



UNIVERSITY OF NAIROBI

COLLEGE OF BIOLOGICAL AND PHYSICAL SCIENCES

SCHOOL OF COMPUTING AND INFORMATICS

**An Evaluation of Real-Time Processing of Call Detail Records Using
Stream Processing**

CATHERINE KITHUSI WAMBUA

P53/73389/2014

A research project report submitted to the School of Computing and Informatics in partial fulfillment of the requirements for the award of the Degree of Masters of Science in Distributed Computing Technology at the University of Nairobi

December 2017.

DECLARATION

I certify that this research project report to the best of my knowledge, is my original authorial work except as acknowledged therein and has not been submitted for any other degree or professional qualification award in this or any other University.

Signature: _____ Date: _____

Catherine Kithusi Wambua (P53/73389/2014)

This research report has been submitted in partial fulfillment of the requirements for the Degree of Master of Science in Distributed Computing Technology at the University of Nairobi with my approval as the University supervisor.

Signature: _____ Date: _____

Dr. Christopher Chepken

DEDICATION

To my beloved parents, for their unrelenting dedication to ensuring that my siblings and I acquired the best education despite all odds.

To my dear sister and brothers, for being the best cheering squad anyone could ever ask for.

To my friends and colleagues, for their advice and support and for affording me time to pursue this degree

ACKNOWLEDGEMENTS

I would like to sincerely thank the almighty God who has been a source of strength, wisdom and perseverance during this project, this research project could not have been done without his ever present sustenance.

My deepest gratitude to my research project advisor and supervisor, Dr. Christopher Chepken for his invaluable advice and motivation. His guidance, feedback and expert opinions were instrumental towards the completion of this project.

I would also like to thank the members of the research panel for their constructive criticisms, input and corrections which aided in the completion of this research project.

ABSTRACT

A common problem plaguing the telecommunication industry is how to process the gigantic amounts of Call Detail Records (CDR) data it generates. Currently, telecommunication companies use batch processing systems to process CDR data at intervals ranging from 5 minutes to 24 hours, and even then, not all data is processed. Present batch processing platforms are vendor based, requiring proprietary software, specialized hardware and licenses. Because of this, processing of CDR data is expensive and has prevented telecommunication companies from gaining all the benefits that could be acquired by the effective and total processing of CDR data. With the strides made in big data recently and especially in stream processing, total processing of CDR data is made possible, furthermore, stream processing facilitates the real-time processing of data.

This research primarily focuses on stream processing of CDR data, this would be of benefit to telecommunication companies seeking to gain complex, intricate and speedy insights into their customers and networks. This research also involves a feature comparison of several stream processing platforms in use today for the purposes of selecting a single suitable platform for this project. The selected platform is then evaluated in terms of performance and resource usage, all in an effort to determine whether the selected stream processing platform is suitable for the real-time processing of CDR data.

TABLE OF CONTENTS

DECLARATION	i
DEDICATION	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	viii
ABBREVIATIONS	ix
CHAPTER ONE	1
INTRODUCTION	1
1.0. Background Information	1
1.0.1. SMSC CDR Processing	3
1.0.2. Batch Processing vs Big Data Processing	4
1.0.3. Implementing Big Data Projects	6
1.1. Problem Statement	7
1.2. Significance of the Research	8
1.3. Research Objectives	8
1.4. Research Questions	9
1.5. Research Assumptions	9
CHAPTER TWO	10
LITERATURE REVIEW	10
2.0. Introduction	10
2.1. Stream Processing	12
2.2. Selecting a Stream Processing Platform	14
2.3. Apache Spark Streaming	19
2.3.1. Apache Spark Streaming Architecture	21

2.4.	Apache Hadoop Yarn.....	23
2.5.	Apache Hadoop HDFS	24
2.6.	Apache Cassandra	25
2.7.	Apache Kafka.....	27
2.8.	Apache Zookeeper	29
2.9.	Graphite and Grafana.....	30
2.10.	Related Work and Research Gap.....	30
CHAPTER THREE		33
METHODOLOGY		33
3.0.	Introduction.....	33
3.1.	System Infrastructure Setup	34
3.2.	Prototype Development.....	37
3.2.1.	Software Development Model.....	37
3.2.1.1.	Requirements Definition and Analysis	37
3.2.1.2.	Prototype Design and Development	38
3.2.1.3.	System Testing.....	45
3.2.1.4.	Software Implementation	46
3.3.	Experimentation.....	47
3.3.1.	Fixed Parameters	47
3.3.2.	Variable Parameters	47
3.3.3.	Evaluation Metrics	48
3.3.4.	Experiment Execution	49
3.3.5.	Result Data Collection and Analysis.....	50
CHAPTER FOUR		53
RESULTS AND DISCUSSIONS.....		53
4.0.	Results	53
4.0.1.	Streaming Platforms Feature Comparison Results.....	53
4.0.2.	System Infrastructure Installation Results.....	54
4.0.3.	Prototype Development Results.....	54
4.0.4.	Experiment Results	54
4.0.4.1.	Performance Metrics Results	55

4.0.4.2. Resource Usage Metrics	57
4.1. Discussions	58
CHAPTER FIVE	65
CONCLUSION AND RECOMMENDATIONS	65
5.0. Conclusion	65
5.1. Challenges	66
5.2. Recommendations for Future Work	67
REFERENCES	68
APPENDIX 1: CLASS DIAGRAM	72
APPENDIX 2: FEATURE COMPARISON.....	73
APPENDIX 3: SOURCE CODE.....	78

LIST OF FIGURES

Figure 1: Sample CDR (StackExchange, 2014)..... 1

Figure 2: Yarn Based architecture for Apache Spark (Stackoverflow, 2016)..... 23

Figure 3: Apache Cassandra Write Process (Ma, 2014)..... 27

Figure 4: Kafka Ecosystem 29

Figure 5: Pseudocode for Apache Spark Streaming prototype 39

Figure 6: Flow of data between the various installed components 42

Figure 7: Logical model for the CDRLOGS5 table..... 44

Figure 8: Executing the Stream processing prototype 46

Figure 9: Performance metrics per batch interval..... 55

Figure 10: Performance metrics for batch interval setting of 3 seconds 56

Figure 11: Correlation between batching and record processing 57

Figure 12: Resource usage metrics vs Batch interval 57

Figure 13: CPU utilization vs Throughput per Second 58

Figure 14: DAG Visualization..... 61

Figure 15: Apache Spark Streaming Storage..... 63

LIST OF TABLES

Table 1: Infrastructure setup 35

Table 2: Evaluation metrics..... 49

ABBREVIATIONS

CDR – Call detail records

SMSC – Short Message Service Center

CSP – Communication Service Provider

GiB – Gigabytes

MiB – Megabytes

KiB - Kilobytes

CPU – Central Processing Unit

QoS – Quality of Service

CQL – Cassandra Query Language

MSISDN – Mobile Station International Subscriber Directory Number

OSS – Operating Support Systems

CHAPTER ONE

INTRODUCTION

1.0. Background Information

A Call Detail Record (CDR) is a formatted collection of information about a chargeable event for use in billing and accounting. A chargeable event is any activity utilizing telecommunications network resources and related services for which a network operator wants to charge for, be it voice services, data services, messaging services or other custom services. Network resources also generate other kinds of detail records that contain data on all the transactions happening in the network.

A major problem facing telecommunication companies today is the effective and reliable processing of CDR data to gain speedy and valuable insights into their customers and network.

```
MSIDN:IMSI:IMEI:PLAN:CALL_TYPE:CORRESP_TYPE:CORRESP_ISDN:DURATION:TIME:DATE
068373748102;208100167682477;351905149071;PLAN1;MOC;CUST1;0612287077;247;12:07:12;01/01/2012
068373748102;208100167682477;351905149071;PLAN1;MTC;CUST2;0600000001;300;12:15:09;01/01/2012
068373748102;208100167682477;351905149071;PLAN1;SMS-MO;CUST1;0613637193;0;12:18:18;01/01/2012
068373748102;208100167682477;351905149071;PLAN1;SMS-MT;CUST1;0612899062;0;12:21:07;01/01/2012
065978198280;208100310191699;356008289837;PLAN3;MOC;CUST1;0612283725;90;12:00:00;01/01/2012
065978198280;208100310191699;356008289837;PLAN3;MOC;CUST1;0613069656;82;12:02:27;01/01/2012
065978198280;208100310191699;356008289837;PLAN3;MOC;CUST1;0613481951;78;12:04:41;01/01/2012
```

Figure 1: Sample CDR (StackExchange, 2014)

Telecommunication companies on average generate terabytes of data every day from their network, a medium sized telecommunication company having twenty million subscribers is capable of generating 20 terabytes of CDR data per day (Kx, 2016). Processing such amounts of data is a challenge for a lot of telecommunication companies. This is credited mostly to the costly nature of processing voluminous data, at the very least, it requires the acquisition of powerful infrastructure, in terms of servers and storages as well as proprietary technologies that attract pricey license, support and maintenance costs. This has imposed the constraint of processing CDR data in batches. Data is gathered into batches of fifteen, hourly or even 24 hour intervals. Furthermore, not all the data is processed, mostly, only the data containing billing information is paid attention to. Batch processing jobs take a 'batch of data' as input and produce a 'batch of

results' as output. Depending on the size of the data, batch processing jobs can take hours to complete (Guller, 2015).

Other than billing information, CSPs within their networks also collect other valuable data such as control plane and user plane data. Control plane data includes channel setup and control information and is instrumental in diagnosing and troubleshooting customer related problems such as dropped calls, call setup success rates. User plane data contains payload information that is transmitted after the necessary channels have been setup, for example, user plane traffic for data services shows how many bytes of data have been downloaded or uploaded and what internet sites have been visited. CSPs also gather a lot of data from the Operation Support Systems (OSS) domain. OSS includes data such as fault management data (alarms generated by the numerous network elements deployed in a network), performance management data (performance data of network elements generated periodically), configuration management data (how the network elements have been configured) and security management data (who has access to the networks and user activities performed on the network elements).

Unfortunately, telecommunication companies also face an additional problem of the existence of data silos in their networks, instead of aggregating the data generated into a central place or collating data from the different data sources to create insightful reports that are global and useful to the entire company, data is kept at its source or point of generation and is overseen by the team owning the source network elements. Reports generated are thus disjointed and can only be meaningful to a select few, mostly the owners of the network elements who are only interested in problems within their domains.

To provide superior service to their customers, telecommunication companies need to process data faster and not only billing data but all the data generated within their networks, they need to be able to respond to their customers' queries quickly and effectively, they need to monitor their network's performance throughout and identify issues before they become major problems, they need to be able to identify fraudulent activities within their network promptly; there are so many ways in which telecommunication companies can utilize CDRs to improve their service and network quality while increasing their profit margins, and these cannot withstand

processing jobs that take hours to complete, they require jobs that complete in milliseconds, they require real-time processing.

With real-time processing, we expect processing times that take no longer than a second to process voluminous data. A major difference between real-time processing and batch processing is the fact that real-time processing does not require the storage of data into persistent storage before processing, data is handled on the fly and stored after processing is complete.

1.0.1. SMSC CDR Processing

A vital network element in a telecommunications company's network is the SMSC (Short Message Service Center). An SMSC is responsible for handling SMS (Short Message Service) operations within a network, this includes routing, storing and forwarding SMSs to their destination. An SMS can be categorized as 'MO (Message Originating)' or '(MT) Message Terminating' depending on its source. MO SMSs are sent by a sender from their mobile phone, MT SMSs are received by a recipient on their mobile phone. Each SMS, whether MO or MT generates a CDR on the SMSC and will contain significant details such as the status of the SMS and error code information which are crucial when handling customer complaints.

Five years ago in 2012, we were part of a team that set out to process SMSC CDRs in a local telecommunication company. The project required processing of SMSC CDRs stored in several files and storing the results in an oracle database. This would assist customer service agents at the call center query the status (delivered, expired, deleted, waiting or error) of an SMS, in case of a customer complaint, and assist accordingly. The data was also useful in troubleshooting SMS billing complaints.

Each CDR file was 2.5 GB in size and each day an average of 250 GB of CDR data was generated. Processing was done on a nightly basis using PERL (Practical Extraction and Report Language) scripts and Oracle data loader and Oracle 9 database. First, CDR data would be transferred from several network elements and stored on the processing server via SFTP (Secure File transfer protocol), then a PERL script would locate the file on the filesystem, open and read it, formatting it and selecting the columns of interest, while at the same time loading the data into an oracle database using oracle data loader. The process would be repeated for all the other files until all

files had been processed. The entire procedure of reading, extracting and loading of all the data would take an average of six to eight hours every night.

Processing was done on a single SUN V890 server, with 64 GB of memory, eight 1.2 GHz processors and 2 TB of external storage, a powerful machine by any standards then.

The decision to process data nightly was arrived at due to the long durations and resource intensive nature of the data loading and database write transactions that would slow down the database. We therefore could not afford to process the data during the day as customer care agents would be affected. During the day, the oracle database exclusively handled database read requests only, allowing customer service agents to query the database without experiencing the performance degradation brought about by the write operations. Furthermore, a large percentage of customer issues are received during the day.

The long processing durations were not the only limitations with this processing method, because of the nightly processing schedule, the latest data customer service agents could query was the previous day's data. This affected resolution times for customers. A customer complaining of an issue that had taken place that day would have to wait longer, for second line engineers to provide the required info. In addition, customer service agents complained of queries that took too long to return results, with some queries taking more than five minutes. This is in spite of all the database tuning operations that had been done to improve performance. This resulted in dissatisfaction in the system and eventually it was retired to be replaced by a vendor sourced system capable of processing voluminous data. Vendor sourced systems as established earlier, require specialized hardware and proprietary software that are inflexible and costly.

1.0.2. Batch Processing vs Big Data Processing

The SMSC CDR processing described in the section above fits the batch processing model, where jobs can be scheduled to be done during off peak hours and take hours to complete (Guller, 2015). Batch processing jobs also require that data is first congregated and stored in a persistent storage before processing, this also introduces a degree of latency in the processing.

Since then, major advancements have been made in how voluminous data is handled, the term Big Data became common place and systems and technologies to handle Big Data developed, one of the first platforms being Apache Hadoop which comprised of the HDFS distributed file

system and Map reduce, the compute engine developed at Yahoo. This revolutionized the Big Data scene, with impressive benchmarks being published for Apache Hadoop, in 2013, Apache Hadoop was able to sort 102.5 TB in 4,328 seconds on 2100 nodes of 64 GB memory, two 2.3Ghz processors and 36 TB storage (Graves, 2013). This encouraged other platforms to be developed to improve on this performance, such as Apache Flink, Apache Samza, Apache Storm Core, Apache Storm Trident and Apache Spark, which are all open source and can be setup on commodity hardware.

One of the technologies that came along as part of the Big Data evolution is Big Data stream processing. Unlike batch processing where data has to be persisted to storage prior to processing, stream processing processes data in streams, as it arrives into the platform and stores it after processing ensuring efficient utilization of server resources as well as better performance. A requirement of a stream processing platform is that it is able to process voluminous data at high throughputs while maintaining low latencies. Stream processing platforms are therefore suited for the real-time processing of Big Data. Examples of stream processing platforms in existence today are Apache Flink, Apache Storm trident, Apache Spark Streaming, Apache Kafka Streams and Apache Samza.

Stream processing platforms can be further categorized into two;

- Native stream processing platforms

Native stream processing platforms process incoming data immediately it hits the platform, one record after the other, resulting in the lowest-possible latency (Zapletal, 2016). Examples of native stream processing platforms are Apache Storm, Apache Samza and Apache Flink.

- Micro-Batching stream processing platforms

On the other hand, Micro-Batching stream processing platforms create short batches from the incoming data which are then processed. These batches are created according to a pre-defined time constant, typically seconds (Zapletal, 2016). Examples of such platforms are Apache Storm Trident and Apache Spark Streaming.

CDR data can be classified as Big Data as they exhibit the three properties of Big Data

- Volume – A medium sized communication service provider (CSP) having twenty million subscribers is capable of generating terabytes of CDR (Call Detail Record) data per day (Kx, 2016)
- Variety – CDRs are generated in different formats depending on the network element type and vendor of the network element. These can be CSV, xml, json, ascii or even unstructured data.
- Velocity – CDRs are continuously and frequently generated throughout the day at varying intervals

1.0.3. Implementing Big Data Projects

One would be mistaken to think that telecommunication companies would be among the first industries to jump on the Big Data bandwagon, seeing as how they have perennially dealt with voluminous data long before the likes of Google and Facebook. According to a research project conducted in 2014 by McKinsey, only 30% of telecommunication companies had established Big Data projects in their companies (Bughin, 2016). The reasons for this low state of adoption ranged from lack of experience and talent pool, lack of quality processes to lack of permission to use data. The fact that only 5% of the telecommunication companies that had adopted Big Data, reported its positive impact on profit is not encouraging. However, the same report does indicate that 45% of telecommunication companies are considering implementing Big Data projects in the near future.

How does one then go about starting and implementing a successful Big Data project? A question that every company should have when thinking about implementing and adopting Big Data solutions is how to do it effectively to realize a lasting impact on their critical success factors. A report done by IBM in 2013 (IBM, Analytics: Real-world use of big data in telecommunications, 2013) states that the most successful Big Data solutions identify the business requirement first and then tailor the infrastructure, data sources and analytics to support the business requirement. Examples of business requirements for CSPs are improving customer experience, driving new products, increasing productivity and optimizing networks. They also implement in

phases, whereby integration – of both data and infrastructure – is carried out step by step rather than performing a big bang deployment. Measurements and metrics are also very important as they aid in monitoring and tracking the success of the implementation, correcting any deviations instantly to stay on course.

The IBM report also indicates that most telecommunication companies value real-time information processing more as some of their business requirements require real-time information for them to be valuable and effective. We identified the business requirement for this project as improving customer experience, the CDR data processed would be used by customer care agents to resolve customer complaints relating to messaging services using real-time information.

1.1. Problem Statement

Batch processing of voluminous data such as CDRs is ineffective. Processing of large amounts of data takes hours to complete and due to the resource intensive nature of batch processing jobs, requires jobs to be scheduled during off peak hours. Batch processing does not also account for the other two characteristics of Big Data, velocity and variety. Since CDR data is categorized as Big Data, batch processing is ineffective in the processing of CDR data. In addition, telecommunication companies are interested in obtaining real-time information from CDR data, implementing a real-time batch processing system would be very expensive due to the costs that would be required to acquire specialized infrastructure, not to mention the licensing and maintenance expenses.

The aim for this project was to evaluate whether real-time processing of CDRs is achievable using a suitable stream processing. This was done by implementing a CDR stream processing solution on virtual machines and utilizing open source stream processing platforms. We first had to perform a feature comparison of three stream processing platforms using eight well-chosen features in order to select a stream processing platform for this project. Using the selected stream processing platform, installed system infrastructure and a stream processing prototype, SMSC CDR data was processed while collecting the performance metrics of throughput and latency and the resource usage metrics of CPU utilization, disk utilization and memory utilization.

These metrics were chosen as they are the most commonly used while benchmarking stream processing systems. High throughputs while maintaining low latencies are desired for real-time systems (Kim & Blafford, 2016). The resource usage metrics were chosen as they facilitate the monitoring of how effectively the selected streaming platform uses resources allocated to it. They also aided in the tuning of resources for optimal system performance. From the experiment results obtained, we were then able to gauge whether the selected stream processing platform is a proper candidate for the real-time processing of CDR data.

1.2. Significance of the Research

The telecommunication industry will benefit from this research, as it demonstrates that real-time processing of CDR data is feasible and in a less costly way. This research also demonstrates how to design, setup, install and optimize a Big Data stream processing cluster for optimal performance. It also demonstrates the performance limits achievable for the size of our cluster and therefore would be useful in system dimensioning.

With real-time processing of CDRs, telecommunication companies will be able to use CDR data in more innovative ways that bring much needed value and insights on the network and customers.

1.3. Research Objectives

The overall objective is “To evaluate real-time processing of CDRs using a suitable stream processing platform”

Specific objectives are as follows

- To select a suitable streaming platform for the real-time processing of CDRs
- To design, setup and optimize the selected platform for the real-time processing of CDRs
- To develop a streaming prototype that processes CDRs
- To measure the performance metrics of latency and throughput and resource usage metrics of CPU utilization, disk utilization and memory utilization on the installed system
- To determine that real-time processing of CDRs via the selected stream processing platform is feasible

1.4. Research Questions

1. Is real-time processing of CDRs using stream processing feasible?
2. What are the ideal parameters for carrying out a feature comparison in order to select a suitable stream processing platform?
3. What is the correlation between the performance metrics of latency and throughput and the resource usage metrics of CPU utilization, disk utilization and memory utilization?

1.5. Research Assumptions

The chief assumption for this project was that CDR data would be used by call center agents to resolve customer complaints related to SMS by querying a database using the complainants MSISDN. This would allow a call center agent to retrieve all SMS history relating to that particular MSISDN for a specific period and analyze status and error codes.

CHAPTER TWO

LITERATURE REVIEW

2.0. Introduction

According to the 3GPP specification 32.297, release 13 a CDR also known as a Call Detail Record is a formatted collection of information about one or more chargeable events for use in billing and accounting. More than one CDR may be generated for a single chargeable event if the event takes a long amount of time to complete or more than one party is to be charged for the event. A chargeable event is any activity utilizing telecommunications network resources and related services for which a network operator wants to charge for. Examples of chargeable events include short messages, calls and data sessions. Events that are normally not charged to the subscribers such as signaling events, roaming events, interconnect events, call transfers etc. are also recorded in a CDR (3GPP, 2016). In summary, every event that takes place in a telecommunication network is documented in a file known as a CDR file which contains details of the parties involved (e.g. user details like MSISDN, IMSI, IMEI), event id, network elements involved, event time stamps, event durations, user locations, error codes generated if any, QoS details if any among others. These details will vary depending on the available fields or columns in a CDR file since they are configurable, however, there are fields that must be included in each CDR file in accordance well-known standards such as the 3GPP specification. Optional fields can vary between network element vendors, CSPs, network element types and release versions. The size of a CDR file depends on configurable parameters such as number of CDRs a file should contain, duration the CDR file should stay open and maximum size of CDR file.

Even though CDRs are continuously being generated in a network, a significant amount of data is generated during the 'busy hour'. The term 'busy hour' denotes a duration during the day when the network is at its busiest and usually lasting for an hour, during this time, CDRs are being generated faster than usual resulting into several CDR files or huge CDR files depending on the CDR file configuration. Busy hour differs from country to country or CSP to CSP depending on their service offerings and occasions. For example, 'busy hour' for voice services for a CSP could be from 1700hrs to 1800hrs, while 'busy hour' for data services for the same CSP could be from 2000hrs to 2100hrs. If a CSP is running a popular service offering, then the amount of CDRs

generated will also increase. During school holidays, SMS traffic increases, attributed to school going children on holiday with access to mobile phones. More CDRs equals more money, in fact, it can be said that the currency of the telecommunication industry is the CDR (Middendorf, 1999). It is common to see CSPs running marketing campaigns to increase network usage during off peak hours in an effort to increase revenue as well as increase network utilization. Network elements are quite expensive to acquire and therefore money is lost any time a network element is lying idle.

Processing of CDR data for purposes other than for billing and rating takes a backseat in most telecommunication companies. One of the major reasons for this is the fact that the processing and storage of huge amounts of data is costly. Processing of large amounts of data necessitates the acquisition of powerful and expensive infrastructure in terms of software, servers and storages as well as proprietary technologies that attract pricey license and support fees. It is therefore not uncommon for network operators to process and analyze limited sets of CDR data, processed at fifteen or hourly or even daily intervals, using batch processing where data is aggregated, stored then processed in batches. It is also not uncommon to encounter a vast majority of data being discarded after the mandated storage period is over. Batch processing is suited for the processing of voluminous historical data and jobs can be scheduled to run after hours when the resource intensive batch processing would not interfere with other jobs or operations. Depending on the size of the data, batch processing jobs can take minutes or even hours to complete (Guller, 2015).

CDRs contain valuable information that can be used by a telecommunication company to gain competitive advantage within the industry as well as outside the industry. A few telecommunication companies in America are using CDRs to create new revenue streams. For instance, Sprint sells some of its data to marketing agencies (IBM, 2013) and this is due to the fact that they have embraced Big Data and implemented Big Data solutions. They can be used effectively for real-time fraud detection, dynamic network optimization, customer experience monitoring and handling, dynamic radio network planning, personalized marketing campaigns and so much more. They can also be used to solve some of the societal challenges facing us today, during the 'Data for Development Senegal' challenge, researchers used anonymized CDR data for

societal development, with emphasis on agriculture, health, transportation and urban planning, energy and national statistics (d4D, 2014).

There is a huge drive today for CSPs to be more customer centric, to be able to understand their customers better and resolve any issues they might have in a timely manner and also provide inventive service offerings that convert them into loyal customers. Using Big Data technologies to process CDR data makes this a reality. This is due to the fact that Big Data processing platforms have been proven to be much faster and better at handling voluminous, varietal and high velocity data than batch processing while running on commodity hardware and open source software. As part of the advancements in Big Data technology, real-time processing of Big Data became possible, where voluminous data is processed within seconds rather than hours. Platforms capable of real-time processing of data are classified under stream processing. According to (IBM, 2013), 40% of CSPs want Big Data to enable them perform real-time processing, this would mean ingesting and analyzing contextual data in real-time.

2.1. Stream Processing

Stream processing as defined by (SQLSTREAM, 2017), is the real-time processing of data continuously, concurrently and in a record by record basis. Stream processing does not treat data as a file but rather as an infinite stream of data or stream of records. Stream processing also does not require that data is stored first before processing, streams of records are processed immediately upon entering the system. Stream processing is designed to process data on the fly i.e. data in motion by utilizing continuously running queries.

But what do we mean by 'real-time'? The word real-time has different meanings depending on the context in which it is used. According to (Rouse, 2011), real-time refers to a system responsiveness that is perceived by a user to be immediate or nearly immediate. The Oxford dictionary defines real-time in computing as the processing of data within milliseconds so that it is available immediately as feedback to the originating process. According to (Barlow, 2013), real-time denotes the ability to process data as it arrives, rather than storing it for later retrieval for processing. It is the processing of data in the present, rather than in the future. Real time processing requires a continual input, constant processing, and steady output of data (Syncsort, 2015). This research project measured two performance metrics – latency and throughput – with

a latency target of processing data in less than 60 seconds. We set performance targets for throughput and latency that we anticipated would allow for effective processing of large volumes of CDRs without affecting a systems stability. The latency targets are thus based on (Rouse, 2011) definition of real-time.

A stream processing solution running on a single machine is not capable of handling high volume, high velocity data (Guller, 2015). Stream processing platforms are therefore deployed as clusters of several similarly configured servers or machines. This enables them to handle large volumes of data arriving at high frequencies as the workload is distributed among the nodes in the cluster. The availability of multiple CPU units or CPU cores in a cluster also increases concurrency, where a job is split into tasks that are executed in parallel on the different CPU cores. In addition cluster deployment facilitates high availability, data processing does not halt due to the failure of a single machine, and the workload is redistributed amongst the active nodes. This ensures system fault tolerance. Of course if there is multiple server failure such that the cluster does not have the required quorum to continue operations, the whole system is affected.

Stream processing is required when computations have to be initiated and done fast and continuously (Wahner, 2014), which is exactly what this project intended to achieve by applying stream processing to the real-time processing of CDR data.

Stream processing platforms can be classified in to two classes; native streaming platforms and micro-batching platforms. In the native streaming platforms, events are processed one by one as they arrive in to the platform in a continuous manner. They are also referred as the true streaming platforms. In micro-batching, events are grouped into mini batches of data then processed as a stream i.e. stream of batches the result of which is a mini-batch of results. Such platforms combine batching and streaming in order to obtain the best of both stream processing and batch processing worlds, batch processing is known to handle large sets of data well (volume), while streaming is renowned for handling fast incoming data (velocity) (Wang Y. , 2016). This is usually at the cost of latency as native streaming platforms have lower latencies (in milliseconds) than micro-batching systems (in seconds).

Some of the popular streaming platforms in use today include Apache Storm, Apache Storm Trident, Apache Flink, Apache Spark Streaming, Apache Kafka Streams and Apache Samza which

are hosted by the Apache Software Foundation (Mayo, 2016). There are also commercial based platforms such as SQLstream Blaze, IBM Infosphere streams, TIBCO's event analytics, Oracle stream analytics, Informatica among others. Nevertheless, we are not keen on evaluating proprietary platforms as they require use of specialized tools, licenses and cost. Moreover, based on previous benchmarks, there is no much difference in performance between the commercial and open source systems, with some open source platforms having better performance.

To choose an appropriate streaming platform for this research project, eight feature characteristics were used to perform a platform comparison. There is currently no industry standard on the features one has to use, with different research papers using differing features. These eight features were chosen as they encompass the desirable features that a critical stream processing system should have for stability and performance.

2.2. Selecting a Stream Processing Platform

The first step after deciding on the streaming platforms to compare is to determine the distinguishing features to use to perform the comparison. Different researchers have used different features in their comparisons. (Bockermann, 2014), used the metrics of scalability and distribution, usability and process modelling, execution semantics and high availability, message processing semantics(guarantees), state handling and fault tolerance to compare three platforms. These three platforms were selected based on message delivery guarantee and platform adoption. The three platforms all guarantee exactly-once message processing, this means that there is no loss or duplication of data records. This is important for processing CDRs as duplicate data can lead to incorrect reports, billing information, wrong aggregations among other problems. In addition, loss of any CDR data is non-negotiable. The three platforms also have high adoption rates by companies, Apache Storm boasts of more than 80 companies (Storm, 2017) using Apache Storm (this value is for both Apache Storm and Apache Storm Trident), Apache Flink is being used by 32 companies (Flink, 2017) while Apache Spark has implementations in more than 90 companies (Spark, Project and Product names using "Spark", 2017) (this figure is for all the libraries running on top of Apache Spark).

(Zapletal, 2016), used the metrics of streaming model, API, guarantees, fault tolerance, state management, latency, throughput and maturity to perform comparisons of Apache Storm trident, Apache Spark Streaming and Apache Flink.

The list of features to use in the comparisons depends a lot on the use case at hand, for this research, we settled on eight metrics as discussed in the following sections. The metrics of latency and throughput were not included in this comparison as they are among the metrics that were being measuring for this research. They are also classified as performance metrics, performance of a platform is expected to differ based on several factors which include the use case at hand, message size, infrastructure cluster size and infrastructure setup and optimization.

i. Fault Tolerance and High Availability

Fault tolerance describes the ability of a platform to withstand failure, to continue operating during failure and to recover to a previous stable state with minimal data loss or corruption. The recovery time should also be minimal. When it comes to CDR processing, this is a desirable feature as it ensures that minimal or no CDR data is lost or corrupted, lost CDR data will lead to misleading reports. Long system recovery times could lead to CDR data buffering at the data sources, and depending on their storage capacity, this could become problematic. It also means that when the system is finally operational, there will be a backlog of data to be processed which could destabilize the system, rendering real-time processing of CDRs ineffective.

Of interest, is to understand what fault tolerance mechanisms are in place for each platform and the actions involved in recovering a failed system. How do the various components of a cluster recover from failure?

Fault tolerance has a direct correlation to high availability. A system that encompasses high availability is capable of tolerating failures better. So, how do the three platforms being compared implement high availability? How is their architecture instrumental in providing high availability? Does this involve both software – tasks and processes and hardware? Are there any single points of failure within a cluster? What is the effect when a single point of failure element goes down and are there any practical mitigations?

ii. Scalability

Scalability is the ability to handle increasing workload by adding system components such as RAM, disk, CPU (vertical scaling) and/or adding more nodes to the cluster (horizontal scaling). The interesting questions to ask under scalability include;

Does scaling the cluster lead to better performance? Does the platform scale linearly? Does the addition of components and nodes require any downtime? How is data stored and accessed by the nodes comprising a cluster? What is the size of the largest cluster of each of the platforms? This is useful in denoting the infrastructure limits of each platform. Scalability is instrumental to telecommunication companies, as the trend currently is to diversify products and enter into new industries in an effort to increase profit margins. Technology is also growing very fast, everyone nowadays owns a computer or more in their hands, smartphone use is ubiquitous and these communicate with telecommunication companies networks all the time. This has led to a linear growth in CDR data. CDR data also increases during special occasions such as school holidays and calendar holidays, successful promotions will also lead to an increase in CDR data. With scalability therefore, it should be easy to add power with minimal downtime to an existing cluster to meet processing demands as well as be able to scale down when CDR data reduces thus conserving system resources.

iii. Message Delivery Guarantees

How many times does a message need to be delivered to the platform for it be successfully processed?

- Exactly once whereby a message is delivered only once, the message can neither be lost or duplicated
- At least once whereby multiple attempts are made to deliver a message
- At most once whereby a message can be delivered zero or one times, so messages may be lost (Zapletal, 2016)

What are the inherent mechanisms in each platform that ensures exactly once message processing? Are they configured by default or do they have to be manually activated? Does the platform guarantee exactly once message processing even after failures?

With respect to CDR processing, exactly once message processing is desired, this is because loss of CDR data or duplicated CDR data will lead to distorted reports and analysis.

iv. Adoption Level and Project Maturity

How many companies are actively using the platform to implement Big Data solutions? Platforms with more users are desirable as it shows trust in the platform and that the platform is stable and offers beneficial features for Big Data processing. How many active committers does the platform have? Committers are people chosen by each platform to perform tasks such as code development, maintain the code, bug reports, software patching for purposes of fixing bugs and documentations. A platform with a huge committer base is able to move faster, it is able to roll out required features and patches; it means it has a sufficient talent pool for maintaining the platform now and in the future. This is important as one would use a platform whose longevity is assured. How many updates are rolled out by the platform in a year? When was the last software release? We need to be convinced that the platform is actively being developed to add more features and maintained and that bugs are fixed promptly. Big Data is rapidly evolving and each platform has to ensure that it is keeping up.

v. Streaming Model

How does each platform achieve streaming? Is the platform a natively streaming platform or does it use micro-batching? What data structures are used to store and process data? What is the mechanism behind the streaming model employed? Does the platform support iterative processing? Iterative processing is useful in processing repetitive jobs that needs data to be stored in memory. With iterative processing, data does not need to be fetched and loaded into memory every time it is needed but can be cached to be used later. Fetching of data especially from persistent storage such as the filesystem is

expensive in terms of resources and slow. Iterative processing leads to faster processing times.

This is useful in CDR processing, especially when we consider fetching large chunks of data from the filesystem to be loaded into memory, and CDR data is voluminous. Also of note is that if data does not fit in memory, then it will be spilled to disk. CDR processing can benefit from stream processing combined with other form of processing such as graph processing or machine learning and iterative processing would be advantageous in such scenarios.

vi. *Multiple Library Support*

Does the platform include other libraries or does it perform stream processing only? For this research, only streaming platforms are required, however, there is an abundance of use cases for CDRs, and we are already foreseeing use cases that require SQL, Machine learning and Graph processing. The ideal platform should include these capabilities

vii. *Programming Language Support*

It is important for a platform to support multiple programming languages, this is so as not to lock out users who are not well versed in the particular language supported by the platform. For this research project, our programming language proficiency dictated that we source for a platform that supports Java, with the limited research project timelines, it is not wise to start learning a new programming language.

viii. *State Management*

State management deals with how the platform maintains state as well as the internal mechanisms for state management. State is required to execute complex execution logic, by keeping track of operations that have completed, those in progress and those that are yet to start. Some platforms maintain and monitor state themselves, without allowing external access to operation state, while other platforms provide interfaces for managing state, which is an advantage. Stream processing can be divided into stateless and stateful stream processing. In stateless stream processing, applications receive streams of records and generate a result based on the information of the last record alone (Celebi, 2016), this is enough for simple stream applications involving simple transformations. Complex

event processing requires stateful stream processing which keeps the state of the variables and operations taking place in an application. State management is critical for exactly once message processing.

The results of this feature comparison are as presented in Appendix 2, from which Apache Spark Streaming was selected as the project's platform of choice.

While we had locked in the processing platform to use, the platform that would act as the data source and generate streams of record and a platform for storing the processed data had yet to be identified.

Apache Kafka was selected to act as the data source, Apache Kafka is a fault tolerant, scalable publish-subscribe messaging system that enables the development of distributed applications. Apache Kafka has an impressive adoption rate, in America, it is in production in nine out of the top ten telecommunication companies (Asay, 2017). It is compatible with Apache Spark Streaming and is used to generate streams of records that can be ingested by Apache Spark Streaming reliably.

Apache Cassandra is a scalable, column family, NoSQL database. In previous benchmarks against similar NoSQL databases such as Couchbase, HBase and Redis it outperformed them in terms of throughput. (Rabl, Sadoghi, & Jacobsen, 2012). Apache Cassandra was chosen as the database platform for storing data persistently.

In the next sections, we shall go through the platforms that were used for this project.

2.3. Apache Spark Streaming

Apache Spark Streaming is a library on top of Apache Spark that deals with stream processing. It is a micro-batch stream processing platform that divides incoming stream of records into batches of records according to a configurable parameter known as the *batch interval*. The batch interval denotes the frequency at which batches are generated and is specified in terms of milliseconds or seconds, supporting durations as low as 500 milliseconds.

The core abstraction in Apache Spark Streaming is the discretized stream also known as a *dstream*. A *dstream* is a continuous sequence of RDDs representing a continuous sequence of data (Spark, Spark Streaming Programming Guide, 2017). A Resilient distributed dataset (RDD) is an immutable collection of elements that can be operated on in parallel and is the core

abstraction in Apache Spark (Spark, 2017). Since RDDs are immutable, it then follows that dstreams are also immutable. Apache Spark provides operations that transform dstreams from one format to another, these are known as transformations. Transformations provide a way of applying computing logic, mostly in functions, to data in dstream form to generate a required result in dstream form. There is an array of transformations available that include map, flatmap, filter, reduceByKey etc. (Spark, Spark Streaming Programming Guide, 2017). There are two types of transformations

1. Narrow transformations

These transformations only act on data that resides on a single partition on the source RDD. These include map and filter.

2. Wide transformations

These transformation act on data that may reside on multiple partitions of the source RDD. During execution, Apache Spark Streaming may have to fetch data from several partitions spread across the cluster, this is known as RDD shuffling. Examples of wide transformations are reduceByKey, groupByKey, repartition and join

This project primarily used the map transformation which returns a new dstream by passing each element in the source dstream through a function (Spark, Spark Streaming Programming Guide, 2017). Other than transformations, there exists actions, actions trigger the actual execution of the dstream operations; examples are print and foreachRDD. Execution of the code in a streaming application will not start until an action has being called. This is known as lazy evaluation, transformations in Apache Spark are lazy by nature. When a transformation is applied on a dstream, it is not executed immediately, instead Apache Spark Streaming keeps a record of all the transformation that have been applied to a dstream in a direct acyclic graph (dataflair, 2017).

A direct acyclic graph (DAG), consists of a dstream and the transformations to be applied to that dstream. On execution of an action such as print, the DAG is submitted to the DAG scheduler for computation. During stream processing, one is able to visualize the DAG from Apache Spark Streaming User Interface, detailing the transformations that have been applied.

Apache Spark Streaming provides the Apache Spark Streaming User Interface for purposes of monitoring job execution and job performance. The user interface provides other details such as batch processing time, number of records processed, scheduling delay, job information, task information and other information.

2.3.1. Apache Spark Streaming Architecture

Apache Spark Streaming follows the master/slave architecture consisting of two major node types, two major daemons and a cluster manager (Dezyre, Apache Spark Architecture Explained in Detail, 2017).

2.3.1.1. Master and Worker Nodes

The two major node types in an Apache Spark Streaming cluster are

1. The master node(s)

The master node is responsible for coordinating the various workers and their executor processes in standalone mode. In the presence of a cluster manager, the master node requests for resources from the cluster manager and assign them to the executor processes running in the worker nodes. This project's cluster implementation consisted of a master node and a secondary master node, only one was active at any moment. The driver program was also run in the master node.

2. The worker nodes

The worker nodes are responsible for the actual execution. They contain the executor processes, which perform a series of tasks concurrently and return results to the master or driver program. One worker node can contain several executor processes, this depends on the resources available, it is however advisable to have one executor process per worker node. Executors are launched once at the beginning of a spark application and run for the entire lifetime of that application. Executors also provide in-memory storage for RDDs that are cached by the application (dataflair, 2017).

2.3.1.2. Driver and Executor Processes

The driver process is launched when the main method of the application is called. The main method of the application is called when an application is submitted to Apache Spark Streaming using the spark-submit command. In local mode, the driver process executes on the same node

that submitted the application. In cluster mode, where Apache Spark Streaming is executed on a cluster sharing the workload, the driver process will run on any node in the cluster. The driver is responsible for running the SparkStreamingContext, similar to an application session. The driver is responsible for several activities including

- Converting the application into a direct acyclic graph to be executed by the executors as a series of tasks
- Storing the metadata of RDDs and their partitions
- Exposing the information about the running Apache Spark Streaming application on the Apache Spark Streaming UI.
- In the presence of a cluster manager, requesting for resources required to launch the executors

There is only one driver per application program.

A running Apache Spark Streaming cluster consists of multiple executor processes, running on the worker nodes. The executors are launched by the master node or a cluster manager and run throughout the lifetime of an Apache Spark Streaming application. The executors receive tasks to be performed from the driver program, execute these tasks concurrently and send back the results to the driver program.

2.3.1.3. Cluster Manager

The use of a cluster manager is not mandatory, Apache Spark Streaming can be executed in local mode, on a single machine. This setup is not advisable for production deployments as it does not account for fault tolerance.

Apache Spark Streaming is compatible with three cluster managers, namely, Apache Hadoop Yarn, Apache Mesos and Spark Standalone. Apache Hadoop Yarn was employed for cluster implementation as it came bundled with the Hadoop distribution– Hortonworks Data Platform (HDP) -installed in this project’s cluster. Hadoop distributions provide tested, multiple components that work well together, in an easy to install format. In addition, Apache Mesos is not provided with this HDP and would require extra configurations to make it work.

Figure 2 shows the components of an Apache Spark Streaming cluster which is using Apache Hadoop Yarn as a cluster manager and Apache Hadoop HDFS as a distributed filesystem.

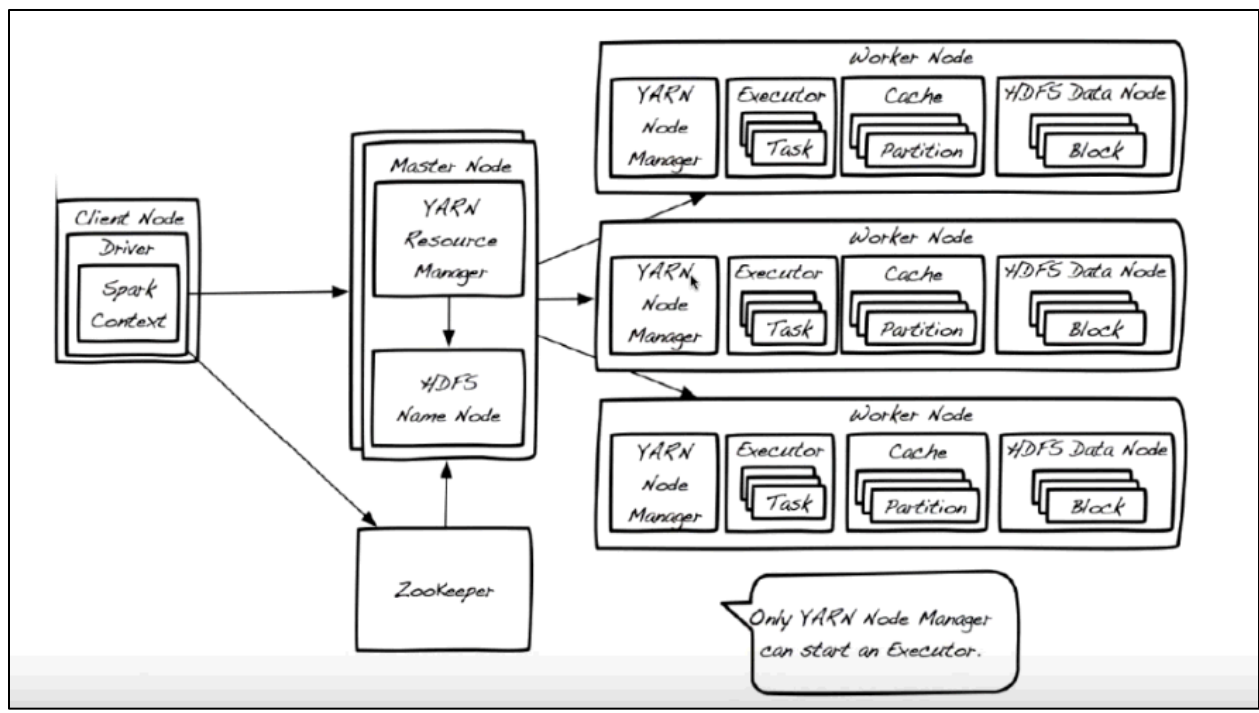


Figure 2: Yarn Based architecture for Apache Spark (Stackoverflow, 2016)

2.4. Apache Hadoop Yarn

Apache Hadoop Yarn is a cluster management tool that provides the functionalities of cluster management and job scheduling. In an Apache Spark Streaming cluster, Apache Hadoop Yarn centrally manages a pool of cluster resources and dynamically shares them between all frameworks running on it.

Apache Hadoop Yarn also follows a master/slave architecture and consists of two main components

1. Node Manager

The Node Manager is a daemon that runs on each slave node. Each slave node in an Apache Hadoop Yarn cluster provides computational resources such as CPU and memory to the cluster forming a pool of resources that a resource manager can use to allocate resources according to application needs. Each node manager tracks the available resources on its slave nodes and sends periodic updates to the resource manager. The node manager is also responsible for starting the executor processes on the Apache Spark Streaming cluster worker nodes, the executor processes run within the Apache Hadoop Yarn containers.

A node manager further contains two components; containers and application master. A container is a collection of all the resources required to run an application and is responsible for the actual task execution. An application can have several containers. The node manager manages the lifecycle of containers, it launches containers as instructed by the resource manager and continually monitors the containers and their resource usage and reports back to the resource manager. The application master is a per-application process and is responsible for negotiating for resources with the resource manager and working with the node managers to execute and monitor tasks

2. Resource Manager

The resource manager is a daemon that runs on the master node, only one resource manager is active in an Apache Hadoop Yarn cluster at any time. The resource manager is responsible for managing resources in the cluster and allocation them to the applications as per their requirements. The resource manager consists of a scheduler and applications manager. The scheduler is a pure scheduler, only allocates resources to applications but does not involve itself in the management or monitoring of applications. The applications manager receives job submissions and maintains a cache of completed applications for a developers use.

These components are shown on Figure 2.

2.5. Apache Hadoop HDFS

An Apache Spark Streaming cluster requires a distributed file system, this facilitates data storage and access by the nodes in the cluster and also helps achieve fault tolerance and scalability. Apache Hadoop HDFS commonly known as HDFS is an open source Java based, distributed file system for storing large volumes of data on commodity hardware. Files are split onto large blocks sized at either 64 Megabytes or 128 Megabytes. HDFS is fault tolerant, it achieves this by replicating blocks of data among the nodes in a cluster, the default block replication level is 3; this means that a block is available on at least three nodes in a cluster. A

HDFS is recommended over other distributed file systems such as SAN (Storage Attached Networks) due to the extra network communication overhead that can cause performance bottlenecks (IBM, HDFS - Apache Hadoop Distributed File System, 2017).

Apache Hadoop HDFS has a master/slave architecture and primarily has two type of nodes

1. The name node

Each cluster can only have one active name node at a time, which runs on the master node. The name node is responsible for managing the filesystem namespace and controlling access to files by cluster nodes (Hadoop, 2013).

2. The data nodes

The data nodes run on the slave nodes, a cluster can have several data nodes, usually one per machine. Data nodes manage the storage attached to the nodes they are running on and are responsible for service read and write requests to the cluster nodes (Hadoop, 2013). The data nodes also perform block creation, deletion and replication upon instructed by the name node.

These components are shown on Figure 2.

2.6. Apache Cassandra

Apache Cassandra is a distributed, NoSQL, column family database designed to store large amounts of data and deliver high availability without a single point of failure (Datastax, 2017). In an Apache Cassandra cluster, all nodes play an identical role, there is no concept of master node and therefore unlike other platforms, no single point of failure. It is highly scalable and fault tolerant and provides very high write throughput and good read throughputs (Perera, 2012). It is known to handle petabytes of data spread across 75,000 nodes (Datastax, 2017) and usually outperforms other similar NoSQL databases in performance benchmarks.

The key components of Apache Cassandra are

1. Token

Apache Cassandra uses a token system to work out which nodes in the cluster will store what partition of data. A token is a range of hashes defined by a partitioner. Tokens are calculated and assigned when a new node joins a cluster.

2. Data center

A datacenter is a collection of nodes. A node is any machine connected to an Apache Cassandra cluster. Each datacenter has a unique name to identify it, nodes use this unique name to know which datacenter they belong to. When a new node is added to the cluster,

one has to assign it to a datacenter, otherwise it will form part of the default datacenter. Nodes in a datacenter are all equal, and according to the token configuration and replication strategy, share workload and data. A collection of datacenters is known as a cluster.

3. Seed node

A seed node is a node within a datacenter from which a new node learns about the cluster and internode communication. Each datacenter must at least have one seed node.

4. Commit log and memtable

When writes occur in an Apache Cassandra database, data is stored in memory in a memtable as well as on disk, on commit logs. Commit logs are durable structures that are designed to hold data even during power failures and are used for recovery in case of failures. Use of commit logs is configurable through a parameter known as `durable_writes`. The memtable stores data written to it in order until a configurable limit is reached, after which data is flushed to the SSTables. During read operations, if data is still on a memtable, read operations are fast.

5. SSTables

SSTables are immutable data structures located on disk, and are populated with sorted data after the memtables are flushed. Each memtable flush results in a new SSTable file, write heavy database operations could lead to multiple SSTables. During read operations, if data is located in the SSTables, the read operation has to query all the SSTable files, this makes the read operations in Apache Cassandra slower than the writes (Datastax, 2017). Periodically, new SSTables are created and old SSTables are consolidated through a process known as compaction

Figure 3 shows the write process in Apache Cassandra

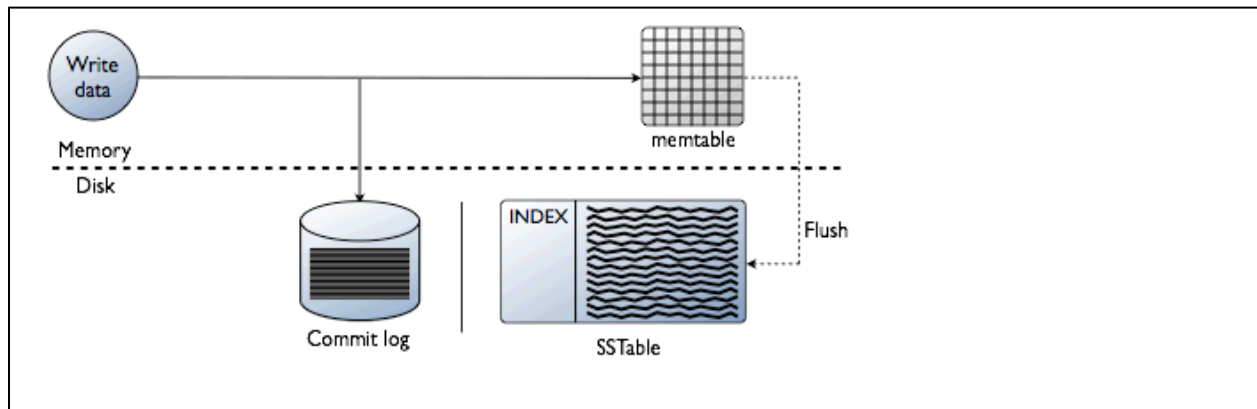


Figure 3: Apache Cassandra Write Process (Ma, 2014)

Compared to common relational databases, Apache Cassandra uses similar concepts of databases and tables. A database in Apache Cassandra is known as a keyspace. When defining a keyspace, one also defines the replication strategy to use, which determines how data in that keyspace will be replicated amongst the available cluster nodes. One also defines whether write operations will write data to the commit logs on the file system.

A table on the other hand is known as a column family which must contain a partition key (primary key) and may contain several columns, each with a unique name and data type and clustering keys. A partition key is responsible for distribution of data across the cluster nodes. Clustering keys serve the purpose of sorting data within a partition. Just like relational databases, Apache Cassandra also has its own language for issuing commands to a database, known as CQL – Cassandra query language.

2.7. Apache Kafka

Apache Kafka is a distributed publish-subscribe messaging systems that facilitates the development of distributed applications by building real-time data pipelines. It is compatible with Apache Spark Streaming and is often used as a data source, generating streams of data for ingestion by Apache Spark Streaming. It is scalable and fault tolerant and capable of handling a trillion messages per day (Confluent, 2017)

The key components of an Apache Kafka cluster include

1. Topics and partitions

Data in an Apache Kafka cluster is stored in a topic. A topic is a stream of messages belonging to a particular user-defined category. Each topic has a unique name and one or more partitions. Each partition is stored in files on the filesystem, distributed and replicated among the cluster node according to the configured partitions. Each topic consists of one or more partitions. Partitions are crucial as they define parallelism in Apache Kafka, more partitions translate to more throughput, as partitions can be written to and read from in parallel.

2. Brokers

Each node in an Apache Kafka cluster is known as a broker. Brokers are responsible for storing and replicating the received data in topics. Brokers are stateless in that they do not track messages read by consumers, consumers have to keep track of messages they have consumed themselves.

3. Producers

Producers send data to one or more brokers which then append this data to a partition. A producer is an application running on a node, the application can be developed using python or Java and is therefore portable to different platforms.

4. Consumers

Consumers read data from one or more topic partitions. Consumers subscribe to a topic or a series of topics and then reads data from them by pulling streams of records. An example of a consumer is Apache Spark Streaming.

An Apache Kafka cluster may consist of one or more brokers that are neither of equal status nor in a master/slave architecture. The brokers however, need a mechanism for communicating and coordinating operations within a cluster. This is where Apache Zookeeper comes in. Figure 4 shows the various components of Apache Kafka including the interrelationship with Apache Zookeeper

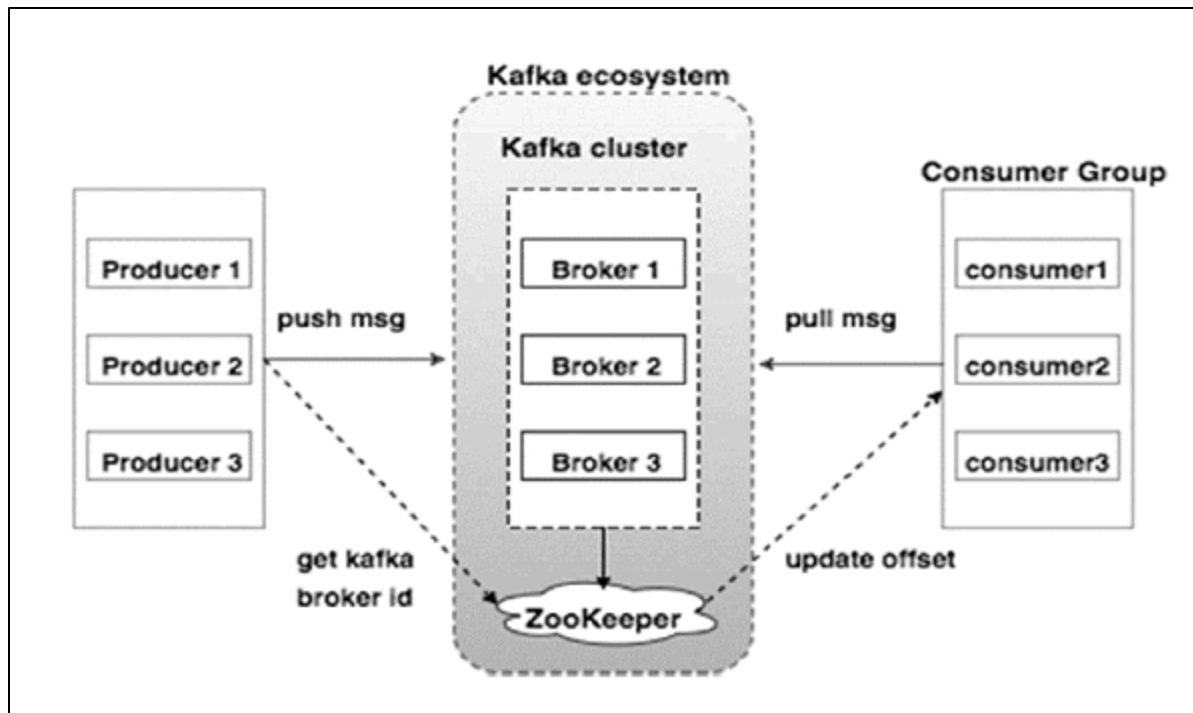


Figure 4: Kafka Ecosystem

2.8. Apache Zookeeper

Apache Zookeeper is a distributed, scalable, fault tolerant co-ordination platform that provides services such as group membership, leader election, coordinated workflow and configuration services as well as distributed data structures such as queues and locks.

Apache Kafka uses Apache Zookeeper for coordinating its brokers, each broker in a cluster coordinates with other brokers in a cluster through Apache Zookeeper. Apache Zookeeper keeps a list of all the brokers in the Apache Kafka cluster it is managing, as well as lists of topics and partitions. Apache Zookeeper also notifies consumers and producers of any new brokers or failed brokers ensuring that data is not sent to a failed broker and consumers do not try to connect to a dead broker.

Apache Zookeeper is also used in other platforms, it works well with Apache Hadoop HDFS, Apache Hadoop Yarn and Apache Spark Streaming. All these three platforms have master/slave architectures. While a cluster can have multiple slave nodes ensuring high availability and fault tolerance, only one master node can be active at any one time. However, having a single master node in a cluster makes it a single point of failure. Therefore, the Apache Hadoop HDFS, Apache

Hadoop Yarn and Apache Spark Streaming clusters have the provisions for running a standby master nodes but have no way of coordinating or monitoring the master nodes. Apache Zookeeper performs this role in these three clusters, as well as other additional roles, chief among them being leader election. If a master node fails, Apache Zookeeper should notice this, and initiate the activities required for making the standby master node, an active master node.

2.9. Graphite and Grafana

Once a cluster has been installed using the platforms discussed in the preceding sections, there needs to be an approach for monitoring them in terms of performance. Both Graphite and Grafana are monitoring and visualization tools that store numeric time series data and produce graphs on demand. Grafana has the advantage of being more flexible when it comes to the creation of reports, having an array of report formats that can be used to render graphs. It however cannot collect raw data from cluster nodes.

On the other hand, graphite has one reporting format, the linear graph, but is capable of periodically collecting raw data from cluster nodes, through a daemon known as collectd.

Collectd is installed on each cluster node and runs in the background, collecting data at intervals based on a configuration file with details on server metrics to monitor and how to monitor them. This data is then forwarded to a centrally located graphite server which stores this data in Round Robin Database (RRD) format.

Grafana then pulls performance data from Graphite and displays them in graphs manually created by users

2.10. Related Work and Research Gap

A research conducted by (Bouillet, et al., 2012) aimed at the processing of CDRs using stream processing and managed to process 6 billion CDRs per day. Their solution employed IBM Infosphere streams platform, a commercial product and a native stream processing platform. Their research project is an experience report detailing their experience in setting up a stream processing solution for processing CDRs, it does not provide information on the performance metrics of latency and the resource usage metrics of disk, memory and CPU. Furthermore, it provides information on the performance metric of throughput, not in seconds, but per day – 6

billion CDRs per day. Without the latency and throughput results in seconds, it is difficult to conclude whether their solution was a real-time CDR data processing solution. The platform of choice used in their research project, IBM Infosphere streams is a commercial product that requires paying for licenses, they do provide a developer edition that can be executed on non-production systems. IBM Infosphere streams inherently guarantees at least once semantics by using re-playable data sources. Exactly once processing semantics are dependent on using external systems that can detect duplicates or restore state after a failure. (Bouillet, et al., 2012), made use of bloom filters for deduplication

(Rebaca, 2017), used the Apache Kafka, Apache Spark MLlib, Apache Spark Streaming and Apache Cassandra platforms to create advanced analytics from streaming CDR data. Rather than evaluate the performance of the streaming platform, they dominantly used Apache Spark MLlib, particularly, KMeans to perform cluster analysis. The versions of the platforms used are also outdated, this research project makes use of newer platforms that are assured to have better performance and features. Their white paper did not provide details on the infrastructure used in terms of hardware.

In their paper, (Agung & Kistijantoro, 2016) discuss the design and implementation of a new parallel processing system for CDR processing using Hadoop MapReduce. Their solution was able to achieve throughputs of 67,000 records per second with latencies as high as 300 seconds. Their platform choice is interesting as they selected to use Hadoop MapReduce in a time where faster and better platforms exist. They provide results for the resource usage metric of CPU utilization but not for memory or disk utilization.

From these, we saw an opportunity to implement a real-time CDR processing platform using Apache Kafka, Apache Spark Streaming and Apache Cassandra platforms, all of which are open source platforms. Apache Spark Streaming is highly scalable and fault tolerant, and most importantly, guarantees exactly once message processing. This research project also measured the performance metrics of latency and throughput in seconds, the results of which aided in the determination of whether the installed solution is capable of real-time CDR processing. The resource usage metrics of CPU utilization, disk utilization and memory utilization were also being measured. This was useful in understanding how resources are utilized in a real-time processing

solution, as well as aid in detecting any resource constraints that would affect the performance of this solution.

CHAPTER THREE

METHODOLOGY

3.0. Introduction

This chapter revolves around the methodology used in the design, implementation and evaluation of real-time processing of CDR data. It includes details on the research design used, system analysis, design and integration, software design and development and how data collection was achieved.

The research project required two major components to be set up for the experiment

1. The system infrastructure

This is the underlying environment installed for the project encompassing the virtual machines provisioned for the project plus the stream processing clusters and tools installed.

2. Software

This is the Java stream processing prototype developed for the real-time processing of CDR data. The prototype was a command line application and was executed on the Apache Spark Streaming cluster as an application.

The keyword 'system' is used to denote the complete implementation, i.e. the system infrastructure and software required to process CDR data in real-time, end to end.

The research design used for this project is based on the experimental research design which involved conducting an experiment to determine whether real-time processing of CDRs is feasible and in particular using Apache Spark Streaming. To answer this question, data on the performance metrics of throughput and latency and resource usage metrics of CPU utilization, disk utilization and memory utilization were collected and analyzed. Our targets were to achieve average throughput rates of 100,000 records per second and latencies of less than 60 seconds. These targets were based on values obtained from the 'Quarter 3 sector statistics report of 2016-2017' by the communication authority of Kenya (CAK, 2017), the top telecommunication company in Kenya recorded a total of 12,696,109,430 SMS messages in a period of 92 days, this translates to 95,834 SMS messages per second assuming even distribution of traffic throughout the day.

The experimental research design enables us to examine how the variables related to the system and software affect the targets set.

This chapter has been further divided into the following three sections, which were performed in order

1. System infrastructure installation

Three clusters for Apache Kafka, Apache Spark Streaming and Apache Cassandra were installed on nine virtual machines. Apache Kafka acted as the input source for streams of records for Apache Spark Streaming to process, the results of which were stored in a database in Apache Cassandra. This section provides details on how these three clusters were installed and configured.

2. Software development

This involved the development of a Java stream processing prototype for Apache Spark Streaming whose purpose was to receive streams of data, process the data while at the same time filtering out surplus data and storing the processed data into Apache Cassandra. This section provides details on the software design and implementation.

3. Experimentation

This section provides a discussion on how the experiment was performed, the fixed and variable parameters used, the evaluation metrics used and how result data collection of performance metrics of latency and throughput and the resource usage metrics of CPU utilization, disk utilization and memory utilization performed.

3.1. System Infrastructure Setup

Three clusters were installed for this research project, one each for Apache Kafka, Apache Spark Streaming and Apache Cassandra. The clusters were installed on a virtualized environment which had enough resources to accommodate this research project's needs. Nine virtual machines were used for this project, 2 virtual machines for Apache Kafka, 2 virtual machines for Apache Cassandra, 4 virtual machines for Apache Spark Streaming and 1 virtual machine for Apache Ambari.

Due to the extensive amount of work required in the installation and configuration of three Big Data clusters, Apache Ambari, an open-source tool that facilitates the rapid provisioning of

Hadoop clusters was deployed. Apache Ambari eased the administration of the clusters as well, instead of performing configurations per host, Apache Ambari was used to distribute configurations to the necessary nodes in the cluster, in parallel. Apache Ambari also provides cluster monitoring services, generating and displaying alarms as well as sending out e-mail alerts. It is also worth mentioning that the Hadoop distribution from Hortonworks Data Platform (HDP) software bundle was utilized. Hadoop distributions pull together all the enhancement projects present in the Apache repository and present them as a unified product so that organizations don't have to spend time on assembling these elements into a single functional component (Dezyre, 2016). HDP was selected as it is completely open source, unlike similar distributions such as Cloudera and MapR which include premium models.

The infrastructure setup is as shown on Table 1

Table 1: Infrastructure setup

	<i>Apache Kafka/Apache Zookeeper Cluster</i>	<i>Apache Spark Streaming Cluster</i>	<i>Apache Cassandra Cluster</i>	<i>Apache Ambari</i>
Nodes	2	4	2	1
Node role	Reads from file, generates streams of data	Performs processing of input data	Column-oriented Database	Cluster provisioning and monitoring tool
Operating system	RHEL 6.5	RHEL 6.5	RHEL 6.5	RHEL 6.5
CPU	8 vCPU	16 vCPU	16 vCPU	4 vCPU
Memory	16 GB	24 GB	48 GB	8 GB
Disk Size	100GB X 6	200 GB X 9	200 GB X 8	350 GB X 1
Network bandwidth	1 Gbps	1 Gbps	1 Gbps	1 Gbps
Filesystem	EXT4	HDFS	EXT4	EXT4

	Apache Kafka/Apache Zookeeper Cluster	Apache Spark Streaming Cluster	Apache Cassandra Cluster	Apache Ambari
Software List	JDK 1.8.0_112	JDK 1.8.0_112	JDK 1.8.0_45	JDK 1.8.0_112
	Python 2.6	Python 2.6	Python 2.7	Python 2.7
	HDP 2.6.1.0 (bundled software)	HDP 2.6.1.0	Datastax Cassandra 3.0.14	HDP 2.6.1.0
	Apache Kafka 0.10.1	Apache Spark 2.1.1	Datastax Cassandra 3.0.14	Apache Ambari 2.5.1.0
	Apache Zookeeper 3.4.6	Hadoop Yarn 2.7.3	Grafana 2.6.0	Graphite 0.9.6
		Apache HDFS 2.7.3		

The cluster sizing of nine nodes was selected based on the performance requirements of 100,000 records per second with latencies of less than 60 seconds. A single node in an Apache Kafka cluster is capable of generating more than 100,000 records per second, indeed in this project’s setup, the installed Apache Kafka cluster was able to achieve throughputs of 150,000 records per second on a single node. It is however advisable to build in fault tolerance in all the three clusters, thus the second node. In addition, the two Apache Kafka nodes were also running the additional services of Apache Zookeeper as well as playing the role of producer. In production deployments, these would be on separate, dedicated nodes.

We settled on an Apache Streaming cluster of 4 nodes, one master node and three worker nodes. This was primarily due to fault tolerance, one of the three worker nodes was also configured as a standby master node, and in case of failure, would take the role of master node. Therefore the number of worker nodes would reduce to two and still be fault tolerant.

The Apache Cassandra cluster had two nodes, also for fault tolerance purposes. In addition, since Apache Cassandra scales linearly (Kuhlenkamp, Klems, & Ross, 2014), the additional second node provided a performance enhancement.

3.2. Prototype Development

In this section, a discussion on the software development model used to develop a stream processing prototype for receiving streams of data, processing them and storing the results into a database, and the steps implemented for the development model is presented

3.2.1. Software Development Model

This research project employed the classic waterfall software development model since the requirements for this project were already known and clearly defined. Moreover, certain tasks had to be accomplished before other dependent tasks could begin.

The specific stages carried out in the development of the stream processing prototype include

3.2.1.1. Requirements Definition and Analysis

Requirements that the prototype should possess were identified and further divided into functional and non-functional requirements

- Functional requirements

These are requirements that a software application should have and are related to the functionalities and behavior of that application. Three functional requirements appropriate for this project were recognized

- The application should be capable of ingesting streams of records generated from Apache Kafka
- The application should be capable of filtering out unwanted data from each stream record
- The application should be able to store processed data into an Apache Cassandra database.

- Non-functional requirements

These are requirements that specify the criteria that was used to assess the operation of an application in terms of quality. We selected to assess the prototype that was

developed based on the performance requirement of throughput and latency. Even though these are system wide targets, the developed prototype played a huge role in ensuring that they were achieved.

- The application is required to process records at an average minimum rate of 100,000 records per second.
- The application is required to process records at an average latency of less than sixty seconds

3.2.1.2. Prototype Design and Development

A stream processing prototype was developed in Java, the function of the prototype being to receive streams of data from Apache Kafka, process this data, filtering out fifteen fields of interest and finally storing the processed data in a database on Apache Cassandra.

Java was selected as the programming language of choice, as we had previous experience with the language. The other programming language options were Scala and Python of which we had limited experience. The Java version used was JDK 1.8.0_45 as it was compatible with Apache Spark Streaming version 2.1.1. The Integrated Development Environment (IDE) used was Netbeans 8.2.

To interface the developed prototype with Apache Kafka, the direct stream approach in combination with Apache Kafka consumer APIs was applied. We used an Apache Cassandra database and to interface to this database we used the Apache Spark Cassandra connector by Datastax.

The stream processing prototype developed contained two classes, the first class was for the receiving, processing and storing streams of data while the second class acted as a mapper class for storing rows of records into Apache Cassandra. Apache Cassandra used this mapper class to determine which fields to map to the columns in the Apache Cassandra table. The mapper class contained a series of getter and setter methods that had to be identical to the column definitions in the table created on Apache Cassandra.

Unified modelling language (UML) was elected for system modelling, UML is considered to be the de-facto modeling language in use today, it as well supports object oriented development which

is a plus as the prototype we were about to develop was object-oriented. A class diagram was created in order to show the relationship between the two classes. The class diagram is as depicted in Appendix 1 which shows two classes, the streaming class and the mapper class. We used the anchor arrow to depict the relationship between this two classes as the mapper class is an inner class in the streaming class.

Additionally, pseudocode was used to map out a high level representation of the prototype as shown in Figure 5.

```
Read and Load Apache Spark Streaming, Apache Kafka and Apache Cassandra properties from a
configuration file
Start a new Java streaming context with a set batch interval
Connect to Apache Kafka brokers and start reading streams of records
For each record stream
    Map a single record stream into a single line of values
    For each line of values
        Trim and split line on “|” character to obtain individual values
        Store each value in a string array
    If length of string array is more than 42
        Read array contents
        Filter out unrequired values
        If there is an existing Apache Cassandra connection
            Insert required values into a table on Apache Cassandra using the
            mapper class
        Else
            Open a new connection to Apache Cassandra
            Insert required values into a table on Apache Cassandra using the
            mapper class
        End if
    End if
End for
```

Figure 5: Pseudocode for Apache Spark Streaming prototype

The stream processing prototype source code is as shown in Appendix 3.

Inputs/Outputs

Thirteen GiB of CDR data was successfully obtained from a local telecommunication company via Secure File Transfer Protocol after obtaining the go ahead from the SMSC team to implement a proof of concept for the real-time processing of CDR for the telecommunication company. After a file sanitization exercise of removing erroneous and duplicate records, the size of the CDR file

reduced to 11 GiB of CDR data containing 19,650,000 records. We found that this size to be sufficient for generating the required input rate of 100,000 records per second for the experiment.

Each line of record in the CDR file has an average size of 600 bytes. This information is significant to Apache Kafka whose throughput performance is affected by large message sizes (Cloudera, 2017). Each line has a maximum of 63 fields, each separated by the special character '|'.

From the 63 fields available, 15 fields of interest were chosen as detailed in the following section. These fields are filtered out during processing by Apache Spark streaming, and therefore, this is the data stored on Apache Cassandra's tables.

- i. col41 - Unique message identifier
- ii. col10 - Destination address
- iii. col16 - Server center timestamp
- iv. col17 - Date when message reached a final status
- v. col18 - Validity period of the message
- vi. col19 - Scheduled delivery time of the message
- vii. col25 - error cause of message delivery failure
- viii. col26 - service type used for charging
- ix. col3 - final status of the short message
- x. col30 - Message Terminator IMSI
- xi. col34 - Message Originator serving MSC
- xii. col35 - Message Terminator serving MSC
- xiii. col37 - Message Originator IMSI
- xiv. col4 – Server center timestamp in YYYY-MM-DD format
- xv. col7 – Originating address

The default producer application provided by Apache Kafka was employed. The use of the default producer application eliminated the need for us to develop a new producer from scratch which saved us time. The producer reads data from a CDR file and converts the data into streams of records and forwards them to Apache Kafka brokers which in turn forward streams of records to Apache Spark Streaming.

The stream processing prototype developed to run on Apache Spark Streaming, received streams of records from Apache Kafka at a set interval. During start up, the developed stream processing prototype read two configuration files, one configuration file was located on the local filesystem, while the second configuration file was located in HDFS. The local configuration file contained details such as

- Cluster manager to use (Yarn)
- Application deployment mode (cluster)
- Executor memory (20G)
- Number of executors (3)
- Number of executor cores (8)
- Apache Cassandra host
- Apache Cassandra concurrent writes (100)
- Apache Cassandra keep alive milliseconds (60000)

The HDFS configuration file contained details such as

- Apache Kafka broker internet protocol (IP) address and port number
- Name of topic which contains streams of records to be read
- The batch interval
- Application name
- Apache Cassandra keyspace name and column family name

The command line parameters for the Apache Kafka Producer included

- Max partition memory bytes (1290)
- Request timeout in milliseconds (40000)

The rest of the parameters for Apache Kafka, Apache Spark Streaming and Apache Cassandra were left at their default values.

The batch interval, also known as a batch duration dictates the interval of processing. Therefore, during this batch interval, the executing prototype received streams of data which were grouped into batches for processing. The larger the batch interval, the bigger the batches.

After each batch of streams of records has been processed, the results were forwarded to Apache Cassandra for storage in batches. Much like the ODBC connector used for interconnecting to

relational databases, Apache Spark Streaming uses the Spark Cassandra connector to interface to Apache Cassandra databases. The results were stored in a table in accordance to the mapper class running as part of the developed prototype.

While the stream processing prototype was executing, metrics were continually being generated in the background. The performance metrics of throughput and latency were displayed on Apache Spark streaming user interface (UI), while the resource usage metrics of disk space utilization, memory utilization and CPU utilization were generated at 10 seconds intervals locally on each machine and forwarded to Graphite which then forwarded the metrics to Grafana.

Logs were also generated in form of output logs and error logs on the individual cluster nodes, these helped confirm the settings under which the program is running and if there were error conditions that should be of concern to us.

The diagram in Figure 6 shows this flow between the various components

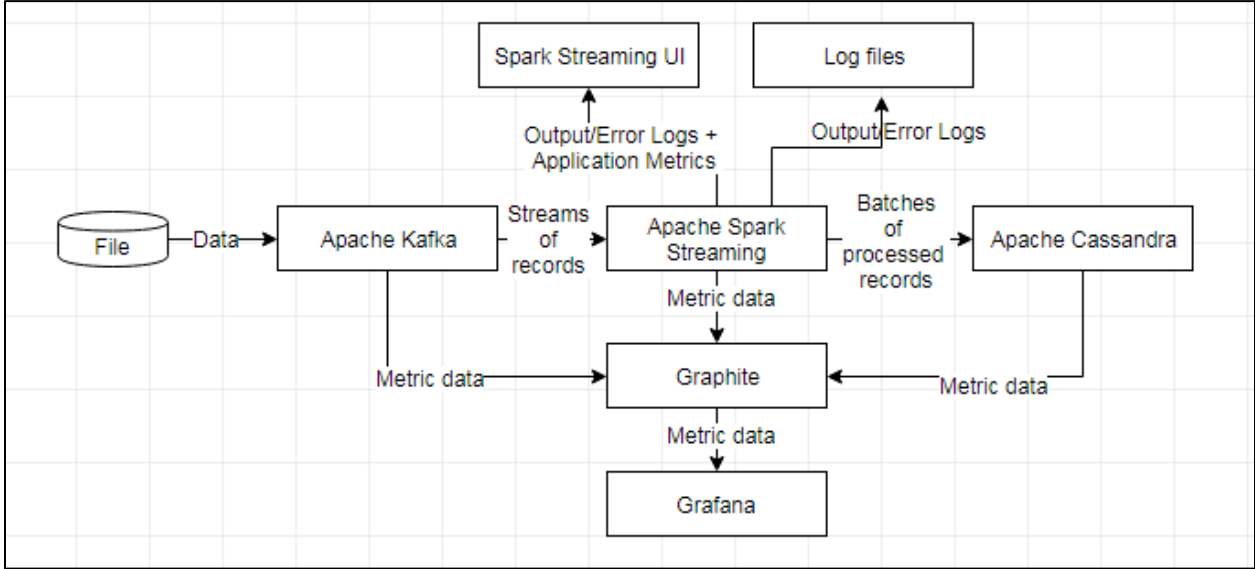


Figure 6: Flow of data between the various installed components

Data Modelling

Data modelling for Apache Cassandra is different from relational databases data modelling. In Apache Cassandra, data modelling is driven by the queries to be executed, while in relational databases, data modelling is based on the data to be stored on tables.

Data modelling in Apache Cassandra is designed to serve two high level goals (Datastax, Basic Rules of Cassandra Data Modeling, 2015)

1. Spread data evenly among the cluster node

The first part of a primary key is known as the partition key, using a hash of the partition key, data is spread amongst the cluster nodes and this is made possible by the use of tokens.

2. Minimize the number of partitions read during query operations

The idea is to have a few partitions for the existing data so that during query operations only a few partitions are scanned. When choosing a partition key, care is made to ensure creation of few partitions that have almost balanced data. If a partition key involves data that is inserted more, then there will be partitions with more data than others. Fewer partitions have an impact on the write performance. Selecting a partition key therefore also depends on the type of transactions a database will handle, is it a write heavy database? Is it a read heavy database? Is it a mixed workload database? A CDR data database is bound to be a mixed workload database, good read performance as well as good write performance is desired, especially considering that we are building a real-time stream processing prototype.

Data modelling around the queries to be executed enabled us to achieve these two goals. The data modelling process therefore involved the following two steps

1. Determining the CQL queries to be executed
2. Creating tables that will satisfy this CQL query by querying as few partitions as possible

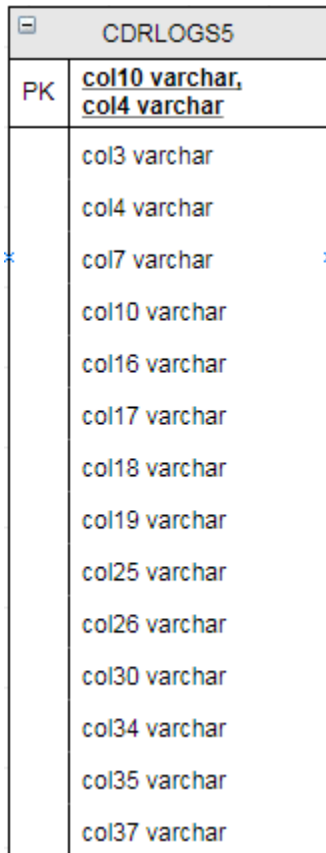
A composite primary key was created, with the first part of the composite primary key being the partition key, while the second part formed the clustering key. While the partition key enforces data distribution among the cluster node in form of partitions, the clustering key is used to group and order items in each partition.

Our solution required a single table named CDRLOGS5 that housed all the result data from Apache Spark Streaming. The CDRLOGS5 table had 15 columns, each column of type varchar. One composite primary key was set to (col10, col4), col4 stores the date (YYYY-MM-DD) when the SMS transaction took place and is derived from col16. The col10 holds the originating addresses, i.e. the sender of an SMS. The col10 was chosen to be the partition key based on the assumption

that more complaints are received from the senders of SMS, it is therefore practical to query the tables, filtering using col10.

This would allow us to write this query, *select * from CDRLOGS5 where col10='254712345678' order by col4;*

The logical model for the CDRLOGS5 table is as shown on Figure 7



CDRLOGS5	
PK	<u>col10 varchar,</u> <u>col4 varchar</u>
	col3 varchar
	col4 varchar
*	col7 varchar *
	col10 varchar
	col16 varchar
	col17 varchar
	col18 varchar
	col19 varchar
	col25 varchar
	col26 varchar
	col30 varchar
	col34 varchar
	col35 varchar
	col37 varchar

Figure 7: Logical model for the CDRLOGS5 table

The columns named col10 and col4 represent the composite primary key, the rest of the columns represent the following data

- i. col41 - Unique message identifier
- ii. col10 - Destination address
- iii. col16 - Server center timestamp
- iv. col17 - Date when message reached a final status
- v. col18 - Validity period of the message
- vi. col19 - Scheduled delivery time of the message

- vii. col25 - error cause of message delivery failure
- viii. col26 - service type used for charging
- ix. col3 - final status of the short message
- x. col30 - Message Terminator IMSI
- xi. col34 - Message Originator serving MSC
- xii. col35 - Message Terminator serving MSC
- xiii. col37 - Message Originator IMSI
- xiv. col4 – Server center timestamp in YYYY-MM-DD format
- xv. col7 – Originating address

3.2.1.3. System Testing

Functional tests were carried out throughout the development process of the Java stream processing prototype to confirm that the code was working as expected and to debug problematic code folds. Other than functional tests, unit testing and integration testing was also carried out

- Unit tests

Unit tests are done to assess an individual unit or group of unites. Unit tests were especially performed during the installation of the 3 clusters of Apache Kafka, Apache Spark Streaming and Apache Cassandra. The 3 clusters were designed and implemented one after the other and therefore when a cluster was up and running, unit tests accompanied by stress tests were done. Ultimately, we were checking whether all the nodes in a particular cluster were communicating, accessing data in a proper manner and load sharing the workload. Logs generated on each node were employed for unit tests, Java’s JConsole and JMX (Java management extensions) were also used to view performance metrics per node. Wireshark was utilized to observe communication between nodes.

With stress tests, we were checking whether a clusters performance in terms of throughput and latency was satisfactory in comparison to published benchmarks. The performance metrics generated by JMX and viewed on JConsole aided in this.

- Integration testing

Once installation and configuration of the three clusters was complete and software development was finished, integration tests were performed with a smaller data set of 2 GiB, this time checking that data generated at the input source (Apache Kafka) was processed successfully by Apache Spark Streaming and stored at the end destination (Apache Cassandra). We were also interested to see whether communication between the three clusters was happening as expected in particular whether responses were being sent back for requests. Integration tests were mainly carried on one worker node, using yarn on local mode and 4 cores with configurations passed through the command line. Running on local mode allowed us to view logs on the console in real-time, following the actions carried out while confirming that the output displayed was as expected. The tools used for integration testing include Wireshark for observing inter-cluster communication including the requests and responses sent, logs from each node and Graphite and Grafana to check on the performance metrics of latency and throughput.

3.2.1.4. Software Implementation

After the testing stage was complete, the developed stream processing prototype was launched on all the 3 worker nodes using yarn on cluster mode and configurations read from configuration files on the local filesystem and HDFS. The command used to execute the stream processing prototype is as shown on Figure 8

```
[spark@thk-oss-spark-nn1 ~]$ spark-submit \  
> --properties-file /opt/project/conf/mySpark.properties \  
> --class com.uonbi.dct.streamingcdrs.DirectKafkaStreamingCDRs \  
> --jars /opt/project/jars/spark-cassandra-connector_2.11-2.0.2.jar,\  
> /opt/project/jars/spark-streaming-kafka-0-10_2.11-2.1.1.2.6.1.0-129.jar,\  
> /opt/project/jars/jsr166e-1.1.0.jar /opt/project/jars/StreamingCDRS-2.0.jar
```

Figure 8: Executing the Stream processing prototype

The prototype picks configuration parameters from the local configuration file 'mySpark.properties', the contents of which are described in section 3.2.1.2 under

‘Inputs/Outputs’, next the Java class that contains the main method is specified together with the packages required to run the prototype.

Streams of data were generated from an 11 GiB input CDR file. Resource usage metrics of CPU, disk and memory utilization were collected from Grafana while the performance metrics of throughput and latency were collected from Apache Spark Streaming UI.

3.3. Experimentation

This section provides details on how the experiment was carried out including the tools utilized and the steps undertaken in the experiment. It also deals with the parameters configured for the experiment and the evaluation metrics used in the collection of result data.

The experiment’s aim was to aid us in answering the research question ‘Is real-time processing of CDRs using stream processing feasible?’ with the objective of processing 100,000 records per second and with latencies of less than 60 seconds on the system.

3.3.1. Fixed Parameters

The following parameters remained unchanged during the experiment

1. Number of nodes: 9 nodes (4 Apache Spark Streaming nodes, 2 Apache Kafka nodes, 2 Apache Cassandra nodes, 1 Apache Ambari node)
2. Apache Kafka batch size: 1500
3. 1 Topic with 24 partitions
4. Input data size: 11 Gib containing 19650000 lines of CDRs
5. Number of spark workers (executors): 3
6. Executor memory: 20 Gib per executor
7. Executor CPU cores: 8 per executor

3.3.2. Variable Parameters

The following parameter was altered during the experiment

1. Batch interval

The reasoning behind this was two-fold, firstly to observe the effect batch interval had on the performance metrics of throughput and latency and resource usage metrics of CPU utilization,

memory utilization and disk utilization. Secondly, to determine the optimal batch interval that would offer the highest throughput and the lowest latencies.

We started off with the minimum supported batch interval of 500 milliseconds, with the goal of determining whether the installed platform was capable of millisecond latencies i.e. latencies that are less than one second. After that a batch interval of 1 second was configured, increasing it in increments of two seconds till a maximum of 11 seconds. For each batch interval, the experiment was conducted in three iterations in order to level out any extremes or outside the norm values. Each iteration resulted in result data related to the performance metrics of latency and throughput and the resource usage metrics of CPU utilization, memory utilization and disk utilization. During result analysis, values obtained from the three iterations were averaged out to obtain the final figures.

3.3.3. Evaluation Metrics

The evaluation metrics were sourced from the performance metrics of throughput and latency and the resource usage metrics of CPU utilization, memory utilization and disk utilization, as these were primarily the metrics that needed to be measured in order to determine whether real-time processing of CDRs was feasible.

The evaluation metrics on Table 2 guided us during result data collection as they describe each metric, dictating what result data is expected from each metric and the tools to use to collect this data

Table 2: Evaluation metrics

Metric	Category	Description	Recording method
Throughput (records per second)	Performance metric	Count of processed records per second	Spark Streaming statistics UI after each iteration under the label 'Input Size'
Latency (Seconds)	Performance metric	Time in milliseconds taken to process a batch of records from the time it arrives in Apache Spark Streaming to the time it is stored in Apache Cassandra tables	Spark Streaming statistics UI after each iteration under the label 'Total Delay'
CPU Utilization (%)	Resource usage metric	Maximum amount of CPU used by the 6 Apache Spark Streaming nodes in percentage	Collectd, Graphite and Grafana
Memory Utilization (GiB)	Resource usage metric	Maximum amount of Memory used by the 6 Apache Spark Streaming nodes in Gb	Collectd, Graphite and Grafana
Disk Utilization (GiB)	Resource usage metric	Average amount of disk used by the 4 Apache Spark Streaming nodes in Gb	Collectd, Graphite and Grafana

3.3.4. Experiment Execution

For each batch interval, the experiment was done in three iterations. The steps performed for each batch interval are

1. Truncate table 'CDRLOGS5' on keyspace 'streamingcdrs1' on Apache Cassandra
2. Change batch interval on Apache Spark Streaming configuration file located in HDFS
3. Start a new Apache Spark Streaming context by executing the developed stream processing prototype in cluster mode

4. On one of the Apache Kafka nodes, run the console producer script to read data from the 11 Gib input CDR file. This is the first iteration
5. When entire file has been read and execution on Apache Spark streaming complete, record throughput and latency values from the Apache Spark Streaming statistics UI. Record CPU, disk and memory utilization values from Grafana
6. Truncate table 'CDRLOGS5' on keyspace 'streamingcdrs1' on Apache Cassandra
7. On one of the Apache Kafka nodes, run the console producer script to read data from the 11 Gib input CDR file. This is the second iteration
8. When entire file has been read and execution on Apache Spark streaming complete, record throughput and latency values from the Apache Spark Streaming statistics UI. Record CPU, disk and memory utilization values from Grafana
9. Truncate table 'CDRLOGS5' on keyspace 'streamingcdrs1' on Apache Cassandra
10. On one of the Apache Kafka nodes, run the console producer script to read data from the 11 Gib input CDR file. This is the third and final iteration for set batch interval
11. When entire file has been read and execution on Apache Spark streaming complete, record throughput and latency values from the Apache Spark Streaming statistics UI. Record CPU, disk and memory utilization values from Grafana
12. Stop currently running Apache Spark Streaming context
13. After execution of the last batch interval has been completed, merge all the result data in one Microsoft Excel file
14. Analyze the result data recorded from all the iterations of each batch interval and generate graphical representations of the data

3.3.5. Result Data Collection and Analysis

The performance metrics of throughput and latency were obtained from Apache Spark Streaming, Streaming statistics UI which contains data for

- i. Batch time – this shows the time a batch was submitted in the format 'YYYY-MM-DD HH24:MI:SS'
- ii. Input Size – Number of records submitted for the batch corresponding to the batch time

- iii. Scheduling delay – time taken by the streaming scheduler to submit jobs of a batch
- iv. Processing time – time taken to process all jobs of a batch
- v. Total delay – total time taken to handle a batch and is a summation of scheduling delay and processing time.

For each batch interval, throughput values were obtained from the ‘input size’ column and recorded after each iteration. This throughput was then divided by the batch interval to obtain the final average throughput in records per second

$$\text{Throughput (records per second)} = \text{throughput}/\text{batch interval}$$

A summation of all the values for throughput per second resulted in total throughput per second, the figure obtained was then divided by 3, as the experiment was done in iterations of three. The end result being an average throughput in records per second for that batch interval.

$$\text{Average throughput (records per second)} = \text{Total throughput (records per second)} / 3$$

For each batch interval, latency values were obtained from the ‘total delay’ column and recorded in milliseconds. To obtain latency in seconds, this latency was divided by 1000

$$\text{Latency in seconds} = \text{latency in milliseconds}/1000$$

A summation of all the values resulted in total latency which was later divided by 3, this is because the experiment was done in iterations of three. The end result being an average latency in seconds for that batch interval.

The resource usage metrics of CPU utilization, disk utilization and memory utilization were obtained from the Grafana. CPU utilization was in percentage form, while disk utilization and memory utilization were obtained in Gigabytes. A visualization was created in form of a table containing all the three resource usage metrics for easy recording. These values had the lowest granularity of one minute. To obtain average CPU utilization in percentage, we summed CPU utilization across all the nodes in the Apache Spark Streaming cluster to obtain *Total CPU Utilization* and then divided the Total CPU utilization by three, as the experiment was conducted in three iterations, to obtain *Average CPU Utilization*.

$$\text{Total CPU Utilization} = \text{CPU Utilization for node1} + \text{CPU Utilization for node2} + \text{CPU Utilization for node3} + \text{CPU Utilization for node4}$$

$$\text{Average CPU Utilization} = \text{Total CPU Utilization}/3$$

To obtain average memory utilization in Gigabytes, memory utilization was summed across all the nodes in the Apache Spark Streaming cluster to obtain *Total Memory Utilization* and then divided the Total Memory utilization by three, as the experiment was conducted in three iterations, to obtain *Average Memory Utilization*.

$$\textit{Total Memory Utilization} = \textit{Memory Utilization for node1} + \textit{Memory Utilization for node2} + \textit{Memory Utilization for node3} + \textit{Memory Utilization for node4}$$

$$\textit{Average Memory Utilization} = \textit{Total Memory Utilization} / 3$$

To obtain average disk utilization in Gigabytes, disk utilization was summed across all the nodes in the Apache Spark Streaming cluster to obtain *Total Disk Utilization* and then divided the Total Disk utilization by three, as the experiment was conducted in three iterations, to obtain *Average Disk Utilization*.

$$\textit{Total Disk Utilization} = \textit{Disk Utilization for node1} + \textit{Disk Utilization for node2} + \textit{Disk Utilization for node3} + \textit{Disk Utilization for node4}$$

$$\textit{Average Disk Utilization} = \textit{Total Disk Utilization} / 3$$

CHAPTER FOUR

RESULTS AND DISCUSSIONS

This chapter has been split into two sections, the first section deals with the results obtained from the research project while the second section offers discussions on the results. The result section has further been split into

- a. Streaming platforms feature comparison results
- b. System infrastructure installation results
- c. Prototype development results
- d. Experiment results

4.0. Results

4.0.1. Streaming Platforms Feature Comparison Results

A feature comparison was performed on three streaming platforms in use today, with the goal of selecting an appropriate streaming platform for use in this research project. The three were selected based on their adoption rate or customer base as evidenced on their 'powered by' webpages, as well as the number of committers involved in the platforms' projects. The three platforms demonstrated that they have the largest customer and committers' base.

The three streaming platforms chosen for comparison were Apache Spark Streaming, Apache Flink and Apache Storm Trident. Eight features were selected for the comparison and they included scalability, fault tolerance and high availability, message delivery guarantees, adoption level and maturity, streaming model, multiple library support, programming language support and state management, the results of which can be found in appendix 2.

The results show that the three platforms have almost similar mechanisms for achieving a high fault tolerant, high availability and highly scalable platform. Three distinguishing features were immediately established

1. Exactly once message processing

All three platforms, Apache Spark Streaming, Apache Storm Trident and Apache Flink provided strong guarantees for exactly once message processing

2. Iterative processing

Apache Storm Trident does not support iterative processing while Apache Flink and Apache Spark Streaming both support iterative processing.

3. Multiple library support

While both Apache Flink and Apache Spark Streaming contain multiple libraries on the same platform, several Apache Flink libraries are still in development.

Eventually Apache Spark Streaming was chosen as the suitable stream processing platform for this research project.

4.0.2. System Infrastructure Installation Results

This research project was installed on a virtual machine environment consisting of nine virtual machines, configured differently depending on the cluster requirements. Three clusters were installed, each for Apache Kafka consisting of two virtual machines, Apache Spark Streaming consisting of 4 virtual machines and Apache Cassandra consisting of two virtual machines. A cluster management and provisioning tool known as Apache Ambari was installed, Apache Ambari allowed us to monitor and rapidly provision the three clusters. The complete installed system was capable of processing CDR data end to end sufficiently.

4.0.3. Prototype Development Results

This research project required the development of a stream processing prototype that would receive streams of records from an input source, process the streams of records while filtering out surplus information and storing the processed results in a database. A stream processing prototype in Java and Apache Maven was developed using the Netbeans IDE 8.2, the result of which was a command line application that fit the stated requirements.

4.0.4. Experiment Results

In this section, we present the results obtained from the experiment conducted. These results align with the evaluation metrics as shown on Table 2. From the results, we expect to determine whether our goal of processing 100,000 records per second with latencies of less than 60 seconds was achieved. We were also interested in observing resource utilization in the installed platform through the resource usage metrics of CPU utilization, disk utilization and memory utilization.

This section has further been divided into two sections. The first section deals with the performance metrics of throughput per second and latency in seconds while the second section deals with the resource usage metrics of CPU utilization, disk utilization and memory utilization.

4.0.4.1. Performance Metrics Results

The graph on Figure 9, shows the performance metrics of latency and throughput, particularly in relation to Apache Spark Streaming batch interval. One of the objectives of having a variable batch interval was to observe its effect on throughput and latency, as well as select a batch interval that offers maximum throughput while maintaining low latencies.

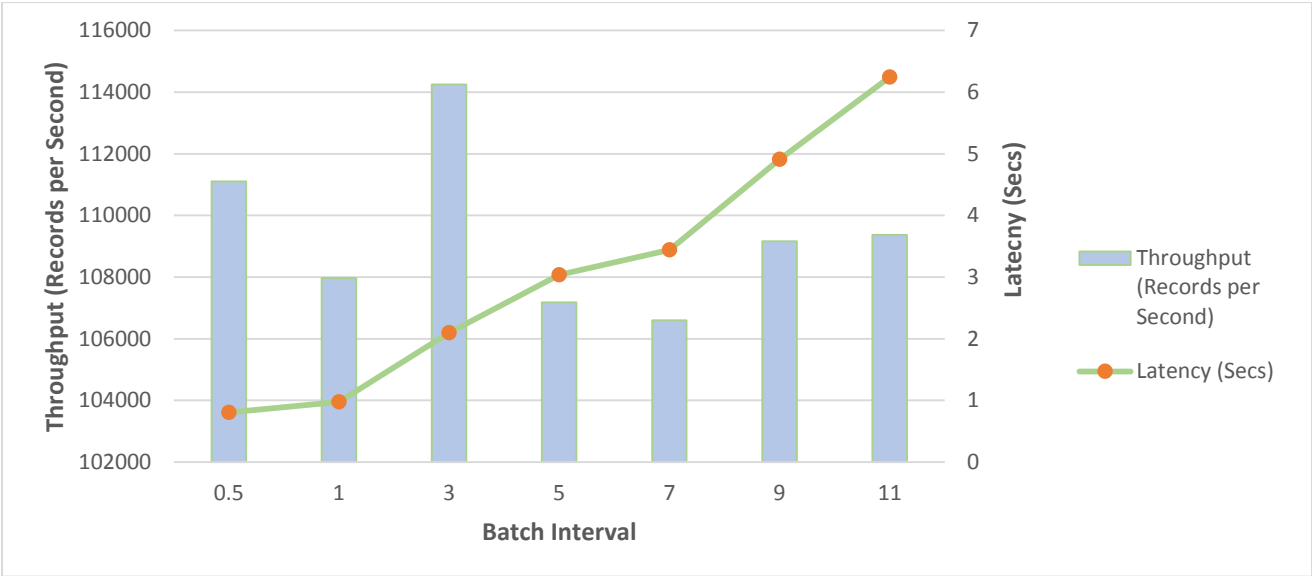


Figure 9: Performance metrics per batch interval

This graph shows that we were able to achieve throughput rates above 100,000 records per second across all the batch intervals, with the highest throughput of 114,244 records per second being obtained for the batch interval of 3 seconds. There is no clear pattern for throughput per second, with throughput rates increasing and decreasing indiscriminately between batch intervals.

Latency on the other hand grows linearly with increasing batch interval. This is expected as a larger batch interval, allows Apache Spark Streaming a longer time to gather input records and form a batch for processing. Of importance though, is to check that latency stays below the set batch interval and this is accomplished for all batch intervals except on the batch interval of 0.5

milliseconds. A latency larger than the batch interval is an indication of a struggling system, a system that cannot cope with the rate of input data and therefore falls behind. The first batch interval of 500 milliseconds demonstrated that we are capable of achieving sub-second latencies, the average latency in seconds recorded is 0.8 seconds which translates to 800 milliseconds. The best performance for this experiment is achieved with a batch interval setting of 3 seconds, at this batch interval, we obtained the highest throughput. Average latency of 2.1 seconds is well below the batch interval of 3 seconds. It would therefore make sense to select the batch interval setting of 3 seconds as the preferred batch interval.

Figure 10 shows the performance metrics of throughput and latency specifically for batch interval setting of 3 seconds only

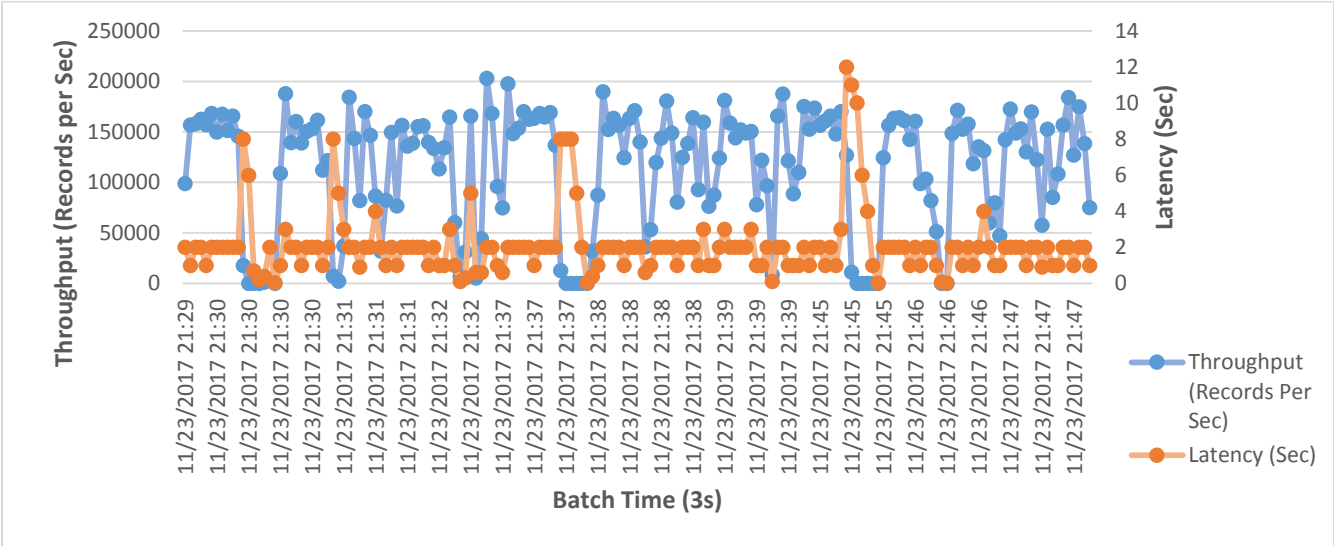


Figure 10: Performance metrics for batch interval setting of 3 seconds

We are able to observe consistent throughputs of more than 150,000 records per second and a maximum latency of 12 seconds. We are also able to observe that the latency spikes are usually followed by a drastic dip in throughput, with throughputs reaching a low of below 100 records per second.

As Apache Spark Streaming is a micro-batch stream processing platforms, we were curious to observe the correlation between batching and record processing, this is shown in Figure 11.

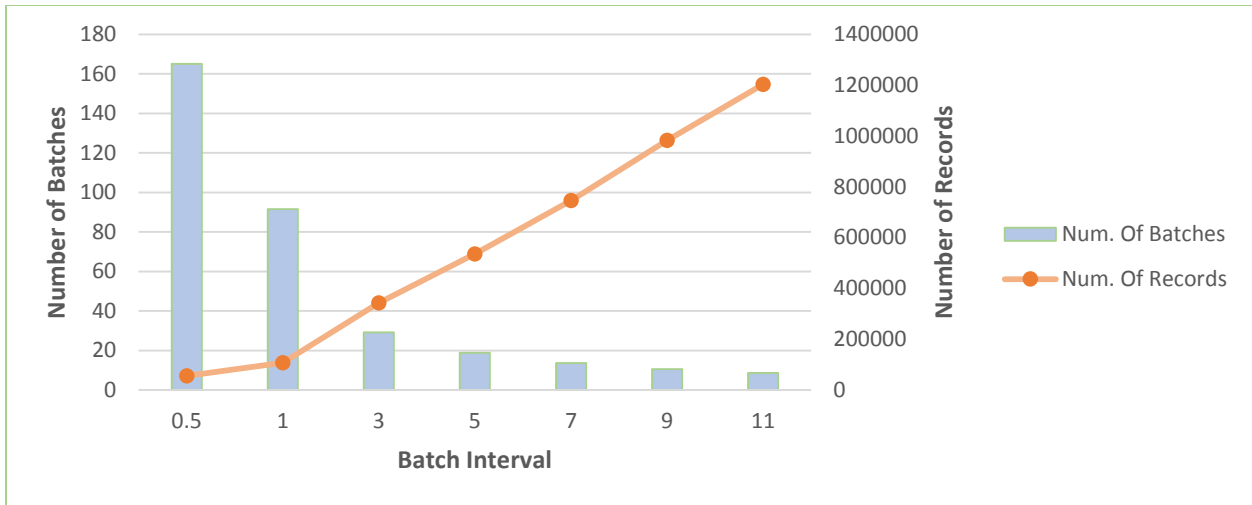


Figure 11: Correlation between batching and record processing

4.0.4.2. Resource Usage Metrics

The graph on Figure 12 displays the resource usage metrics of average disk utilization in Gigabytes, average memory utilization in Gigabytes and average CPU utilization in percentage, in relation to batch interval settings.

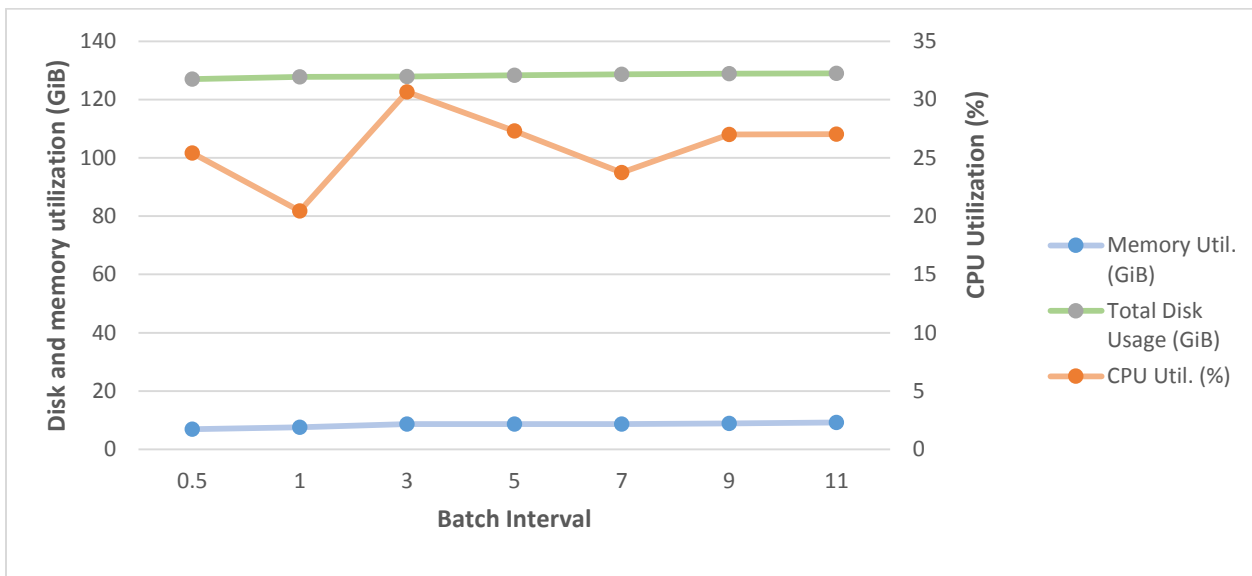


Figure 12: Resource usage metrics vs Batch interval

From this graph, we are able to tell that memory utilization and disk utilization remain almost constant, with very little variation between batch interval settings. However, CPU utilization

varies across batch intervals, with a maximum value of 30 % recorded at batch interval of 3 seconds.

Since memory utilization and disk utilization remain almost constant across batch intervals, we were interested in finding out whether there was any correlation between CPU utilization and throughput. This is because one would expect higher CPU utilization when processing more records at a higher throughput. This resulted in the graph on Figure 13.

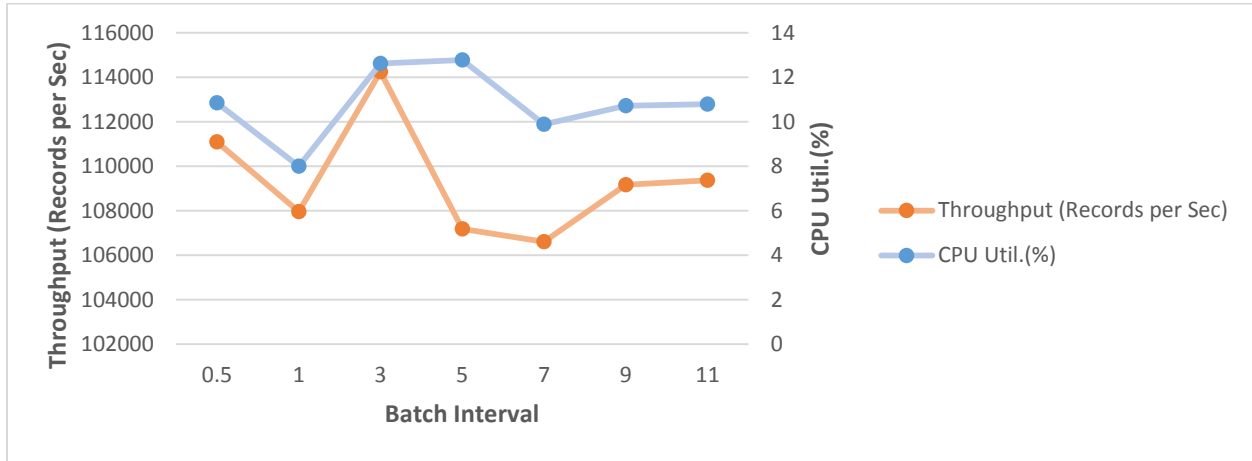


Figure 13: CPU utilization vs Throughput per Second

4.1. Discussions

We start off the discussions section with the performance metrics of latency and throughput followed by the resource usage metrics of CPU utilization, disk utilization and memory utilization. Thereafter a discussion on the outcomes of the prototype development and system infrastructure installation is presented.

We were able to achieve our goal of processing 100,000 records in less than 60 seconds across all our batch interval settings. Moreover, we were able to obtain sub-second latencies for the batch interval setting of 0.5 seconds, achieving an average latency of 0.8 seconds. Since the batch interval setting of 3 seconds had the highest throughput, it was selected as the preferred batch interval for processing. This batch interval recorded the highest throughput of 114,244 records per second with an average latency of 2.1 seconds.

In Figure 10, we have a graph displaying throughput and latency values specifically for batch interval setting of 3 seconds, the graph shows that our system did not maintain a throughput of 100,000 records per second throughout, we are able to register throughputs as high as 200,000

records per seconds and as low as 100 records per second. We also notice latency spikes as high as 12 seconds. These momentary latency spikes are immediately followed by a dip in throughput per second. This is due to Apache Spark Streaming controlling the data ingestion rate by requesting Apache Kafka to slow down the rate of sending data. This enables Apache Spark Streaming to control the input rate thus achieving stable processing times.

If latencies continually increase without any abatement, drastic measures need to be taken. This can be done by optimizing the platform using the available configuration parameters in Apache Spark Streaming or by scaling the system either vertically (by adding processing components) or horizontally (by adding more cluster nodes). Apache Spark Streaming provides several parameters for controlling the ingestion rate, chief among these are `spark.streaming.backpressure.enabled` and `spark.streaming.kafka.maxRatePerPartition`. The `spark.streaming.backpressure.enabled` parameter facilitates the controlling of ingestion rate based on the current processing time so that the system receives only as fast as it can process (Spark, 2017). It is advisable to set this parameter in production systems to avoid the system being overburdened due to an influx of record streams. The `spark.streaming.kafka.maxRatePerPartition` parameter dictates the maximum rate per partition at which data can be read from Apache Kafka, again this ensures that Apache Spark Streaming can process the data it receives efficiently, while maintaining good throughputs and delays. These parameters were not configured in our system as we sought to establish the maximum load our system could handle.

Figure 9 demonstrates that our system is capable of sub-second latencies. For the batch interval setting of 0.5 seconds, we had an average latency of 0.8 seconds. This does seem to answer the research question 'Is real-time processing of CDRs using stream processing feasible?' Taking the definition of real-time as the ability to process data in less than a second, it seems that our system is indeed capable of real-time processing of CDR data. However, we have since learnt that a stable Apache Spark stream processing system should have latencies below the set batch interval and the batch interval setting of 0.5 seconds with an average latency of 0.8 seconds does not signify a stable platform. Selecting such a batch interval for a production system could lead to significant stability and performance problems.

Figure 11 shows the 'micro-batch' stream processing nature of Apache Spark Streaming. We are able to tell that records are grouped into batches at regular intervals, depending on the batch interval. Low batch interval results in more batches than high batch intervals. This can be explained by the fact that high batch intervals are allowed more time to gather streams of records into a batch, therefore their batches will have more streams of records. Low batch intervals have less time to gather streams of records into a batch, thus, their batches will contain fewer streams of records, meaning that more batches have to be generated to complete processing of an entire data set.

Figure 12 displays the resource usage metrics of CPU utilization, memory utilization and disk utilization. Memory and disk utilization remain almost constant while CPU utilization fluctuates across batch intervals.

CPU as a resource is very important in distributed applications as it is majorly a source of concurrency. Use of multiple CPU cores increases concurrency. In Apache Spark Streaming, multiple CPU cores enhance parallelism, allowing multiple tasks, each assigned to a different core to run at the same time. The additional unit for parallelism in Apache Spark is the number of partitions each RDD has, and as we are using the direct stream approach to connect to Apache Kafka, there is a one to one mapping between the number of partitions in an Apache Kafka topic and Apache Spark Streaming. Each executor has been allocated 8 cores, our system has 3 executors bringing the total CPU core count to 24 cores. This means that a maximum of 24 tasks can be run in parallel. Increasing tasks leads to better performance. CPU utilization is highly dependent on the transformations included in an Apache Spark Streaming application. Shuffle operations that require data to be shuffled across partitions in various executors lead to higher CPU utilization. The maximum CPU utilization noted in our system was 31%, this is reasonable as this project's prototype makes use of one transformation as shown in Figure 14, the map transformation, which is not known to be CPU intensive.

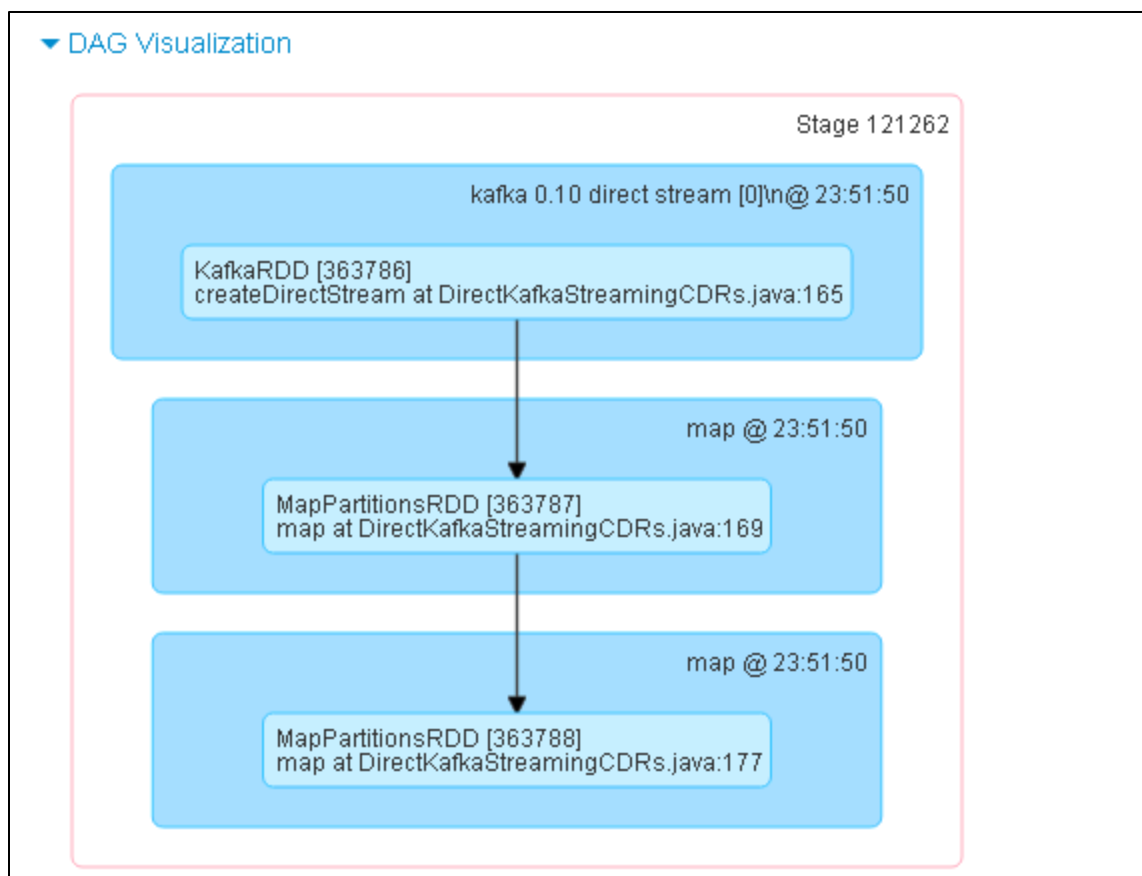


Figure 14: DAG Visualization

From the graph in Figure 13, we are able to decipher a correlation between CPU utilization and throughput per second. CPU utilization increases with increase in throughput per second and decreases with decrease in throughput per second. This is reasonable as we expect CPU utilization to be affected by throughput rates. The question then is, resource usage for our system seems reasonable, why aren't we processing data faster? We found that performance of the system is impacted by the target downstream systems, in our case, this was Apache Cassandra. Apache Cassandra's performance has a direct impact on Apache Spark Streaming. If Apache Cassandra cannot write to its tables as fast as Apache Spark Streaming processes, then performance is affected. Apache Cassandra's performance is affected by various factors including message size, replication factor and data modelling. Performance metrics displayed on Grafana demonstrated that the installed Apache Cassandra cluster was capable of writing 30,000 rows per second which is well below the 100,000 records per second target.

Apache Spark Streaming is an in-memory processing platform. When it receives streams of records from Apache Kafka, they are written into a dstream which is a continuous sequence of RDDs. RDDs are stored in memory and distributed across the executor nodes. During task execution, Apache Spark Streaming makes use of the BlockManager which provides an interface for fetching blocks both locally and remotely using stores such as disk and memory (Laskowski, BlockManager — Key-Value Store for Blocks, 2017). For each of the stores, the BlockManager creates its own instances known as MemoryStore and DiskStore. The MemoryStore is responsible for storing data into memory and it does this in form of deserialized Java Objects or ArrayBuffers. At the start of each job, the MemoryStore stores blocks as values and as bytes into memory. This is on the driver node. The BlockManager then distributes these blocks to all the executors, one block per executor. Each block is an RDD.

The following log excerpt shows the MemoryStore storing blocks of data in memory, as values and as bytes, it also shows the estimated size of these blocks of data with blocks of value having a larger size than blocks of bytes

```
MemoryStore: Block broadcast_3 stored as values in memory (estimated size 15.1 KB, free 366.2 MB)
MemoryStore: Block broadcast_3_piece0 stored as bytes in memory (estimated size 6.8 KB, free 366.2 MB)
```

When the job is finished, the BlockManager removes the RDDs from memory on each executor. This goes to show that we do not expect heavy memory utilization from such processing, as data is stored in form of de-serialized Java object, there is very little memory consumption.

Memory utilization on Apache Spark Streaming is also dependent on caching. For computations that require iterative processing, one may opt to persist RDDs in memory so that data is not fetched from disk every time it needs to be used, if data cannot fit into memory, then the excess data is spilled onto disk. Apache Spark Streaming provides the STORAGE_LEVEL parameter to control this, with options such as MEMORY_ONLY, DISK_ONLY and MEMORY_AND_DISK. These dictate where RDDs should be stored. The STORAGE_LEVEL configured will also have an effect on CPU utilization, data fetching from disk is more CPU intensive than data fetching from memory. As we obtained memory utilization values from graphite which monitors the node, memory utilization also includes Java heap usage. Apache Spark Streaming is written in Scala and Scala code also runs on a Java virtual machine, which requires its own memory. Our prototype did not

persist RDDs into memory, firstly, because we did not have any computations requiring iterative processing and secondly, persisting data into memory would have affected our experiment results on the second and third iterations.

Our Apache Spark Streaming prototype was developed using the Java programming language, as such, it executes in a Java Virtual machine which requires its own memory in form of Java Heap. We can attribute a lot of the memory utilization to this heap usage. Each executor has been allocated 18 GiB of Java heap.

Disk utilization remains almost constant in this experiment. From Apache Spark Streaming UI, no data was written onto the disks. Disk utilization also depends on the STORAGE_LEVEL used, if DISK_ONLY, data is stored on disk, if MEMORY_ONLY, data is stored in memory, if it cannot fit in memory, then excess data is spilled onto the underlying disks.

Figure 15 is used to corroborate the fact that no disk was utilized by Apache Spark Streaming in the processing of CDRs

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Time	Shuffle Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
											(GC Time)					
driver	10.102.16.75:40208	Active	71	490.3 KB / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
1	thk-oss-spark-wn1:57807	Active	71	490.3 KB / 11.3 GB	0.0 B	8	0	0	23984	23984	24 min (21 s)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	thk-oss-spark-wn2:51762	Active	71	490.3 KB / 11.3 GB	0.0 B	8	0	0	23984	23984	25 min (17 s)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	thk-oss-spark-wn3:49963	Active	71	490.3 KB / 11.3 GB	0.0 B	8	0	0	23984	23984	25 min (21 s)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Figure 15: Apache Spark Streaming Storage

However, we do expect disk space to be used in storing logs from the platform – Apache Spark logs, HDFS logs and Hadoop Yarn logs. Considering that the disk utilization in our graphs represents the total disk space used by all our 4 nodes in the Apache Spark Streaming cluster, the disk utilization is very minor. Each node has a total of 1.6TiB of disk space while the maximum total utilization is at 130 GiB.

Performance of the entire system is heavily dependent on the system infrastructure installed and the prototype developed. With proper dimensioning on the system infrastructure, performance is not affected. The environment used also has an impact on performance. There has been a big debate on whether performance on virtual environments is at par with performance on bare metal environments, this is due to the nature of virtualized environments where resources are shared amongst co-located virtual machines plus the performance overhead brought about by the hypervisor. However, there have been several Big Data benchmarks that have been done on virtual environments, in fact, Apache Spark's sorting of a petabyte of data benchmark was conducted on a virtual environment (Xin, 2014)

No severe performance degradations were noticed on our system. The prototype developed will also have an impact on performance based on the logic implemented and the Apache Spark Streaming transformations used.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

5.0. Conclusion

This project set out to determine whether real-time processing of CDRs is feasible. We had to accomplish five objectives in order to answer this question, select a suitable streaming platform for the real-time processing of CDRs, develop a stream processing prototype that processes CDRs, design, install and optimize the selected platform for the real-time processing of CDRs, measure the performance metrics of latency and throughput and resource usage metrics of CPU, disk and memory usage on the installed platform and finally determine that real-time processing of CDRs via the selected stream processing platform is feasible.

We were able to achieve all the five objectives as detailed in this report. For the first objective, we made a feature comparison of five stream processing platforms in use today and ended up settling on Apache Spark Streaming for our project. The second objective was met by installing three clusters of Apache Kafka, Apache Spark Streaming and Apache Cassandra. Apache Kafka was responsible for generating streams of data from an input file, these streams of data were then forwarded to Apache Spark Streaming for processing using a prototype developed in Java and finally, the results were stored in an Apache Cassandra database. We managed to collect the resource usage metrics and performance metrics from the installed system, with performance metrics being collected from Apache Spark Streaming statistics UI, while the resource usage metrics were collected using Graphite and Grafana. The metrics data allowed us to analyze the performance of the system and determine whether real-time processing of CDRs is feasible.

Our experiment results demonstrate that we were able to achieve the targeted throughput of 100,000 records per second with latencies of less than 60 seconds comfortably. We however were not able to obtain reliable sub-second latencies, we managed to attain an average latency of 0.8 seconds for the batch interval setting of 0.5 seconds, and this indicates that our system was struggling to keep up with the input rate. With this result and based on the Oxford dictionary definition of real-time, we are able to state that Apache Spark Streaming is not a suitable candidate for the real-time processing of CDR data, however, with maximum latency of 6.2 seconds across all our batch interval settings, it is a perfect candidate for near-real time

processing of CDR data. Based on (Rouse, 2011) definition of real-time, where real-time refers to a system responsiveness that is perceived by a user to be immediate or nearly immediate, a maximum latency of 6.2 seconds can be construed to be real-time, and this will be contingent on a project's latency requirements. For this research project, the target latency requirements were 60 seconds and the installed system managed to surpass this, this leads us to conclude that Apache Spark Streaming is a suitable candidate for the real-time processing of CDR data. With further optimizations to the clusters of Apache Kafka, Apache Spark Streaming and Apache Cassandra, it is possible to achieve improved throughputs and latencies.

This research project demonstrates that performance depends on several varied factors, this include; the batch interval, the system infrastructure setup and configuration, the data sources and processed data storage or destination platforms, volume of data, record size and the transformations and processing logic of the stream processing application. All these factors need to be put into consideration when implementing an Apache Spark Streaming solution.

In this project, we are able to demonstrate that Apache Spark Streaming can run reliably on virtual machines. It also shows that monitoring and processing of CDR data can be done entirely on open source technologies which are economical. We also expound on how to set up a platform for CDR processing from dimensioning server resources, to installing and integrating the required clusters, developing a stream processing prototype and monitoring the entire platform for performance and resource bottlenecks

5.1. Challenges

One of the challenges we encountered was in the use of the java programming language as the preferred programming language. Apache Spark Streaming is developed using the Scala programming language and therefore there is a lot of in depth documentation and material geared towards Scala. For a person with no Scala programming experience, this can be a bit daunting.

An additional challenge arose due to the fact of the amount of work required to complete the research project. This project required a lot of technologies which are distinct in their implementation mechanism and operation. Integrating all these to perform took a lot of time and required a lot of research.

5.2. Recommendations for Future Work

CDR data contains a lot of valuable information that can be used to provide rich insights into a telecom's subscribers and networks, our project only focuses on resolving customer complaints. However, the combination of Apache Spark Streaming and CDR data provides opportunities for complex processing, especially considering that Apache Spark Streaming is a library within Apache Spark and Apache Spark contains other libraries for graph processing, machine learning and Spark SQL. It would be interesting to see CDR data applied to graph processing and machine learning and the insights that could be derived from such applications.

Also of interest would be to perform a similar experiment using a pure streaming platform such as Apache Flink and comparing the performance.

Due to time constraints, we were not able to develop the user interface for querying CDR data by customer care agents, nevertheless, we managed to query the Apache Cassandra tables for CDR data using an MSISDN information successfully. Other than the user interface, implementing a visualization tool on top of the data contained in Apache Cassandra would be a plus, the visualization tool would be used to monitor the SMS status and error codes, possibly generating an alert when a threshold has been crossed.

As demonstrated in this research project, setting the correct batch interval is crucial to Apache Spark Streaming's stability, however, this is bound to be affected by the ingestion rate. To ensure a stable platform, dynamic batch intervals can be configured such that the batch interval changes dynamically with respect to the ingestion rate, ensuring that processing times are well below the set batch interval.

REFERENCES

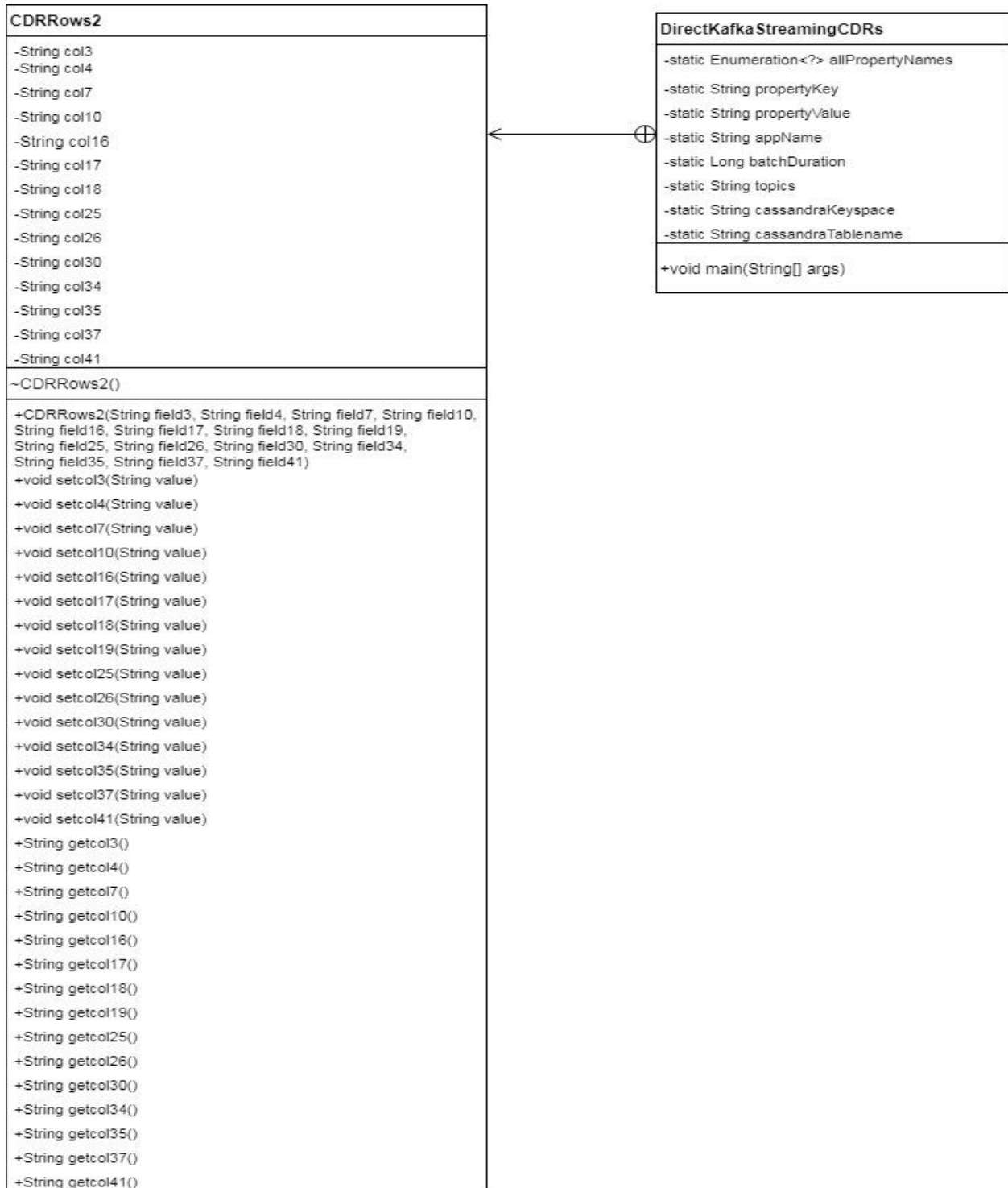
- 3GPP. (2016, June). 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Telecommunication management; Charging management; Charging Data Record (CDR) file format and transfer (Release 13) 3GPP TS 32.297 V13.2.0 (2016-06).
- Agung, M., & Kistijantoro, A. I. (2016). High Performance CDR Processing with MapReduce.
- Asay, M. (2017, February). *An inside look at why Apache Kafka adoption is exploding*. Retrieved from TechRepublic: <https://www.techrepublic.com/article/an-inside-look-at-why-apache-kafka-adoption-is-exploding/>
- Barlow, M. (2013). *Real-Time Big Data Analytics: Emerging Architecture*.
- Bockermann, C. (2014, May 16). A survey of the stream processing landscap.
- Bouillet, E., Kothari, R., Kumar, V., Mignet, L., Nathan, S., Ranganathan, A., . . . Verscheure, O. (2012). Experience report: Processing 6 billion CDRs/day - From research to production.
- Bughin, J. (2016). Reaping the benefits of big data in telecom. *Journal of Big Data*.
- CAK. (2017). *THIRD QUARTER SECTOR STATISTICS REPORT FOR THE FINANCIAL YEAR 2016/2017 (APRIL-JUNE 2017)*.
- Celebi, U. (2016, February). *How Apache Flink™ Enables New Streaming Applications*. Retrieved from Data Artisans: <https://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications>
- Celebi, U. (2016, February). *How Apache Flink™ Enables New Streaming Applications*. Retrieved from <https://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications>
- Cloudera. (2017). *Configuring Kafka for Performance and Resource Management*. Retrieved from https://www.cloudera.com/documentation/kafka/latest/topics/kafka_performance.html
- Confluent. (2017). *What is Apache Kafka?* Retrieved from <https://www.confluent.io/what-is-apache-kafka/>
- d4D. (2014). *Orange Data For Development Challenge in Senegal*.
- dataflair. (2017, April). *Lazy Evaluation in Apache Spark – A quick guide*. Retrieved from <https://data-flair.training/blogs/apache-spark-lazy-evaluation/>
- Datastax. (2015, February). *Basic Rules of Cassandra Data Modeling*. Retrieved from <https://www.datastax.com/dev/blog/basic-rules-of-cassandra-data-modeling>
- Datastax. (2017). *What is Apache Cassandra*. Retrieved from <https://academy.datastax.com/planet-cassandra/what-is-apache-cassandra>
- Dezyre. (2016, 12 24). *Cloudera vs. Hortonworks vs. MapR - Hadoop Distribution Comparison*. Retrieved from <https://www.dezyre.com/article/cloudera-vs-hortonworks-vs-mapr-hadoop-distribution-comparison-/190>
- Dezyre. (2017, March). *Apache Spark Architecture Explained in Detail*. Retrieved from <https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338>
- Evans, R. (2015, June). *Scaling Apache Storm (Hadoop Summit 2015)*. Retrieved from Slideshare: <https://www.slideshare.net/RobertEvans26/scaling-apache-storm-hadoop-summit-2015>

- Flink, A. (2017). *Data Streaming Fault Tolerance*. Retrieved from Apache Flink: https://ci.apache.org/projects/flink/flink-docs-release-1.3/internals/stream_checkpointing.html
- Flink, A. (2017). *Fault Tolerance Guarantees of Data Sources and Sinks*. Retrieved from Apache Flink: <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/connectors/guarantees.html>
- Flink, A. (2017). *File Systems*. Retrieved from <https://ci.apache.org/projects/flink/flink-docs-release-1.3/internals/filesystems.html>
- Flink, A. (2017). *Jobmanager High Availability*. Retrieved from https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/jobmanager_high_availability.html
- Flink, A. (2017, August). *Powered by Flink*. Retrieved from <https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink>
- Flink, A. (2017). *Streaming*. Retrieved from <http://flink.apache.org/features.html>
- Graves, T. (2013). GraySort and MinuteSort at Yahoo on Hadoop 0.23.
- Guller, M. (2015). *Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large Scale Data Analysis*.
- Hadoop, A. (2013, April). *HDFS architecture guide*. Retrieved from Apache Hadoop: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#NameNode+and+DataNodes
- IBM. (2013). *Analytics: Real-world use of big data in telecommunications*.
- IBM. (2017). *HDFS - Apache Hadoop Distributed File System*. Retrieved from IBM Analytics: <https://www.ibm.com/analytics/hadoop/hdfs>
- Itzchakov, Y. (2016, July). *Exploring Stateful Streaming with Apache Spark*. Retrieved from <http://asyncified.io/2016/07/31/exploring-stateful-streaming-with-apache-spark/>
- Jiang, X. (2017, May). *A Year of Blink at Alibaba: Apache Flink in Large Scale Production*. Retrieved from Dataversity: <http://www.dataversity.net/year-blink-alibaba/>
- Kim, S., & Blafford, R. (2016, July 06). *Stream windowing performance analysis: Concord and Spark Streaming*.
- Kuhlenkamp, J., Klems, M., & Ross, O. (2014). *Benchmarking Scalability and Elasticity of Distributed Database Systems*.
- Kx. (2016, July 23). *Telecoms industry: Solutions and Data Profiles*.
- Laskowski, J. (2014). *Dynamic Allocation (of Executors)*. Retrieved from Mastering Apache Spark 2 (Spark 2.2+): [view-source:https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dynamic-allocation.html](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dynamic-allocation.html)
- Laskowski, J. (2017). *BlockManager — Key-Value Store for Blocks*. Retrieved from <https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-blockmanager.html>
- Mayo, M. (2016, March). *Top Big Data Processing Frameworks*.
- Middendorf, B. (1999, September). *MANAGING TELECOM'S CURRENCY: THE CDR*.
- Mishra, K. (2015, January). *How Spotify Scales Apache Storm*. Retrieved from Spotify Labs: <https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/>
- Perera, S. (2012, July). *Consider the Apache Cassandra database*. Retrieved from IBM Developer Works: <https://www.ibm.com/developerworks/library/os-apache-cassandra/index.html>
- Philip, N. (2014, October). *The Key to Success with Big Data Projects (Updated)*. Retrieved from <https://www.qubole.com/blog/big-data-project-success/>

- Rabl, T., Sadoghi, M., & Jacobsen, H.-A. (2012). Solving Big Data Challenges for Enterprise Application Performance Management.
- Rebaca. (2017). Big Data Governance using Kafka-Spark-Cassandra Framework.
- Rouse, M. (2011, January). DEFINITION real-time analytics.
- Spark, A. (2014). *Spark Streaming Programming Guide*. Retrieved from <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- Spark, A. (2017). Retrieved from <https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD>
- Spark, A. (2017). *Lightning-fast cluster computing*. Retrieved from <https://spark.apache.org/>
- Spark, A. (2017). *Project and Product names using "Spark"*. Retrieved from <http://spark.apache.org/powered-by.html>
- Spark, A. (2017). *Spark Streaming Programming Guide*. Retrieved from <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- SQLSTREAM. (2017). *Stream processing- definition*. Retrieved from <http://sqlstream.com/feature/stream-processing-definition/>
- StackExchange. (2014). *Open Data*. Retrieved from StackExchange: <https://opendata.stackexchange.com/questions/1840/download-a-sample-database-of-calls-in-telecommunication>
- Storm, A. (2012, August). *Storm 0.8.0 and Trident released*. Retrieved from <http://storm.apache.org/2012/08/02/storm080-released.html>
- Storm, A. (2015). *Fault Tolerance*. Retrieved from <http://storm.apache.org/releases/current/Fault-tolerance.html>
- Storm, A. (2015). *Guaranteeing Message Processing*. Retrieved from Apache Storm: <http://storm.apache.org/releases/1.1.0/Guaranteeing-message-processing.html>
- Storm, A. (2015). *Project Information*. Retrieved from <http://storm.apache.org/about/scalable.html>
- Storm, A. (2015). *Storm State Management*. Retrieved from Apache Storm: <http://storm.apache.org/releases/1.1.0/State-checkpointing.html>
- Storm, A. (2015). *Trident State*. Retrieved from <http://storm.apache.org/releases/1.1.1/Trident-state.html>
- Storm, A. (2017). *Apache Storm Powered By*. Retrieved from <http://storm.apache.org/releases/current/Powered-By.html>
- Syncsort. (2015, November). *The Difference Between Real Time, Near-Real Time, and Batch Processing in Big Data*. Retrieved from <http://blog.syncsort.com/2015/11/big-data/the-difference-between-real-time-near-real-time-and-batch-processing-in-big-data/>
- Wahner, K. (2014, September 10). Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse.
- Wang, Y. (2016). Stream Processing Systems Benchmark: StreamBench.
- Xin, R. (2014, November). *Apache Spark officially sets a new record in large-scale sorting*. Retrieved from Databricks: <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>
- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., . . . Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.
- Zapletal, P. (2016, Feb 2). Comparison of Apache Stream Processing Frameworks: Part 1.

Zuffer, A. (2017, October). *Apache Flink High Availability for Fault Tolerance in Distributed Systems*. Retrieved from Medium: https://medium.com/@fathima_93409/apache-flink-high-availability-for-fault-tolerance-in-distributed-systems-dcdfcce251d9

APPENDIX 1: CLASS DIAGRAM



APPENDIX 2: FEATURE COMPARISON

Comparison Feature	Apache Flink	Apache Storm (Trident)	Spark Streaming
<p>Fault tolerance and High availability</p>	<p>Apache Flink's fault tolerance mechanism depends on continuously creating distributed snapshots of the distributed streaming data flow and operator state on persistent storage such as HDFS or a database. When a failure occurs, the affected distributed streaming data flow is stopped (a parallel data flow is started to ensure availability), the system restarts the operators and resets them to the last successful checkpoint. Input streams are reset to the current point in the state snapshot. Each snapshot will contain the offset in the stream when the snapshot was started for each stream data source and a pointer of the state that was stored for each operator. (Flink, Data Streaming Fault Tolerance, 2017)</p> <p>This can be problematic for large complex jobs where state can grow over time and become very huge since checkpointing requires to go over all the state data. To ensure high availability, Apache Flink recommends deploying multiple task managers and a standby job manager. The Job manager's responsibility is to assign tasks to the task manager, in case of a task manager failure, the affected work is redistributed to the remaining active task managers (Zuffer, 2017). In case of a Jobmanager failure the standby job manager becomes active. For HA Jobmanager, Apache Zookeeper has to installed, Apache Zookeeper will continually monitor the active</p>	<p>Apache storm architecture consists of two types of nodes, the master node (Nimbus) and the worker nodes (supervisors). Nimbus is responsible for running the stop topology and tracking and assigning tasks to supervisors which execute and keep track of worker processes. If a worker process fails, the supervisor restarts it, if the supervisor fails and nimbus is unable to reach it, tasks assigned to the failed machine are reassigned to active supervisors. Nimbus is stateless, it does not keep topology state internally but requires Apache Zookeeper to maintain state. If Nimbus fails, no new topologies are assigned to supervisors and existing topologies cannot be deactivated or activated. It is restarted by a monitoring service, Nimbus then gathers the meta information from Apache Zookeeper and start keeping track of the supervisors. It is advisable to run Nimbus on high availability, a feature that was introduced in version 1.0.0, this will still depend on Apache Zookeeper for state maintenance and leader election (Storm, Fault Tolerance, 2015). Without a standby Nimbus node, the solitary Nimbus node becomes a single point of failure. Apache Storm Trident checkpoints the state of bolt (processing units) operations periodically in memory and backs it up to Redis (database). A checkpoint has to be committed by the checkpoint spout for the topology checkpoint to be successful, each</p>	<p>The most important concept of fault -tolerance in Apache Spark streaming is the RDD. The main abstraction in Apache Spark Streaming is called a dstream which is a sequence of RDDs. RDDs are immutable data structures that are modified by applying transformations that result in a new RDD. A sequence of transformations create a direct acyclic graphs that stores the transformations and intermediate results during processing. In case of failure, the DAG is reran to recompute the associated RDD. Apache Spark Streaming also supports checkpointing, data that is checkpointed includes metadata information such as configuration, dstream operations and incomplete batches and generated RDDs. Checkpointing is a configurable parameter and not automatically enabled. In case of failure, recovery is initiated using the checkpointed data, driver recovery will utilize the metadata information stored. Apache Spark Streaming also uses Write Ahead Logs (WAL), which save received data to persistent storage, WAL are written to before data is processed by Apache Spark Streaming. Recovery involves replaying the contents of this log. If using a reliable data source such as Apache Kafka or Apache Flume, which send ACKs after data has been successfully replicated, then there is no data loss. Unreliable receivers do not implement the ACK mechanism and therefor are bound to lose or duplicate data. The ACK mechanism is not</p>

	<p>job keeper as well as store checkpoint and job state information. Without a standby jobmanager, the single jobmanager becomes a single point of failure (Flink, Jobmanager High Availability, 2017)</p>	<p>bolt has to send an ACK back to the checkpoint spout, once all bolts have sent their ACKS, the checkpoint is complete (Storm, Storm State Management, 2015). Each tuple (data) fetched from a spout (source) is assigned a unique identifier and passed to a bolt or series of bolts for processing. Once processing is complete, the executing task sends an ACK back to the spout (Storm, Guaranteeing Message Processing, 2015).</p> <p>In case of a supervisor failure and recovery, a bolt is reinitialized to its last known state and affected tuples (data) are replayed. Each tuple is assigned a unique transaction identifier so on tuple replay, no duplicate data is processed (Storm, Guaranteeing Message Processing, 2015)</p>	<p>enabled by default and therefore has to be implemented.</p> <p>The fault recovery mechanism on Apache Spark Streaming therefore depends on checkpointing, Write Ahead Logs and sending ACKs back to the data sources. An Apache Spark Streaming cluster will involve a combination of master nodes and worker nodes. It is advisable to run multiple master nodes for high availability, otherwise the solitary master node becomes a single point of failure, only one node can be leader at any one time. This will require Apache Zookeeper for cluster co-ordination and leader election. Multiple worker nodes are advisable for workload sharing and high availability. The worker nodes host executor processes that actually handle processing tasks, if one executor fails, the other active executors continue operating (Spark, Spark Streaming Programming Guide, 2014). An Apache Spark Streaming cluster also contains a driver process which initiates the SparkStreamingContext, the driver process runs on one single node at a time, it is therefore a single point of failure. This can be mitigated by running the Apache Spark Streaming Cluster in cluster mode using resource managers such as Hadoop Yarn and Apache Mesos, the resource managers will try and restart the driver automatically if it fails. However, driver failures are recovered using the metadata that is checkpointed to the filesystem</p>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Scalability

Apache Flink allows the addition of task managers and job managers to handle increasing workload and/or provide high availability. Addition of task managers can be done seamlessly without the need of restarting the cluster. Addition of a job manager will require the installation and configuration of Apache Zookeeper will ensure the distributed co-ordination via leader election of the running job managers. Only one job manager can be a leader at any one time (Flink, Jobmanager High Availability, 2017).

Apache Flink supports several distributed file systems for data storage such as HDFS and S3, the filesystem provides an acknowledgment that data has been stored once it has been replicated to a configurable quorum of nodes. All nodes should have visibility of the stored data and this is achievable by use of the absolute file path (Flink, File Systems, 2017) We are yet to know how many nodes the largest cluster of Apache Flink can handle, though Alibaba uses Apache Flink for real-time data processing and have a cluster containing "thousands" of nodes (Jiang, 2017). Scalability is linear

Different parts of the Apache Storm trident cluster can be scaled independently by tweaking their parallelism i.e. the addition of more tasks, worker processes and executors (Storm, Project Information, 2015), this however depends on the application being executed. For instance Apache Storm recommends one worker process per node, one executor per CPU core for CPU bound tasks or 1-10 executors per CPU core for I/O bound tasks (Mishra, 2015). Concerning the Apache Storm Trident infrastructure, a cluster can have two Nimbus nodes, but only one can be the leader at any time. The cluster can have several supervisors for workload sharing. Addition of supervisors does not necessarily affect the running cluster, an exception would be where the cluster's resources are over-utilized or constricted, in this case on adding the new nodes, Apache Storm will kill the current workers and start them on all the nodes, redistributed the workload. Apache Storm depends on distributed file systems for data storage, it is known to work with HDFS and S3. Apache storm is linearly scalable with the largest cluster having 2300 nodes (Evans, 2015).

Apache Spark Streaming scales linearly. Scalability can be achieved by adding more executor processes, workers and system resources such as CPU, disk and memory on a running cluster. Apache Spark Streaming has a feature that enables elastic scaling of executors, that is executors are dynamically added to the cluster in response to workload (Laskowski, 2014). Apache Spark Streaming however, recommends running one executor per node. Apache Spark Streaming supports several distributed filesystems such as HDFS and S3. HDFS contains a feature for block replication which replicates data across the nodes with HDFS The largest known cluster for Apache Spark has 8000 nodes (Xin, 2014).

<p>Message delivery guarantees</p>	<p>Apache Flink guarantees exactly once message processing, however, this is dependent on both the data source and data sinks used, for instance, using Apache Kafka as a data source guarantees exactly once semantics while twitter streaming API as a data source can only guarantee at most once semantics. Apache Flink's checkpointing mechanism makes it possible to have exactly once message processing even in the presence of failures (Flink, Fault Tolerance Guarantees of Data Sources and Sinks, 2017).</p>	<p>Apache Storm Trident provides exactly once processing guarantees. It achieves this in three ways</p> <ol style="list-style-type: none"> 1) Each batch is given a unique identifier, in case of tuple replays, the batch will maintain this exact identifier 2) State updates are ordered among batches, and processing of batches is sequential 3) Checkpointing mechanism where state is stored in memory and backed up to persistent storage and the use of record acknowledgement (ACKS) to source spouts <p>ACK mode is configurable though, one can choose not to use ACKs as they are known to affect throughput</p>	<p>Apache Spark Streaming guarantees exactly once message processing with reliable data sources such as Apache Kafka and Apache Flume. This is achieved by</p> <ol style="list-style-type: none"> 1) Sending record acknowledgements back to the source that a record has been received and replicated successfully. 2) Use of write ahead logs to store received data that can be used for recovery in case of failure. <p>The ACK mechanism is not enabled by default and has to be explicitly implemented</p>
<p>Adoption level & project maturity</p>	<p>>32 companies using, including Alibaba, Researchgate, Ericsson and Telecom Portugal</p> <p>34 committers on Apache</p> <p>Graduated to an Apache top level project in January 2015</p> <p>4 libraries still in beta</p> <p>Made 8 software releases in 2016 and 6 (so far) in 2017, last software release was in August, 2017</p>	<p>>80 companies using including Yahoo!, Spotify, The weather, Baidu, Alibaba, Yelp, WebMD</p> <p>35 committers (both Storm core and trident) on Apache</p> <p>Graduated to an Apache top level project in September 2014</p> <p>Flagship release on September, 2011.</p> <p>Made 6 software releases in 2016 and 5 in 2017 (so far), last software release was in September, 2017</p>	<p>>90 companies using including AsialInfo, Yahoo, Nokia Solutions and Networks, Amazon and big industries</p> <p>54 committers on Apache for the entire Spark framework</p> <p>Spark graduated to an Apache top level project in February 2014, Spark streaming (alpha) released on February 2013</p> <p>Made 9 software releases in 2016 and 3 in 2017 (so far), last software release was in October, 2017</p>
<p>Streaming model</p>	<p>Apache Flink provides both Stream processing and batch processing on the same platform. In batch processing, data is treated as a stream of data and is operated on for a bounded period of time i.e. until the whole data file has been processed, the output is one batch of results (Flink, Streaming, 2017). Apache Flink supports iterative processing</p>	<p>Micro-batching using tuples. The inbuilt mechanism for generating batches uses 'tick tuples', a configuration that allows the setting of a frequency at which to receive tick tuples and normal tuples (Storm, Storm 0.8.0 and Trident released, 2012). Apache Storm Trident does not support iterative processing</p>	<p>Micro-batching using dstreams. The inbuilt mechanism for generating batches is the use of the batch interval which can be set as low as 500ms. The batch interval represents the frequency at which batches are produced, with each batch containing zero or more records. Each batch is an RDD, after processing, a batch of results is obtained (Spark, Spark Streaming Programming Guide, 2014). Apache Spark Streaming supports iterative processing</p>

Multiple library support	Apache Flink contains libraries for batch processing and stream processing as mentioned before. Other libraries currently in development are for complex event processing, machine learning and graph processing (Flink, Streaming, 2017).	Apache Storm Trident is a layer on top of Apache Storm that enables stateful, micro-batched stream processing	Apache Spark Streaming is a library on top of Apache Spark which also contains libraries for SQL, machine learning, graph processing and stream processing, all of them in production
Programming language support	Supports both Java and Scala	Apache Storm Trident is designed to be used by any programming languages e.g. Java, Python, Ruby, Perl, Scala (Storm, Project Information, 2015).	Supports Scala, Java, Python
State management	Apache Flink periodically checkpoints the state of operations and data structures holding data, this is stored in memory and backed up to persistent storage. Apache Flink mostly manages the state of the application and also provides an interface through which state can be managed, one can query, update, list or clear the state of an element. Managing state allows Apache Flink to be fault tolerant as it periodically checkpoints the state of the application, keeping track of operations, so that in case of failures, it can reset back to the last successful checkpoint, this also ensures that exactly once processing is maintained (Celebi, How Apache Flink™ Enables New Streaming Applications, 2016).	Apache storm stores operator state internally in memory and backs it up to persistent storage or stores state externally in a database. Apache Storm Trident also provides an interface for querying and updating state. State is important to Apache Storm Trident as it internalizes fault-tolerance logic within a state. It also ensures exactly-once processing semantics (Storm, Trident State, 2015).	Apache Spark Streaming achieves stateful stream processing through checkpointing on persistent storage, this involves configuring a checkpoint directory in an application to store state between batches. Apache Spark Streaming also provides an interface for querying and updating state. Stateful stream processing enables exactly once processing semantics (Itzchakov, 2016).

APPENDIX 3: SOURCE CODE

```
package com.uonbi.dct.streamingcdrs;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.*;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaInputDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.kafka010.ConsumerStrategies;
import org.apache.spark.streaming.kafka010.KafkaUtils;
import org.apache.spark.streaming.kafka010.LocationStrategies;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import com.datastax.spark.connector.japi.CassandraJavaUtil;
import com.datastax.spark.connector.japi.CassandraStreamingJavaUtil;
import java.io.Serializable;

import java.util.*;
import org.apache.hadoop.conf.Configuration;

public class DirectKafkaStreamingCDRs{

    private static Enumeration<?> allPropertyNames;

    private static String propertyKey;
    private static String propertyValue;
    private static String appName;
    private static Long batchDuration;
    private static String topics;
    private static String cassandraKeyspace;
    private static String cassandraTablename;

    public static class CDRRows2 implements Serializable
    {

        private String col3;
        private String col4;
        private String col7;
        private String col10;
        private String col16;
```



```
private String col17;  
private String col18;  
private String col19;  
private String col25;  
private String col26;  
private String col30;  
private String col34;  
private String col35;  
private String col37;  
private String col41;
```

```
CDRRows2() {}
```

```
public CDRRows2(String col3,String col4,String col7,String col10,String col16,String col17,String col18,String  
col19,String col25,String col26,String col30,String col34,String col35,String col37,String col41)
```

```
{  
    this.col3 = col3;  
    this.col4 = col4;  
    this.col7 = col7;  
    this.col10 = col10;  
    this.col16 = col16;  
    this.col17 = col17;  
    this.col18 = col18;  
    this.col19 = col19;  
    this.col25 = col25;  
    this.col26 = col26;  
    this.col30 = col30;  
    this.col34 = col34;  
    this.col35 = col35;  
    this.col37 = col37;  
    this.col41 = col41;
```

```
}
```

```
public void setcol3(String value){col3 = value;};  
public void setcol4(String value){col4 = value;};
```

```
public void setcol7(String value){col7 = value;};  
public void setcol10(String value){col10 = value;};  
public void setcol16(String value){col16 = value;};  
public void setcol17(String value){col17 = value;};  
public void setcol18(String value){col18 = value;};  
public void setcol19(String value){col19 = value;};  
public void setcol25(String value){col25 = value;};  
public void setcol26(String value){col26 = value;};  
public void setcol30(String value){col30 = value;};  
public void setcol34(String value){col34 = value;};
```

```
public void setcol35(String value){col35 = value;};
public void setcol37(String value){col37 = value;};
public void setcol41(String value){col41 = value;};
```

```
public String getcol3(){return col3;};
public String getcol4(){return col4;};
public String getcol7(){return col7;};
public String getcol10(){return col10;};
public String getcol16(){return col16;};
public String getcol17(){return col17;};
public String getcol18(){return col18;};
public String getcol19(){return col19;};
public String getcol25(){return col25;};
public String getcol26(){return col26;};
public String getcol30(){return col30;};
public String getcol34(){return col34;};
public String getcol35(){return col35;};
public String getcol37(){return col37;};
public String getcol41(){return col41;};
```

```
}
```

```
public static void main(String[] args) throws Exception
{
    Map<String, Object> kafkaParams = new HashMap<>();
    SparkConf sparkConf = new SparkConf();
    Configuration hdfsConf = new Configuration();
    FileSystem fs = FileSystem.get(hdfsConf);
    Path consumerPropertiesFilePath = new Path("hdfs:///project/conf/myConsumer.properties");
    FSDataInputStream is = fs.open(consumerPropertiesFilePath);
    Properties myConsumerProperties = new Properties();

    //load properties from inputstream to properties obj
    myConsumerProperties.load(is);

    //retrieve properties
    allPropertyNames = myConsumerProperties.propertyNames();

    while (allPropertyNames.hasMoreElements())
    {
        propertyKey = (String) allPropertyNames.nextElement();

        if(propertyKey.startsWith("app"))
```

```

{
    topics = myConsumerProperties.getProperty("app.topics");
    appName = myConsumerProperties.getProperty("app.appname");
    batchDuration = Long.parseLong(myConsumerProperties.getProperty("app.batchduration"));
    cassandraKeyspace = myConsumerProperties.getProperty("app.cassandra.keyspace");
    cassandraTablename = myConsumerProperties.getProperty("app.cassandra.tablename");
}
else
{
    propertyValue = myConsumerProperties.getProperty(propertyKey);
    kafkaParams.put(propertyKey, propertyValue);
}
}

Set<String> topicsSet = new HashSet<>(Arrays.asList(topics.split(",")));
sparkConf.setAppName(appName);

//Read messages in batch of batchDuration seconds
JavaStreamingContext jssc = new JavaStreamingContext(sparkConf, Durations.seconds(batchDuration));

// Start reading messages from Kafka and get DStream
final JavaInputDStream<ConsumerRecord<String, String>> stream =
    KafkaUtils.createDirectStream(jssc, LocationStrategies.PreferConsistent(),
        ConsumerStrategies.<String,String>Subscribe(topicsSet,kafkaParams));

// Read each line from javainputdstream and return javadstream
JavaDStream<String> lines = stream.map(new Function<ConsumerRecord<String,String>, String>() {
    @Override
    public String call(ConsumerRecord<String, String> kafkaRecord) throws Exception {
        return kafkaRecord.value();
    }
});

JavaDStream<CDRRows2> resultToCassandra = lines.map(new Function<String, CDRRows2>()
{
    private CDRRows2 cdrsContent;
    @Override
    public CDRRows2 call(String arg0) throws Exception
    {
        String[] cols = arg0.trim().split("\\|"); //or arg0.split(Pattern.quote("|")); | special char in regex
        if(cols.length >= 42)
        {
            //retrieve date part from scts
            cols[4]=cols[16].substring(0, 10);

```

```

        cdrsContent = new
CDRRows2(cols[3],cols[4],cols[7],cols[10],cols[16],cols[17],cols[18],cols[19],cols[25],cols[26],cols[30],
        cols[34],cols[35],cols[37],cols[41]);
    }
    return cdrsContent;
}
});

//store records into cassandra using CDRRows case class- mapper class
CassandraStreamingJavaUtil.javaFunctions(resultToCassandra)
    .writerBuilder(cassandraKeyspace,
cassandraTablename,CassandraJavaUtil.mapToRow(CDRRows2.class)).withBatchGroupingKey(CassandraJavaUtil.B
ATCH_GROUPING_KEY_PARTITION).saveToCassandra();

//resultToCassandra.print(100);
jssc.start();
jssc.awaitTermination();
}
}

```