies listed in the US stock exchange. This is some way can present an overall picture of the market. As mentioned before in the EMH, the market will respond with the information about the stock. Due to the global pandemic happened in March 2020, the index drastically decreased. Later in early April, the US government announced the Quantitative Easing(QE) to save the US stock market. Then the indexes went up to reflect such effort made by the government.

From the diagram 3.1, it can also be concluded that the change of the stock prices depends very much on the previous state. Therefore, the change of price is following the Markov chain where the Markov chain describes the stochastic process of changing.

Geometric Brownian Motion

Geometric Brownian Motion model is a stochastic model with continuous time, where the random variable follows the Brownian motion [4]. The stock prices also have a similarity with the Markov process which is also known as the Wiener Process. Traditional Brownian Motion describes the random motion of a part in a fluid. The mathematical properties of the stochastic process in linear dimension can be defined by Geometric Brownian Motion. The stock prices behave similarly to the stochastic process in continuous time where they both have long term trend and short term fluctuation. The study [4] shows that using Geometric Brownian Motion can have a reliable prediction of a short term with Mean Absolute Percentage Error (MAPE) less than 20%.

There are 2 components in the Geometric Brownian Motion model which are Drift and diffusion. 2 metrics are important for this model which are the mean and standard deviation of the return. The return of the stock prices at time k is given by the following equation:

$$r_k = \frac{S_k - S_{k-1}}{S_{k-1}} \tag{1}$$

Where S is the price of the stock.

The mean μ of the return can be calculated by summing of the return within the historical range divided by the total number of data points.

$$\mu = \frac{1}{|k|} * \sum r_k \tag{2}$$

The mean or the expected value can be used in the drift function to reflect the longer-term trend. If the mean is negative, it indicates the return is negative on average within the

historical period. When the average return is negative, it suggests that there will be a loss.

The positivity of the mean determines the if the stock price goes up or down. However, the stocks prices never grow smoothly. Standard deviation will help to incorporate random shocks. Standard deviation determines the magnitude of the movement. The equation of standard deviation σ is given as:

$$\sigma = \sqrt{\frac{1}{|k|} * \sum (r_k - \mu)^2} \tag{3}$$

With the mean and standard deviation being introduced, the drift function is defined as follow:

$$drift_k = \mu - \frac{1}{2}\sigma^2 \tag{4}$$

Since the drift function (4) contains only μ and σ, this function is constant. The value determines if the stock prices is going to increase from a longer-term perspective. If not applying random shock to the series, the series will change smoothly. Recall the Markov process of the stock prices indicates that the current state is impacted by the previous state. Therefore, to use the drift can be used to determine the next state. Given the previous stock price at time *k-1*, to calculate the price at *k*:

$$S_k = S_{k-1} * e^{\mu - \frac{1}{2}\sigma^2} \tag{5}$$

Diffusion can be used to reflects shorter-term fluctuations. such fluctuation normally is introduced by random shock. Recall we discussed previously, the market will always respond to the news. Therefore, the curve cannot be smooth. It is changing almost every time. However, different random event may apply different scale of impact on the stock price. In the diffusion function, the standard deviation σ is used to control the magnitude of the fluctuation in a constant range.

$$diffusion_k = \sigma * z_k \tag{6}$$

where z is the effectiveness of the random shock at time k Since the drift function is constant, the diffusion process will help to simulate the trend under different scenarios by applying different scale of random shocks. In summary, to model the stock price at time *k*

given the value at *k-1*, adding diffusion term to function (5) gives:

$$S_k = S_{k-1} * e^{(\mu - \frac{1}{2}\sigma^2 + \sigma * z_k)} \tag{7}$$

The above equations allows to make prediction for the next data point. If to predict of the stock price for the next 4 times, the function is given by:

$$S_{k+4} = S_k * e^{(\mu - \frac{1}{2}\sigma^2 + \sigma * z_k)} * e^{(\mu - \frac{1}{2}\sigma^2 + \sigma * z_{k+1})} * e^{(\mu - \frac{1}{2}\sigma^2 + \sigma * z_{k+2})} * e^{(\mu - \frac{1}{2}\sigma^2 + \sigma * z_{k+3})} \tag{8}$$

the above function can also be generalized to make the prediction of the stock price at time k given the start price *S0*:

$$S_k = S_0 * \prod_{i=1}^{k} e^{(\mu - \frac{1}{2} * \sigma^2 + \sigma * z_i)} \tag{9}$$

Notably, the drift term is constant and each time the diffusion updated by multiplying its previous value. Therefore, function (9) can be rewritten as:

$$S_k = S_0 * e^{((\mu - \frac{1}{2} * \sigma^2) * k + \sigma * \sum_{i=1}^{k} z_i)} \tag{10}$$

The final function 10 will be used to generate synthetic series. The stochastic process then can be simulated by Geometric Brownian Motion. The Geometric Brownian Motion does not guarantee the mean value of the generated series remaining the same as the original series because of the random shock. In another word, the longer-term trend of the generating series can be different from the original series. Although the theoretical value of the mean should be 0 in an infinite amount of time, the mean of generated series can still be greater than or less than 0. This is because it is impossible to have an infinite number of data points.

In more general scenarios, Geometric Brownian Motion can also be used to predict the future changes of the stock prices. This is based on EMH where the past prices are already interpolated. The stock prices changing is a Markov process. Therefore, the prediction can represent the future changes at some level based on the assumptions of EMH and Markov process.

3.2 Simulate Real-Time Data Streams: Ingestion Platform

3.2.1 Overview

Realtime Streaming data is another important aspect of this project. Realtime is the problem that needs to be resolved which is related to the velocity of the big data. When real-time trading data arrive in the system, the system also has to be able to analyze the volatility and risk of real-time. This component is responsible to create real-time data streams that can be configured in terms of the velocity and data source. This a Spring boot web application that integrates with Kafka. It can collect the data source in various format and from various sites. There are 3 endpoints available at the moment.

1. Retrieve Data **GET /retrieveData/symbol/function/timeInterval**
   This allows retrieving the trading records for a specific symbol in the given time interval in JSON format. Functions are available for *TIME_SERIES_INTRADAY*, `TIME_SERIES_DAILY`, `TIME_SERIES_WEEKLY`, and `TIME_SERIES_MONTHLY`.The available options for time interval are 1 minute, 5 minutes, 15 minutes, 30 minutes, and 60 minutes when `TIME_SERIES_INTRADAY`.
2. Download File **GET /retrieveData/symbol/function/timeInterval/download**
   The path variables are almost the same except the download at the very back. This will download the record as a CSV file. Then the file will be scanned line by line. It will also send each line as a message through Kafka.
3. check files **GET /file/symbol**
   This will allow getting all the files for a particular symbol. This is not the main endpoint but an endpoint for debugging purpose.
4. check files **GET /file/symbol/date**
   This will allow getting all the files for a particular symbol on the user-defined date. This is not the main endpoint but an endpoint for debugging purpose.

3.2.2 Kafka

delivering real-time trading record. A messaging system is a producer-consumer model where the producer sends the message to the consumer and the consumer receives the message from the producer. Kafka as an open-source messaging system has been used a lot in big data architecture. It is designed to be fast, scalable, and durable [7].

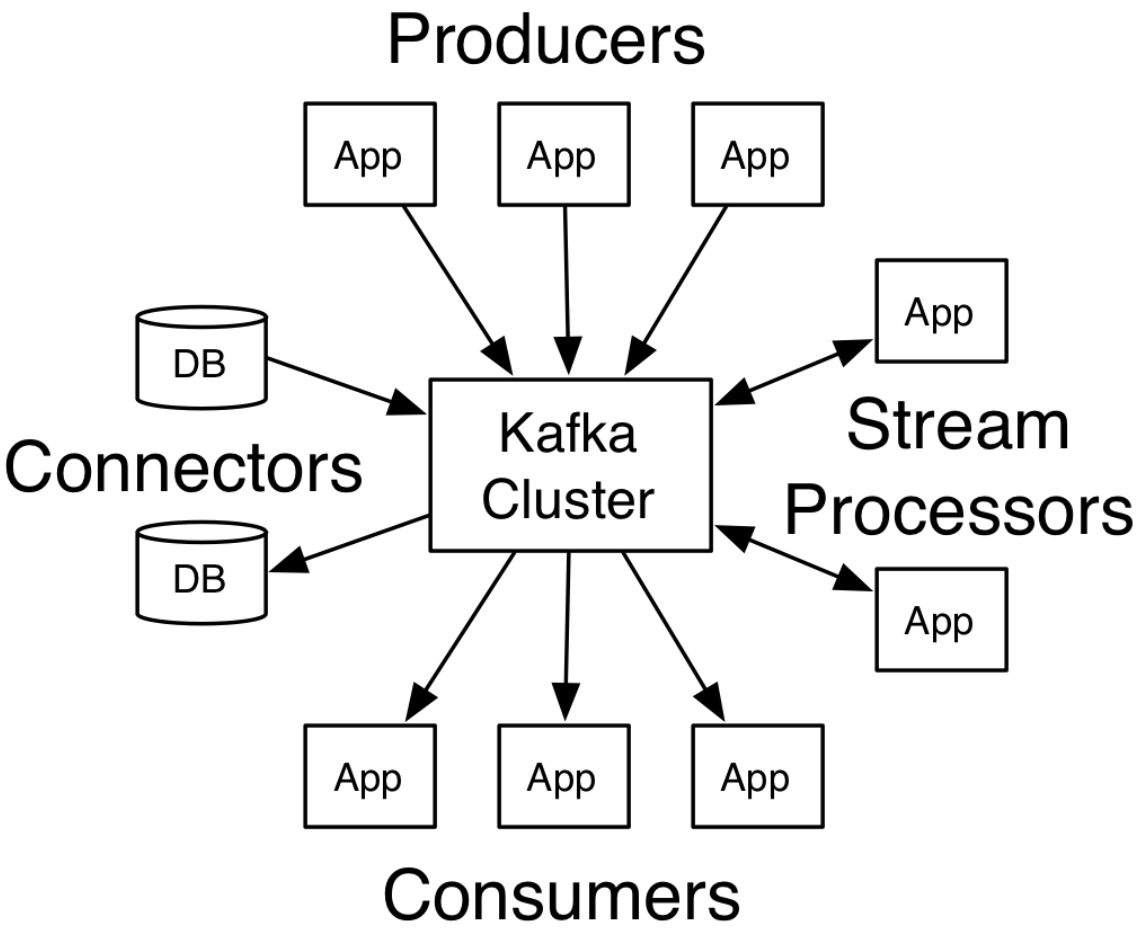

Figure 3.2: Kafka

The producer will publish a message to one or more Kafka topics. The consumer will be able to receive the message if it subscribes the same Kafka topic. Using a Queue data structure allows the message being processed sequentially. A Kafka topic lives with Kafka broker together with the partition id. Kafka is also a distributed architecture that can supports message replication among the nodes in the cluster. This keeps the availability of the data. Based on the above features, Kafka can help to achieve delivering the trading record in real-time.

3.2.3 Implementation

Setup Kafka Cluster

The features of Kafka have been introduced previously. To set up a Kafka Cluster on a local machine, the Docker container was used. Using the Docker container allows creating isolated virtual environments. There is no requirement of having multiple physical machines if using a swarm of Docker engines. The virtual instances also included a Zookeeper instance

which manages the Kafka broker. The message logs are partitioned within the virtual Kafka instances and mounted into the host machine. The port number is also exposed to the host machine to make it accessible from outside of the Docker container. Docker-compose is a YAML file that contains the configurations about the each Docker containers including base image, dependencies, network configurations and interaction between the container and the host machine.

Migrating Kafka to Google PubSub

Google PubSub is another messaging system developed by Google. It follows almost the same design of Kafka but provides large storage capability in the native cloud environment. It also provides a UI to allow tracking messages. These are huge advantages compare to local Kafka in a Docker container. Thus, Kafka is replaced by Google PubSub.

Web Service

Spring Boot is a popular web application development framework. It is a Java framework that allows developers to focus on the business logic without worrying too much on the other details. The most important concept in Spring boot is Aspect-Oriented Programming (AOP) and Inversion Of Control (IOC). AOP allows modularizing the functions that are being used multiple points of an application. This is also known as cross-cutting concerns [1]. The common functionalities are defined in one place when using AOP. However, these functionalities can be adjusted without modifying the class to be used in the new features. In this application, there are 3 modules created and implemented. FileManager is responsible for managing files in the folder including scan file records. IngestionManager is responsible for retrieving the data from Datasource and download the data in either CSV or JSON format. The messaging module implemented the Kafka API and is responsible for sending and receiving the message.

When there is a GET request received by the download endpoint managed by IngestionManager, Kafka will send a message containing the file path, symbol name, and function when this the download is completed. IngestionManager will also generate a timestamp that is unique to every downloaded file. The message will be received by the web application itself. Kafka provides Java APIs which have call back functions to check if a message has been sent successfully. A log message will be printed either onSuccess or onFailure.

Listing 3.1: Kafka message sender callback

```
future.addCallback(new ListenableFutureCallback<>() {
            @Override
            public void onFailure(Throwable throwable) {
```

```
                LOGGER.error("Unable to send message: [{}]",
                    throwable.getMessage());
            }

            @Override
            public void onSuccess(String message) {
                LOGGER.info("Sent message [{}]", message);
            }
        });
```

The message body contains the symbol, the highest price, the lowest price, opening price, closing price and the trading volume; After the message is successfully sent, FileManager will start working on reading the file and extracting the records. FileManager Interface will manage every file in every folder as if a data pool manager. When the message is received by the FileManager, it will use the file path contained in the message payload to read the file. Then, the content of the file will be saved in memory as a HashMap. Typically, a file of the 1-minute bar in a trading day will only have up to 480 trading records. This is tolerant for JVM to load the entire file. The HashMap will be parsed to message sender again. This time, each Entry in the HashMap will be sent as a message in user-defined frequency. To achieve this, it implemented Java Runnable interface to control the frequency. An integer value is passed into the application as Springboot properties [2]. This thread will sleep for the interval passed in. Also, the return of the stock price will be calculated and saved into the InfluxDB.

Listing 3.2: Thread sleeping time management

```
try {
    LOGGER.info("The next message will be sent after [{}]
        milliseconds",getSleepTime(timeInterval));
    Thread.sleep(getSleepTime(timeInterval));
    } catch (InterruptedException e) {
         LOGGER.error("Thread interrupted");
    }
```

If no value parsed into the application, it will use the frequency in the file. If the requested file is 1 minute, the message will be sent every minute. The message will be received by the Spark App to perform real-time analysis. The downloaded files will also be categorized by their symbol since the symbol is unique for every stock.

This service is one of the core components of the entire architecture. It also has integrated with Swagger UI to document the endpoints that are currently available. Data Ingestion is

very important to satisfy the variety and the velocity requirement of big data. The Ingestion Platform itself is highly configurable in terms of data sources, stock markets, as well as the velocity of simulating the real-time data. The platform can be used as a stress test tooling for big data applications by adjusting the frequency of sending messages.

3.3 Real Time Processing: Spark App

The real-time analysis application was chosen to use Apache Spark. As it has been introduced in the previous chapter, the benchmark results of Spark and Hadoop MapReduce indicates Spark has better performance when processing the real-time data stream. Due to the feature of in-memory processing, it shows faster in terms of processing. Spark Streaming is another reason for this framework being used. By using this, the application can consume the messages sent by Kafka. [31].

To use Spark in the application, the spark context has to be configured. The configuration includes spark checkpoint, Google Cloud credentials, Application name, and the number of thread to be used int local machine. Spark context also allows choose configuration at runtime depends on different requirements [8]. In this application, there are 2 configurations required. The first one is for spark streaming context. When running spark locally, the checkpoint directory is set to a local folder. When running spark on the cloud, this is set to a folder in Google Cloud Storage (GCS). The application is set to local[*] which means as many as worker threads as logical cores available [31]. Another configuration is the Google Cloud Storage configuration for loading the historical file. Since the Google Cloud Storage backed up GFS, providing the link starting with *gs://* link to the file location can load the file. The Hadoop Distributed File System is also implemented based on GFS. Therefore, it is also important to add the Hadoop configuration when loading the file from GCS. After the SparkStreaming context has been created, it will be connected to the messaging system to consume the message. Spark streaming provides *Dstream* which is a high-level abstraction of Discretized Stream. When the input stream is received by Spark Streaming, it will be divided into small batches and then processed by Spark Engine.

3.3.1 Receiving the Messages

A *DStream* contains a series of Resilient Distributed Datasets (RDD). The features of RDDs have been introduced in the previous chapter. Notably, there are some operations can be applied to a *Dstream* object. The operations can be categorized into Transformations and Actions. These operations often have no impact on RDDs since RDDs are immutable. Map is one of the Transformations. When mapping a *Dstream*, it creates a new *Dstream* of different objects. The map function takes the original RDD type as a parameter and returns the target RDD type. Another function in Transformations is Filter. The filter function also

takes in the original RDD type and return a boolean variable. This function is applied when filtering for the resulting RDDs. In this application, the source of the *Dstream* is from the Ingestion Platform in the forms of the message. In the map function, a deserialization method is defined to convert the input stream of String type to a Java Object. In the code, it defined a class GCSRecord which contains all fields of the trading records. Spark can apply the map function in small batches. Figure 3.3 illustrates the process of applying map function to the incoming stream. The transformed *Dstream* will be used to perform real-time data analysis.

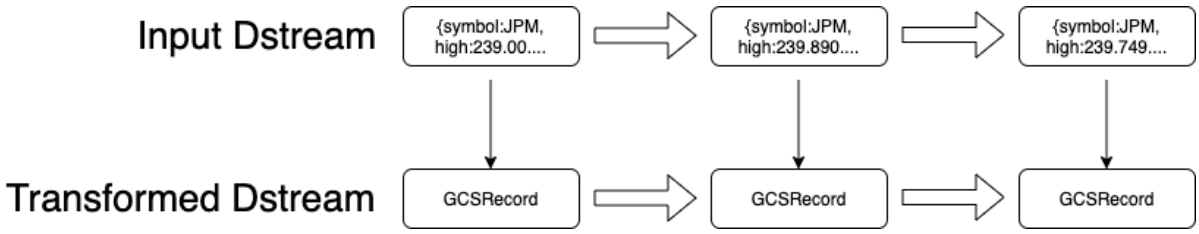

Figure 3.3: Mapping Dstream

In terms of the Actions operations, Action is a method to access the actual data in an RDD. Action operations are suitable for the small size of data [31]. The incoming dataset compared to the historical data is extremely small. Therefore, some Action operations can be applied. Reduce is one of the most important and commonly used one which is used to perform aggregations. The reduce function has to input parameters which are the RDDs and return the aggregated results. The reduce function was used to find the return of the stock prices. Recall the equation for calculating the return stated in equation (1), it requires 2

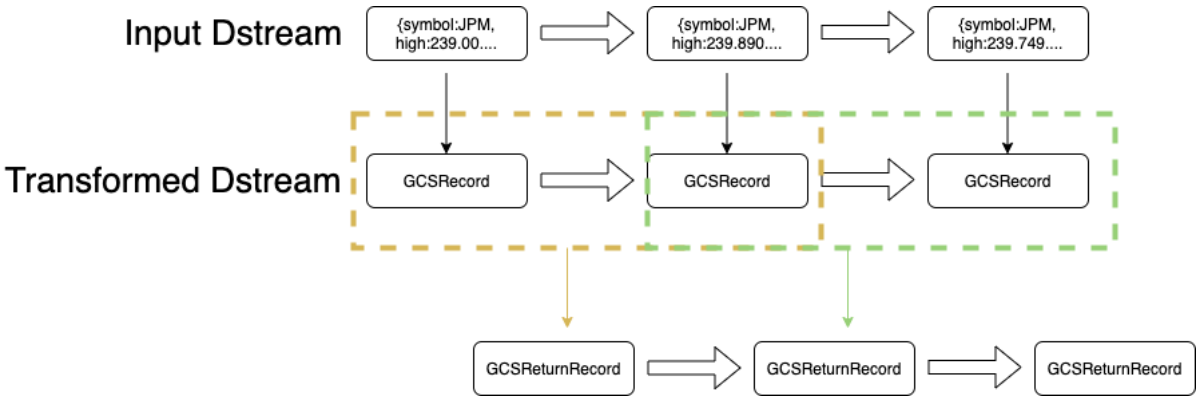

Figure 3.4: Reduced Dstream

parameters. The current stock price and the previous stock price. The figure 3.4 shows the workflow of aggregations over previous 2 GSCRecords to calculate the return. SparkStreaming provides a sliding window API which can be used. The window length of 1.5 minutes was defined. The reduce function will aggregate all of the RDDs within this batch interval. In the configuration of this project, the number of RDDs within this window should

be 2. After the return RDD is calculated, each RDD will be delivered into the InfluxDB instance. The timestamp is created at the time of which the InfluxPoint object is created.

3.3.2 Processing the Data

This project aims to calculate the volatility in real-time. Therefore, the statistics calculated include mean and standard deviation for both opening prices and closing prices and an additional field using mean divided by the standard deviation. When a new message coming in, it will use the historical data into the memory and append the new data at the end of the historical datasets. Since RDDs are immutable, Spark provides another API called *DataSet* [8]. The datasets object allows joining, aggregation, and other relational operations. RDDs are operated with functional programming constructs that include and expand based on map and reduce. SparkSQL provides APIs for aggregation including mean and standard deviation. The RDDs in *Dstream* processed sequentially. Each RDD will be used to created a new *DataSet* and union with the historical *Dataset*. Then calculate the average and standard deviation at the current timestamp.

Listing 3.3: Aggregation in real-time

```
Dataset<GCSRecord> tempDataSet =
    context.createDataset(gcsRecordJavaRDD.rdd(), gcsRecordEncoder);
dataset = dataset.union(tempDataSet);
double avgOpenReturn = dataset.agg(functions.avg("openReturn"));
double stdOpenReturn = dataset.agg(functions.stddev("openReturn"))
```

The historical *DataSet* was used repeatedly. Therefore, it could be cached into the memory for saving loading times. This transformation also creates a new *Dstream*. The new *Dstream* was connected by a sink to InfluxDB and store the calculated RDDs there. Figure 3.5 illustrates the overall flow of *Dstream*. Reactiveinflux [28] was used as SparkStreaming InfluxDB connector. It is a Non-blocking InfluxDB driver supported by Spark.

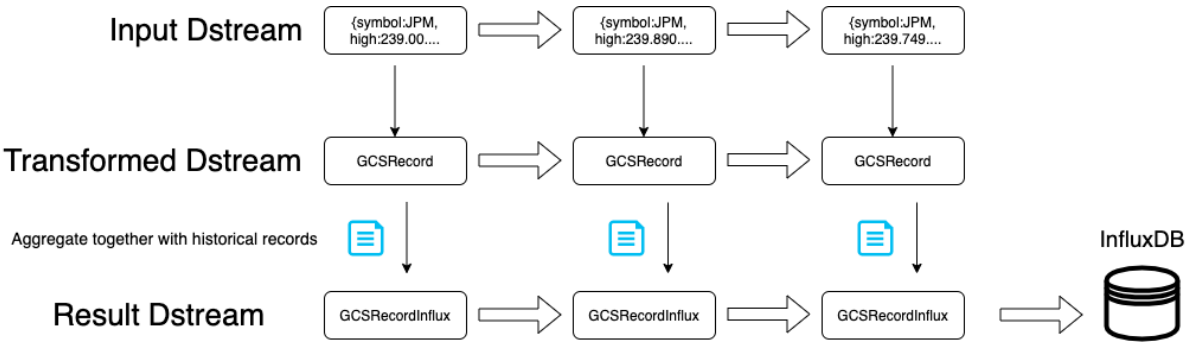

Figure 3.5: Overall Flow

3.3.3 Deploy to Dataproc

Dataproc [22] is developed by Google which provides a Hadoop or Spark Clusters in the cloud. It allows dynamical resources allocation for better performance. For this purpose, the Dataproc cluster was created with 1 master node and 3 worker nodes. The Spark job can be submitted through the *gcloud* command line. A Spark job is considered as an executable jar where all the classes, methods, and dependencies are packaged. Without further programmatic work, the methods of packaging have to be configured. *Maven* was used as dependency management as well as package management. Since Spark is written in Scala, it should compile Scala and bundle into the jar. The submission of a Spark job included the executable jar, a cluster name, and the Google Cloud credentials.

3.4 Result Persistence: InfluxDB

InfluxDB [17] is an open-source time-series database. InfluxDB has been used widely for log monitoring and other time series related applications. Influx data also provides visualization utilities. The data visualization tool also provided by Influx data has the functionalities of creating real-time visualization dashboard. InfluxDB provides SQL like queries so that users can query the database without requirement of learning new syntax. InfluxDB is also supported for distributed storage and strong performance. Therefore, it is a perfect persistent service in this project.

To deploy InfluxDB to Google cloud, a Kubernetes engine instance was setup with Ubuntu 18.4 Operating System. InfluxDB can be installed and configured via command line. After installation, the instance has to be accessible from other machines. So, the next step was to expose the port number together with external IP addresses.

In the Ingestion Platform, the Java client of InfluxDB was added as a maven dependency. When writing a record to the database, the connection was established programmatically. In contrast to the Ingestion Platform, the InfluxDB connection is handled by ReactiaveInflux in Spark App. The connection information is configured in the context.

Listing 3.4: Using Java client to write data into InfluxDB

```
public void saveToInflux(HashMap<String,String> record){
     InfluxDB client = InfluxDBFactory.connect(connectionURL);
     String databaseName = "final_year_project";
     String retentionPolicyName = "one_day_only";
     client.query(new Query("CREATE RETENTION POLICY " + retentionPolicyName
            + " ON " + databaseName + " DURATION 1d REPLICATION 1 DEFAULT"));
     client.setRetentionPolicy(retentionPolicyName);
     client.setDatabase("final_year_project");
     Point point = Point.measurement("time_series_return")
     .fields(new HashMap<>(record)).time(System.currentTimeMillis(),
        TimeUnit.MILLISECONDS)
     .build();
     LOGGER.info("writing to influx db");
     client.write(point);
 }
```

Listing 3.5: Using ReactiaveInflux to write stream in to InfluxDB

```
JavaDStream<JavaPoint> InfluxReturnStream = returnStream.map(message
    ->tradingRecord.getInfluxPoint(message,"return"));

sparkInflux.saveToInflux(InfluxReturnStream);
```

3.5 Summary

In this chapter, the core components and their implementations have been introduced. Figure 3.6 provides an overview of the integrated architecture of the whole design. The core components InfluxDB, Ingestion Platform, the method for generating synthetic data, and Spark App have been detailed explained regarding the implementations. The Ingestion Platform is responsible for collecting the trading data from various sources. The calculated return of the prices will be stored into InfluxDB. At the same time, it will also send the records through the messaging system. The messages are then received by the Spark App and perform real-time analysis. The Spark App also interacts with GCS to read the historical files into the memory. Synthetic Data Generation Utils is the implementation of Geometric Brownian Motion. The generating series are also stored in GCS. An actual use case and performance evaluation will be discussed in the next chapter.

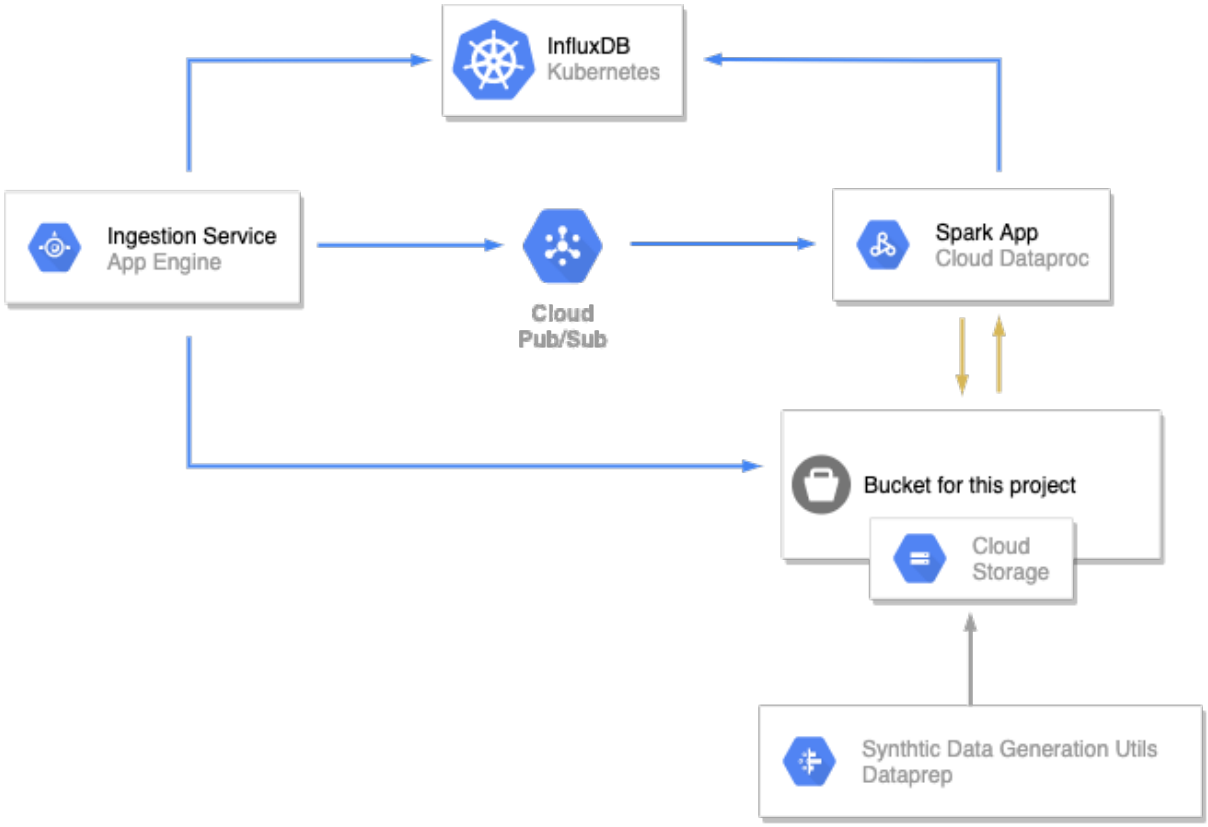

Figure 3.6: Integrated Architecture

4 Case Study: J.P. Morgan Stock Price Volatility

In this chapter, it will focus on the performance of the architecture on an actual use case. This chapter is divided into background introduction, workflow of the architecture, and the evaluation of the architecture in terms of the latency.

4.1 Background

J.P.Morgan Chase & Co is an American investment bank. It is the largest bank in the United States [18]. Its main business include investment bank and financial services, assets management. As one of the largest banks in the market, it represents the banking and financial services industry at some level. The stock prices of J.P.Morgan also reflects the confidence of the traders in the stock market. Therefore, this is a typical example for to be used for analysis of the volatility.

The J.P.Morgan historical stock prices dataset is collected from IEXCloud [16] for the past 1 year dated back to 1st April 2019. The collected dataset contains the following field:

- **date**, the timestamp of the data
- **high**, the highest price at the current timestamp
- **low**, the lowest price at the current timestamp
- **open**, the opening price at the current timestamp
- **close**, the closing price at the current timestamp
- **volume**, the trading volume at the current timestamp
- **average**, the average price at the current timestamp

To be more specific about each category of the price, in diagram 4.1 illustrates the opening price, closing price, the highest price, and the lowest price. When the opening price is greater

than the closing, it means the the return at the current timestamp is negative. In contrast, the return is positive. This is important to understand when calculate the return.

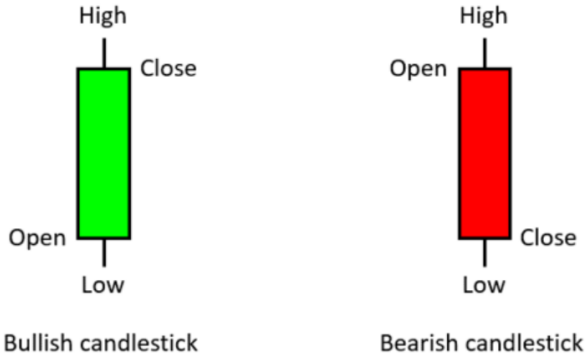

Figure 4.1: The Candlestick Diagram

4.2 Data Fusion

The project uses the historical stock prices from JP Morgan Chase as an example retrieved from IEXCloud. The table 4.1 gives an overview of the raw data collected from IEXCloud. The table is indexed by timestamp and parsed into DataFrame using Pandas [24]. The prices

	date	high	low	average	volume	open	close
0	2020-02-24-09:30	132.380691	132.326294	132.354324	0	132.349971	132.357324
1	2020-02-24-09:31	131.910000	131.000000	131.686000	7299	131.650000	131.825000
2	2020-02-24-09:32	132.260000	131.875000	131.990000	2242	131.920000	132.140000
3	2020-02-24-09:33	132.110000	131.775000	131.914000	1136	131.860000	131.990000
4	2020-02-24-09:34	131.950000	131.585000	131.663000	1900	131.950000	131.590000
...	...	...	...	...	...	...	...
84642	2019-04-01-15:55	104.590000	104.555000	104.566000	2524	104.570000	104.590000
84643	2019-04-01-15:56	104.670000	104.600000	104.630000	1776	104.600000	104.670000
84644	2019-04-01-15:57	104.670000	104.620000	104.636000	1681	104.670000	104.620000
84645	2019-04-01-15:58	104.665000	104.640000	104.657000	3403	104.640000	104.665000
84646	2019-04-01-15:59	104.660000	104.600000	104.633000	1718	104.650000	104.630000

Table 4.1: Historical Price Overview

that are usually used for analysis are opening and closing prices. A visualization for both opening series and closing series is created. The diagram 4.2 is the visualization of historical prices in past one year. The x-axis is the time and y-axis shows the open price and the close price. The diagram 4.2 is created by Plotly [27]. This Library allows to create interactive

visualizations in python. Recall that the stock price changing is a stochastic process. Therefore, the Geometric Brownian Motion model is used to generate the synthetic data Although the general trend of this stock prices is growing, it is not growing smoothly. It is

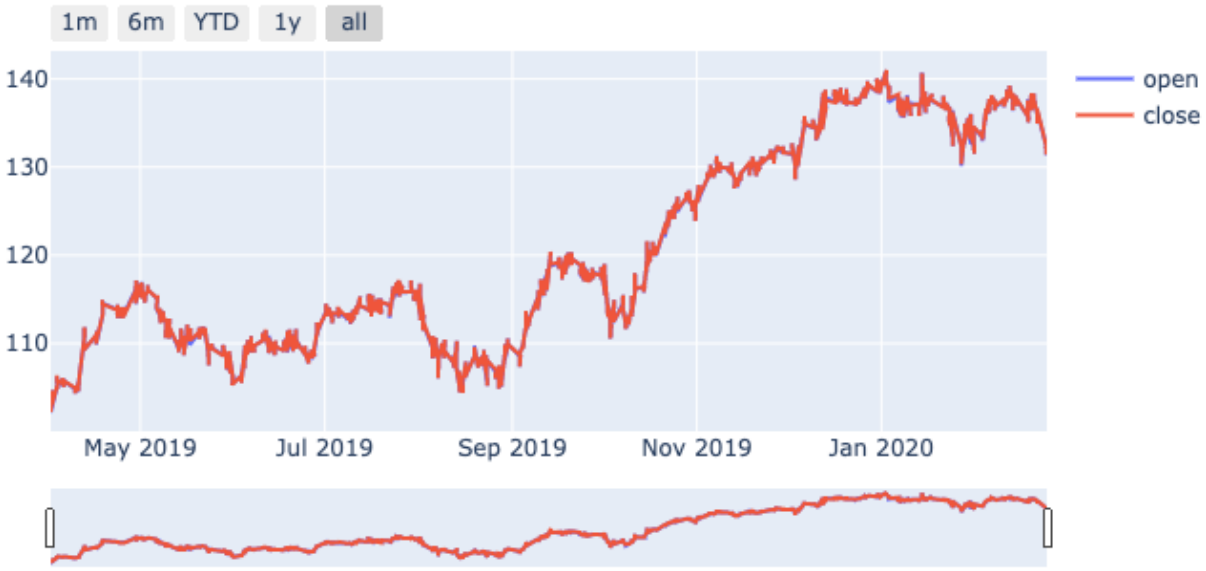

Figure 4.2: JPM historical

observable that the prices are fluctuating constantly. An initial analysis of the series indicates that the diffusion term could be relatively large in the entire dataset. To calculate the drift term and diffusion term, it must require the descriptive statistics especially mean and standard deviation.

The first step is to calculate the return of both opening series and closing series. Recall the function (1) for calculating the return, it uses the prices between 2 timestamps to calculate the return. The build in functions in pandas [24] has the ability to calculate the return as well as moving average.

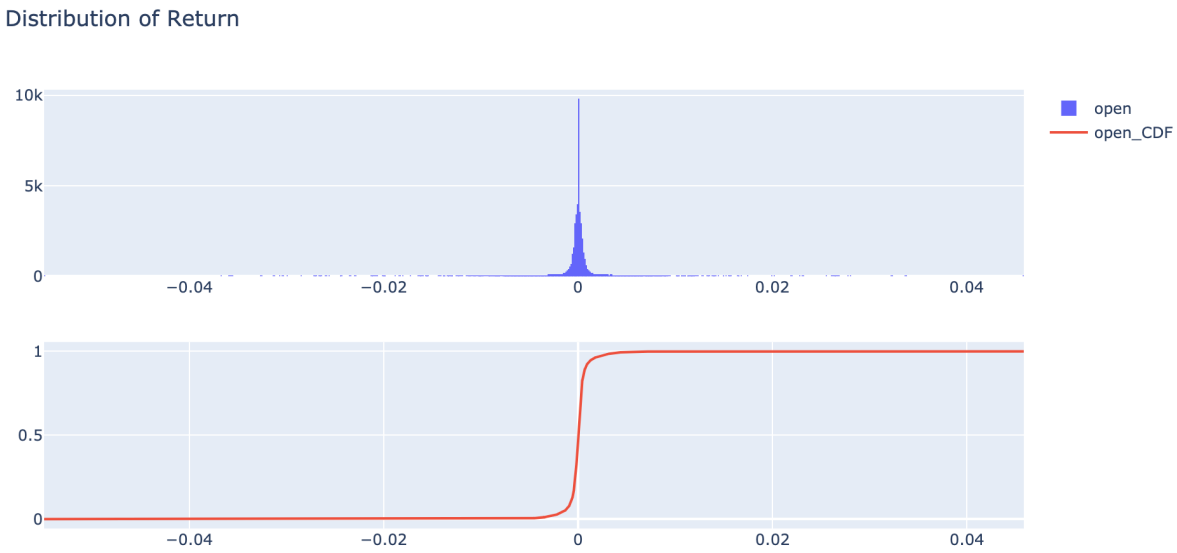

Figure 4.3: Distribution of mean

Using the mean and standard deviation, we can use equation (4) and equation (6) to get drift and diffusion. The distribution of the return of opening is showing in figure 4.3. The majority of the return values are distributed around 0. Therefore, the mean should be close to 0. Since the prices changed with large amplitude, the standard deviation should be much larger than mean.

The descriptive statistics are important because they will be used in Drift(4) and Diffusion (6). The table shows that the mean of the return on open price and close price are both -0.000002 whereas the standard deviation are 0.001304 and 0.001295 respectively. This means in a longer-term, there is likely to have a loss.

	open	close
count	84646.000000	84646.000000
mean	-0.000002	-0.000002
std	0.001304	0.001295
min	-0.054925	-0.055331
25%	-0.000268	-0.000273
50%	0.000000	0.000000
75%	0.000273	0.000275
max	0.045852	0.046643

Table 4.2: Descriptive Statistics of Opening and Closing Prices

We use the open price as an example and to show the impact of random shock on the stock price using Geometric Brownian Motion. We use function 9 to generate the 200 simulations over the next 500 trading minutes. For each of the simulations, the drifts and diffusions for the next 500 trading minutes were calculated.

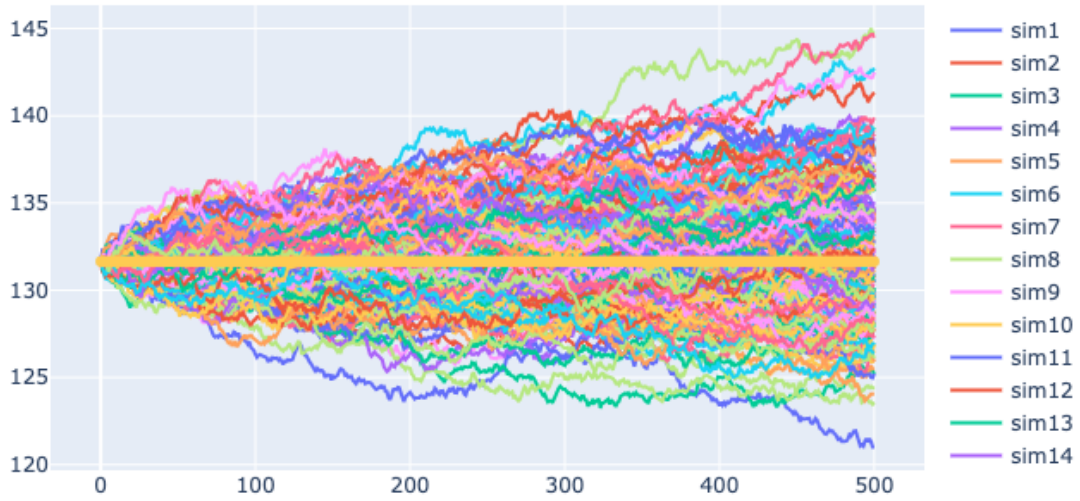

Figure 4.4: Geometric Brownian Motion simulations

The figure 4.3 shows the stock prices will not necessarily fall due to the effect of the random shock. It can have various behaviours due to random shock. Using this technique, we generate a large file containing the open price and the close price and the return for both. There is a total number of 20 million data points generated using this technique. Due to the size of the dataset is too large, pandas will slow down the speed in processing the data at this scale. Therefore, we use pySpark to process it. Again, the first step is to calculate the

summary	open_return	close_return
count	20000000	20000000
mean	-5.77346929528028E-7	-7.63066010053184E-7
stddev	0.001304045280573...	0.001295300598713...
min	-0.00756449414955...	-0.00723812346957...
max	0.007219574289302...	0.00660210623657714

Table 4.3: Descriptive Statistics of Large Dataset

descriptive statistics of the return. PySpark also provides build-in functions to achieve this. Using PySpark allows processing a large volume of data in a relatively short time. It is observable that the mean values slightly changed while the standard deviation remained the same in table 4.3. This is because of the effect of random shock. This file was then partitioned into small chunks and stored in Google Cloud Storage.

4.3 File Partition

The generated file size is 2.05GB. The file is partitioned and stored in Google Cloud Storage. The partition number depends very much on the number of cores. The default number of partitions is 200 by Spark. When loading the files, Spark will only keep the memory utilized. If the memory is not sufficient for the entire file, Spark will only read the partitions that within the allowance of the memory and spill the rest of the partitions on the disk. According to Spark documentation, the best practices for the number of partitions. If an RDD has too many partitions, the task scheduler will take more than the execution time to schedule the tasks. This will significantly increase the latency in actual real-time processing. If the number of partitions is to less, it will result in some of the cores in the idle state hence resulting in less concurrency. The number of partitions is initially set to 22 after the file is generated. When reading the file through the spark application, the file gets repartitioned. In local environments, the computer has 4 cores and files are partitioned into 8 chunks. When performing the calculation, it takes 2 minutes locally. However, the latency could be

reduced drastically using cluster on the cloud. The Dataproc was used in order to speed up the calculation with 3 nodes and each node has 4 cores. The file then was partitioned into 24 chunks. The execution time was much faster. The speed of computation can be determined by hardware as well as the number of partitions. The goal is to utilize the cores and maximize the parallelism in execution.

4.4 Data Flow

With the historical data stored in GCS, the new incoming data are ingested from Alpha Vintage. In this case study, we use the one minute stock price of J.P. Morgan Chase & Co. In the Ingestion Platform, the endpoint to download the file was used. When this endpoint gets queried, it will request the target file from Alpha Vintage. The function used is TIME_SERIES_INTRADAY with 1 minute interval and the symbol for J.P. Morgan Chase & Co. is JPM. When the file is downloaded, as explained in the previous chapter, the FileManager will start to read the content of the file and send them to Message Sender. The message sender will send the trading records in user defined frequency. The default value in this case was 1 minute. However, during testing, this rate was adjusted to 30 seconds to test the limit of the system. The results will be discussed later. The trading records are persist in Google PubSub and and waiting to be polled by the Spark App. There is a topic created for simulate the trading data flow in real time. This topic is also subscribed by the Spark App so that the Spark App has the ability to access the persistent messages.

The Spark App will continuously consume the messages and processing them. The incoming messages are deserialized into a Java Class. When the Spark App is running, it will load in the partitions into the memory.

In this project, the means and standard deviations for both open price and close at real time were calculated. These 2 statistics represents the volatility of the stock price. Volatility represents the historical fluctuations of the price at the past. The value of this property changes as new trading records coming in. The reason why volatility is important is because it is an indicator of the optimism or the fear in the market. This is also important for assessing the performance of the stock. As in has been introduced in the Geometric Brownian Motion, the fluctuation (the diffusion term) is directly related to standard deviation. The magnitude of the volatility depends on the standard deviation (variance) as well. With higher value of standard deviation, the large magnitude of fluctuation can appear on the stock price. Traders use volatility to measure the risk of the stock. If the volatility increase, it increases the chance of mis predicting.

Calculations for big data can be expensive. Especially the aggregation, most of the aggregation involves iterating whole data. The aggregation functions in Spark provides the

capability of calculating mean and standard deviation. There is no inclusion of shuffling when using aggregation alone. Recall the diagram 3.5, the RDDs of the historical data will be used together with incoming record to perform aggregation.

The calculated volatilities at real time are stored in InfluxDB. In influxDB, the two measures was created. The first measure is the return of the stock price and the second measure was created for tracking volatility at real time. The measurements of the volatility and returns will persist in the database. Since the instance is hosted on cloud, Google provide high availability for reading and writing data. The configurations for InfluxDB for the Ingestion Platform and the Spark App are different because of the different usage of the InfluxDB. In the Ingestion Platform, the Java client was created with hardcoded endpoint. In the Spark App, it uses the connector as the stream will be written directly to InfluxDB.

4.5 Result

The results are evaluated by examining the timestamp of 2 measurements and comparing the latency. The return traces are populated into the measurement in a fixed rate of 1 minute. The latency is measured by the time difference between the return and the statistical moments. For example, the first return record arrived to the database at 25th March 2020, 17:45:20 and the associated statistical moments arrived at 25th March 2020, 17:45:54. Therefore, there is a 34 seconds latency. The return of which calculated between the current trading record and the previous data arrived at the database before the new records coming in. The average time of statistical moments is computed take up to 40 seconds. This has a negative effect on decision making when the data are arriving in 1-minute frequency. The screenshot 4.5 of the measurement in the InfluxDB instance hosted on the Compute Engine. In the screenshot 4.5, the return is computed using the equation 1.

time_series_return

time	closeReturn	openReturn	symbol	volume
2020-03-25T17:45:20.16Z	"0.005158437730287457"	"0.0025747915644924735"	"JPM"	"63659"
2020-03-25T17:46:21.187Z	"6.204243702692125E-4"	"0.00725980541947191"	"JPM"	"86025"
2020-03-25T17:47:22.601Z	"0.0046105919003115545"	"0.003973181028060657"	"JPM"	"273881"
2020-03-25T17:48:23.496Z	"0.0029853548629366067"	"0.0"	"JPM"	"73084"
2020-03-25T17:49:24.502Z	"9.916027564045304E-4"	"6.900012545476564E-4"	"JPM"	"83663"
2020-03-25T17:50:24.945Z	"0.0051780752715333556"	"0.005390192111975334"	"JPM"	"133929"
2020-03-25T17:51:25.467Z	"0.005657324840764422"	"0.003319802105797298"	"JPM"	"288166"
2020-03-25T17:52:26.607Z	"-0.010334582095336442"	"0.006670087224217469"	"JPM"	"134738"
2020-03-25T17:53:29.101Z	"0.004312450602489748"	"-6.239603324891263E-4"	"JPM"	"105946"
2020-03-25T17:54:30.805Z	"-0.011644466613499015"	"0.005214152700186103"	"JPM"	"87440"
2020-03-25T17:55:31.428Z	"-8.65730423496025E-4"	"-0.003584737110770897"	"JPM"	"116825"

Figure 4.5: The Return of The Time Series

The timestamp uses the system time instead of the original trading time. The reason for not

using the original trading time is for the purpose of latency investigation. In the flow introduced in Figure 3.5, the return was computed using reduce function. This step involves shuffling. However, when the size of the RDD is 2 under the normal circumstance. Therefore, it is not time-consuming. The MapReduce flow in the SparkStreaming produces the results and stored into the InfluxDB.

The return RDD is then used to compute the statistical moments. As it produces a *Dstream* of the return RDDs, without the need of collecting the RDDs in the stream, it can then be used for computing the mean and standard deviation together with the historical dataset. The computation of mean and standard deviation could be slow as it will aggregate the 20000000 rows of data. The union is a very efficient operation in Spark. It will only combine 2 partitions without moving any other partitions around. Therefore, the new incoming data unions with the historical dataset and then the aggregation functions are applied. In figure

JPM_HISTORICAL

time	avgCloseReturn	avgOpenReturn	stdCloseReturn	stdOpenReturn	tag	volatilityClose	volatilityOpen
2020-03-25T17:45:24.457Z	-7.835853285830102e-7	-4.685048019188985e-7	0.0012971786321215445	0.0013038394360777598	"return"	-0.0006040689456173441	-0.00035932706816129567
2020-03-25T17:45:54.402Z	-7.835844579345799e-7	-4.6850428135916427e-7	0.0012971779114676124	0.0013038387117232147	"return"	-0.0006040686100243885	-0.00035932686853573084
2020-03-25T17:46:57.921Z	-7.835835872880843e-7	-4.685037608005869e-7	0.0012971771908148813	0.0013038379873698768	"return"	-0.0006040682744319921	-0.0003593266689104988
2020-03-25T17:47:58.28Z	-7.835818460008975e-7	-4.685027196869024e-7	0.0012971757495130224	0.001303836538666823	"return"	-0.0006040676032488774	-0.00035932626966103274
2020-03-25T17:48:57.468Z	-7.835801047214497e-7	-4.68501678577845e-7	0.001297174308215968	0.0013038350899685983	"return"	-0.0006040669320679997	-0.00035932587041289743
2020-03-25T17:49:41.989Z	-7.835783634497408e-7	-4.685006374734148e-7	0.0012971728669237178	0.0013038336412752023	"return"	-0.0006040662608893594	-0.00035932547116609303
2020-03-25T17:50:13.414Z	-7.835766221857709e-7	-4.6849959637361163e-7	0.0012971714256362718	0.0013038321925866352	"return"	-0.0006040655897129563	-0.00035932507192061944
2020-03-25T17:50:49.152Z	-7.835757515566879e-7	-4.684990758254452e-7	0.0012971707049943503	0.0013038314682441625	"return"	-0.0006040652541255938	-0.0003593248722983817
2020-03-25T17:51:28.736Z	-7.835748809295396e-7	-4.684985552784355e-7	0.00129716998435363	0.0013038307439028971	"return"	-0.0006040649185387905	-0.0003593246726764766
2020-03-25T17:52:06.558Z	-7.835731396810473e-7	-4.684975141878864e-7	0.0012971685430757925	0.0013038292952239877	"return"	-0.0006040642473668619	-0.0003593242734336646

Figure 4.6: Statistical Moments

screenshot 4.6, the measurements for volatility is computed and stored into the database. Note the first record is not the result, the statistical moments of all historical datasets are calculated on application start. Therefore, when estimating the latency, this record will not be considered. In average, the trace gets populated on average 40 seconds late than the arriving of the original data. This means the aggregation time for data takes 40 seconds for both opening prices and closing prices. There are 2 statistical properties and for two types pf prices calculated in the total. Therefore, to compute each of them, it takes about 10 seconds in worst case.

The results suggested the system can tolerant with the trading frequency at 1 trade per minute. In the algorithm trading environment, the trading frequency can be much higher. Therefore, a more robust system would be required to measure risk in real-time. At the

moment, the response time for the system can satisfy the requirement of trading at 1-minute frequency. The parallelism of Spark Engine plays a critical role. This is directly related to the partition which has been mentioned in previous sections.

4.6 Evaluation

4.6.1 System setup

The system contains a Google PubSub instance, an influxDB instance, a Spring boot web app for ingesting the data from the source, a Spark application for processing the streaming data, and a Google Cloud Storage for files storage. The Dataproc cluster provided by Google running Hadoop to perform Spark jobs execution. To measure the real-time risk for a specific stock price, the first step is to collect enough amount of historical dataset. In this case study, because there was no enough amount of historical dataset available, the Geometric Brownian Motion technique was used to generate the synthetic data. This technique uses the mean and standard deviation from sample space to ensure the generate data complies with the statistical properties of sample data. The Hadoop cluster was setup containing 3 nodes. Each node has 4 cores and 500 GB of disk size. The configuration of this cluster can satisfy all requirements of such a system. Dataproc will allocate the resources dynamically. The file partitioner was also configured manually to make the best use of the environments. This is the only place that involves system tuning. For different cluster configurations, the number of partitions is different. For this specific case, 24 was tested to have the best performance.

Due to the limited budget on Google Cloud, the most economical configuration for the cluster was chosen as such. Dynamically generating synthetic data and file partition is not available for this project. This is considered as further improvements to be made.

4.6.2 Performance

The performance of the system can be evaluated from the computation speed and stability. These 2 metrics are the key performance indexes of the system as well as the key objectives.

The speed of computation is evaluated by computing the delay of the statistical moments. In the screenshots of the databases, it is observed that the average delay is 30 seconds for means and standard deviations for both opening price return and closing price return. The system can support trading in 1 minute. The designed big data processing systems that allow quick retrieval of large items of historical data and allow for high-speed computation. The performance can be further improved with stronger hardwares and more efficient

approaches of computation resources allocation. In a study, [37] investigated the Spark-GPU acceleration when running Spark SQL queries. It suggested that for SQL queries, it can be up to 4.83 times faster.

The most critical component to evaluate the stability of the system is the messaging system. The proposed components in this architecture are connected by messaging. Messaging is also an approach to simulate real-time trading. When using PubSub in this project, the messages are guaranteed to be delivered. Unlike Kafka, PubSub requires manually ACK on the consumer side. This means any failure message will be persisted and observable until they are consumed. Apache Bahir is developed as a connector particularly for Google PubSub and Spark Streaming. It makes use of the manually ACK feature of Google PubSub to prevent data loss during the message travel.

4.7 Discussion

4.7.1 Data Selection

The historical dataset for J.P.Morgan Chase & Co. only available till 1st of April 2019. The file size is not large enough to satisfy the volume in Big Data. Therefore, we had to use Geometric Brownian Motion to generate synthetic data. The key finding in the dataset generated by GBM is that the mean of return is not stable. Since we applied random shock on the drift, the mean return is determined by the random variable. Thus, each simulation can have different results. However, the standard deviation remains the same as the magnitude of change is bounded by the original standard deviation. Therefore, the generated data can only represent part of the original universe. Based on this fact, the results computed by the system can only be referenced based on the generated data universe. If there were more historical datasets available, the result could be more representative.

4.7.2 Technologies reviews

In the literature review section, the main focus was on MapReduce and its open-source implementations, particularly in Spark and Hadoop. However other open source technologies were also considered. Recall one of the goals of the project is to produce data stream and control the speed, the streaming tool was considered as the first options. The major technologies include Apache Spark, Apache Flink, and Apache Storm. The benchmark results conducted for the above three technologies. The study [13] showed that SparkStreaming process the new events in a stepwise manner while the other tow is linear. The causes for this is because of the micro-batching design in SparkStreaming [31]. Although the study concludes that Apache Flink and Apache Storm are more successful in near real-time processing with lower latency, it is also worth to mention Spark can process

data with higher throughput. However, under the low throughput rate such as under 50000 per seconds, there is no difference among Storm, Flink, and SparkStreaming. The original design of this project was to have the capability of handling the data from multiple sources at the same time. Therefore, it was expected to have a higher throughput rate. The second reason why SparkStreaming was chosen as the streaming technology is that it is as a part of the Spark ecosystem that can integrate well enough with other Spark suites such as Spark SQL.

The Hadoop MapReduce was also mentioned as another MapReduce paradigm. However, it is impossible to use Hadoop standalone to achieve the object of processing real-time events. Another consideration for not using Hadoop MapReduce is that it was required to perform an aggregation over a large volume of Historical datasets frequently. Since in-memory processing is not supported by Hadoop MapReduce, an extra latency could have been introduced if not using Spark. The usage of SparkStreaming for this project also proved that SparkStreaming has the capability of processing the data in near real-time in this case study. However, the experiment of handling messages from multiple sources have not been conducted. This is considered as an improvement to be made in future.

In terms of the simulation component, Kafka was initially used for messaging system set up on the local machine. As it was introduced, the log files will be stored on disk in different partitions. However, when having a large volume of incoming messages, the disk overhead gets large. This is a challenge to storage ability to the local machine. However, when deploying to Google Cloud, the Kafka cluster requires extra VM instances. Due to the above considerations, Google PubSub was replaced for native cloud environments with equally well performance.

4.7.3 Challenges

This project focused on the velocity and volume aspects of big data. With the help of cloud services, the large volume of data can be stored easily. The velocity is measured by the latency of computation. During the implementations of the architecture, there are some challenges faced and worth mentioning. The first challenge was encountered during the deployment of the Ingestion Platform. The Google App Engine uses internal production server while Spring boot uses tomcat as an embedded server. Normally, when having the Spring boot running on the server or local machine, it uses Tomcat as the default server. It will handle the incoming requests from the clients. However, App Engine is a fully managed platform and allows to scale the application with a minimum amount of configuration. There was the compatibility issue with the App Engine and Tomcat server. The workaround was to remove the Tomcat dependency from Spring boot and use the default servlet configuration. Then the application was successfully deployed on App Engine.

Another challenge faced was running the parameters for Spark Engine and file partition. To achieve the maximum parallelism, it uses all worker nodes available. There were some experiments conducted to test the speed of processing regarding the number of partitions. It concluded that around 22 to 24 partitions the Spark Engine has the maximum processing speed. A problem for applying to reduce function for calculating the return is that when the messages are failed or delivered lately, the return is calculated incorrectly. This is highly depending on the stability of the messaging system. There is no feasible solution for these challenges and this requires future work.

4.7.4 Future Work

The first and the most important work is to have a disaster recovery mechanism for messaging module to ensure the correctness of the return. The messaging module acts like a middleware that connects the Ingestion Platform and Spark App together. As discussed previously, if the message module is down or even a message is delivered late, the computation result is different. Therefore, the message module has to be stable and continuously deliver the message in user-defined frequency.

Another future work is for testing the performance of multi-source processing. The design of the system has put this requirement into consideration. Therefore, SparkStreaming was chosen to handle the situation of a high throughput rate. Due to time constraint, there is no actual performance testing in terms of this aspect.

Finally, the automation process of generating synthetic data is another important future work. The purpose of generating synthetic data is the volume of the original data is not large enough to satisfy the requirement of Big Data. However, if for specific stock prices, a large volume of historical datasets is available, the step of generating synthetic data is no longer required.

4.7.5 The 3Vs: Velocity, Volume, Variety

The Velocity refers to the latency. In this project, the Decision Latency is measured by the time difference between the arrival of return and the computed mean return and standard deviation return. The latency is brought down to 10 seconds to aggregate over a single column. This is because the configuration that utilize the worker nodes. The time spent to calculate the statistical moments is made up of several components. The first one is aggregating the historical dataset. The aggregation function requires to iterate all rows. If the number of rows is defined as N, the time complexity is O(4N) as it will calculate mean and standard deviation for both opening and closing prices. The second cause of latency is that when the application starts, it will also take O(N) time to load the historical datasets. The union operation can be treated as constant time complexity. Therefore, the total time

complexity is O(4N) and can be written as O(N) for computing the statistical moment every time.

With the help of cloud services, Big Data storage is not an issue anymore. Volume can easily be resolved by adding more hardwares. Google Cloud Storage is built on top of Google File System which is easy to scale.

Variety is the aspect that did not get chance to experiment in this case study. The idea of variety is to investigate if different stock prices have influence on each other. Due to the time constrain, this experiment was not conducted. However, as mentioned above in 4.7.4, the design of the system has put this into the consideration. SparkStreaming is able receive the messages from multiple sources. The initial idea was to set the context to each stock price when it is ingested by Kafka to allow SparkStreaming to process different stock prices in one spark job. This will potentially affect the speed of calculation.

5 Conclusion

This paper has contributed to reviewing the suitable technologies to build low latency decision system. The reviews the latency problem, particularly in High-Frequency Trading industry, suggested that higher latency can result in decreasing in market quality. This project aims to simulate a trading environment and perform near-real-time analysis hence to reduce the latency as much as possible.

This project has tested under the trading frequency of 1 trade per minute, the decision time can take up to 40 seconds based on 20 million rows of the historical dataset. The users for such systems is targeted to analysts in the financial service industry. This architecture involves Big Data technologies and aims to solve the challenges of Big Data in terms of Velocity and Volume. This project answers the questions the response of retrieving large data in a relatively smaller amount of time. The system also achieves the average computation time of a single statistical moment of in 10 seconds. The frequency challenges the system from the computation speed aspect. This trading frequency is lower-bounded by the computation time. The current system is capable of analyzing the risk of trading in near real-time and help to decide before the next transaction is made.

In the current implementation, the system can potentially fail if the messaging module is congested. The default configuration of the window size is 1.5 minute in SparkStreaming to allow 2 RDDs to exist at the same time. This can allow calculating the return using the current price and the previous price. For example, if the messaging module failed to deliver one message, the RDDs in the defined window length will be incorrect and hence the return is calculated incorrectly.One possible fix can use a load balancer and monitor the healthiness of the messaging module. When it is discovered offline, replace with backup servers.

In future, it may worth investigating the possible approaches to decrease the latency from the optimization of Logical Execution Plan in Spark MapReduce, dynamically resource allocation algorithm on the cloud environments (particularly in file partition), and finally, the Spark Engine parameters tuning.

6 Appendix

This chapter contains the some of the code snippets and pictures in some of key components in this architecture.

6.1 A1: Calculate return of stock price

```
def get_return_stats(df, range_of_time='default'):
  df_copy = pd.DataFrame()
  for col in df.columns:
      if df[col].dtype != 'datetime64[ns]':
          df_copy[col] = df[col].pct_change(num)
          df_copy[col+'_movingAvg'] = get_moving_average(df_copy[col],num)
  return df_copy
```

6.2 A2: Geometric Brownian Motion

```
def brownian_motion(df,base_price,num_simulations, size):
    #volitility measured by std of return
    # df is return
    mean = df.mean()
    variance = df.var()
    volitility = df.std()
    res = {}
    t = np.linspace(0,1,int(size)+1)

    for i in range(1,num_simulations):
        temp = [k for k in range(size+1)]
        temp[0] = base_price
        noise = standard_brownian_motion(size)
        for j in range(1, int(size+1)):
            drift = (mean - 0.5 *variance)*noise
```

```
            diffusion = volitility*noise[j-1]
            #print('drift:' + str(drift))
            #print('diffusion: '+str(diffusion))
            temp[j] = base_price*np.exp(drift + diffusion)
        res.update({i:temp})
    return res
def standard_brownian_motion(size):
    np.random.seed()
    dt = 1/size+1
    W = np.random.standard_normal(int(size+1))
    W = np.cumsum(W)*np.sqrt(dt)
    return W
```

6.3 A3: Calculate return in PySpark

```
df_open_lag =
    df.withColumn('prev_open',func.lag(df['open']).over(Window.orderBy('index')))
df = df_open_lag.withColumn('open_return', (df_open_lag['open'] -
    df_open_lag['prev_open']) / df_open_lag['open'])

df_close_lag =
    df.withColumn('prev_close',func.lag(df['close']).over(Window.orderBy('index')))
df = df_close_lag.withColumn('close_return', (df_close_lag['close'] -
    df_close_lag['prev_close']) / df_close_lag['close'])
```

6.4 A4: Calculate moving average in PySpark

```
df=df.withColumn('moving_average_open',func.avg('open_return').over(Window.orderBy('index')
df=df.withColumn('cum_sum_open',func.sum('open_return').over(Window.orderBy('index').rangeB
df = df.withColumn('moving_average_open',df['cum_sum_open']/df['index'])


df=df.withColumn('moving_average_close',func.avg('close_return').over(Window.orderBy('index
df=df.withColumn('cum_sum_close',func.sum('close_return').over(Window.orderBy('index').rang
df = df.withColumn('moving_average_close',df['cum_sum_close']/df['index'])
```

6.5 A5: Request historical 1 minute trading data from IEXCloud

```
def get_file(num_days, symbol):
    URL = 'https://cloud.iexapis.com/v1/stock/' + symbol + '/intraday-prices'
    df = pd.DataFrame()
    PARAMS = {'token': 'secret_token'}
    columns_to_drop = ['marketAverage', 'marketNumberOfTrades', 'minute',
        'label', 'notional', 'numberOfTrades',
                    'marketHigh', 'marketLow', 'marketVolume',
                        'marketNotional', 'marketOpen', 'marketClose',
                    'changeOverTime', 'marketChangeOverTime']

    for i in range(1, num_days):
        logging.info('getting record ' + str(i) + 'remaining: ' + str(num_days
            - i))
        day = datetime.date.today() - datetime.timedelta(days=i)
        PARAMS.update({'exactDate': day.strftime("%Y%m%d")})
        r = requests.get(url=URL, params=PARAMS)
        data = r.json()
        try:
            data[0]['date']
        except:
            continue
        df_temp = pd.DataFrame.from_dict(r.json())
        df_temp['date'] = df_temp['date'].str.cat(df_temp['minute'], sep='-')

        df_temp.drop(columns_to_drop, inplace=True, axis=1)
        df_temp.fillna(df_temp.mean(), inplace=True)
        df = df.append(df_temp)
    return df
```

6.6 A6: Available endpoints in the Ingestion Platform

Api Documentation

Api Documentation

Apache 2.0

file-manager : File Manager

Show/Hide | List Operations | Expand Operations

GET	/v1/file/{symbol}	getFileInformation
GET	/v1/file/{symbol}/{date}	getFileInformation

ingestion-manager : Ingestion Manager

Show/Hide | List Operations | Expand Operations

GET	/v1/retrievdData/{symbol}/{function}/{timeInterval}/download	retrieveDataAsFile
GET	/v1/retrieveData/{symbol}/{function}/{timeInterval}	retrieveData

[BASE URL: / , API VERSION: 1.0]

Bibliography

[1] URL: https://docs.spring.io/spring/docs/4.3.12.RELEASE/spring-framework-reference/html/aop.html.

[2] URL: https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html.

[3] S. Agarwal and B. R. Prasad. "High speed streaming data analysis of web generated log streams". In: *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*. 2015, pp. 413–418.

[4] W Farida Agustini, Ika Restu Affianti, and Endah RM Putri. "Stock price prediction using geometric Brownian motion". In: *Journal of Physics: Conference Series* 974 (2018), p. 012047. DOI: 10.1088/1742-6596/974/1/012047. URL: https://doi.org/10.1088%2F1742-6596%2F974%2F1%2F012047.

[5] Fernando Almeida. "Big Data: Concept, Potentialities and Vulnerabilities". In: *Emerging Science Journal* 2.1‘ (). ISSN: 2610-9182. DOI: https://doi.org/10.28991/esj-2018-01123.

[6] *Apache Hadoop*. URL: https://hadoop.apache.org/.

[7] *Apache Kafka*. URL: https://kafka.apache.org/.

[8] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Association for Computing Machinery, 2015, 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[9] Matthew Baron et al. "Risk and Return in High-Frequency Trading". In: *Journal of Financial and Quantitative Analysis* 54.3 (2019), 993–1024. DOI: 10.1017/S0022109018001096.

[10] Vinayak R. Borkar, Michael J. Carey, and Chen Li. "Big Data Platforms: What's Next?" In: *XRDS* 19.1 (Sept. 2012), 44–49. ISSN: 1528-4972. DOI: 10.1145/2331042.2331057. URL: https://doi-org.elib.tcd.ie/10.1145/2331042.2331057.

[11] PhD Burns William E. "Efficient-market hypothesis (EMH)." In: *Salem Press Encyclopedia* (2020). URL: http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=ers&AN=113931145.

[12] S. Chickerur, A. Goudar, and A. Kinnerkar. "Comparison of Relational Database with Document-Oriented Database (MongoDB) for Big Data Applications". In: *2015 8th International Conference on Advanced Software Engineering Its Applications (ASEA)*. 2015, pp. 41–47.

[13] S. Chintapalli et al. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1789–1792.

[14] V. Gadepally and J. Kepner. "Big data dimensional analysis". In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014, pp. 1–6.

[15] Joel Hasbrouck and Gideon Saar. "Low-latency trading". In: *Journal of Financial Markets* 16.4 (2013). High-Frequency Trading, pp. 646 –679. ISSN: 1386-4181. DOI: https://doi.org/10.1016/j.finmar.2013.05.003. URL: http://www.sciencedirect.com/science/article/pii/S1386418113000165.

[16] *IEX Cloud: Financial Data Infrastructure*. URL: https://iexcloud.io/.

[17] *InfluxDB 1.8 documentation*. URL: https://docs.influxdata.com/influxdb/v1.8/.

[18] *JPM.N - JPMorgan Chase & Co. Profile*. URL: https://www.reuters.com/companies/JPM.N.

[19] Kyong-Ha Lee et al. "Parallel Data Processing with MapReduce: A Survey". In: *SIGMOD Rec.* 40.4 (Jan. 2012), 11–20. ISSN: 0163-5808. DOI: 10.1145/2094114.2094118. URL: https://doi.org/10.1145/2094114.2094118.

[20] Eliezer Levy and Abraham Silberschatz. "Distributed File Systems: Concepts and Examples". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), 321–374. ISSN: 0360-0300. DOI: 10.1145/98163.98169. URL: https://doi-org.elib.tcd.ie/10.1145/98163.98169.

[21] A. I. Maarala et al. "Low latency analytics for streaming traffic data with Apache Spark". In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, pp. 2855–2858.

[22] *Migrating Apache Spark Jobs to Dataproc | Migrating Hadoop to GCP*. URL: https://cloud.google.com/solutions/migration/hadoop/migrating-apache-spark-jobs-to-cloud-dataproc.

[23] *NoSQL Databases Explained*. URL: https://www.mongodb.com/nosql-explained.

[24] *pandas*. URL: https://pandas.pydata.org/.

[25] J. Patel. "An Effective and Scalable Data Modeling for Enterprise Big Data Platform". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 2691–2697.

[26] Andrew Pavlo et al. "A Comparison of Approaches to Large-Scale Data Analysis". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Association for Computing Machinery, 2009, 165–178. ISBN: 9781605585512. DOI: 10.1145/1559845.1559865. URL: https://doi.org/10.1145/1559845.1559865.

[27] *Plotly Python Graphing Library*. URL: https://plotly.com/python/.

[28] Pygmalios. *pygmalios/reactiveinflux*. URL: https://github.com/pygmalios/reactiveinflux.

[29] *RDD Programming Guide*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds.

[30] S. Sagiroglu and D. Sinanc. "Big data: A review". In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. 2013, pp. 42–47.

[31] *Spark Streaming: Apache Spark*. URL: https://spark.apache.org/streaming/.

[32] Xinhui Tian et al. "Latency critical big data computing in finance". In: *The Journal of Finance and Data Science* 1.1 (2015), pp. 33 –41. ISSN: 2405-9188. DOI: https://doi.org/10.1016/j.jfds.2015.07.002. URL: http://www.sciencedirect.com/science/article/pii/S2405918815000045.

[33] Madhavi Vaidya and Shrinivas Deshpande. "Critical Study of Performance Parameters on Distributed File Systems Using MapReduce". In: *Procedia Computer Science* 78 (2016). 1st International Conference on Information Security & Privacy 2015, pp. 224 –232. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.02.037. URL: http://www.sciencedirect.com/science/article/pii/S1877050916000399.

[34] Alpha Vantage. *ALPHA VANTAGE*. URL: https://www.alphavantage.co/.

[35] A. Verma, A. H. Mansuri, and N. Jain. "Big data management processing with Hadoop MapReduce and spark technology: A comparison". In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. 2016, pp. 1–4.

[36] Hugh J. Watson. "Tutorial: Big Data Analytics: Concepts, Technologies, and Applications". In: *Communications of the Association for Information Systems* 34 (2014). DOI: 10.17705/1cais.03465.

[37] Y. Yuan et al. "Spark-GPU: An accelerated in-memory data processing engine on clusters". In: *2016 IEEE International Conference on Big Data (Big Data)*. 2016, pp. 273–283.