A Comparison of Real Time Stream Processing Frameworks

Jonathan Curtis
Technological University Dublin, Ireland

A Comparison of Real Time Stream Processing Frameworks

Jonathan Curtis

C06038514

A dissertation submitted in partial fulfilment of the requirements of
Dublin Institute of Technology for the degree of
M.Sc. in Computing (Advanced Software Development)

2018

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: __________________________________

Date: **04 March 2018**

ABSTRACT

The need to process the ever-expanding volumes of information being generated daily in the modern world is driving radical changes in traditional data analysis techniques. As a result of this, a number of open source tools for handling real-time data streams has become available in recent years. Four, in particular, have gained significant traction: Apache Flink, Apache Samza, Apache Spark and Apache Storm. Despite the rising popularity of these frameworks, however, there are few studies that analyse their performance in terms of important metrics, such as throughput and latency. This study aims to correct this, by running several benchmarks against these frameworks.

Key words: *Stream Processing, Apache Flink, Apache Kafka, Apache Samza, Apache Spark, Apache Storm, Latency, Throughput*

ACKNOWLEDGEMENTS

I would like to thank my supervisor Andrea Curley for all her help. I would also like to thank my family for supporting me through this process.

Contents

Table of Figures

Table of Tables

1 INTRODUCTION

1.1 Background

Huge repositories of data, oftentimes terrabytes in size, are generated daily by modern information systems and digital technologies (Acharjya & P, 2016, p. 511). We now live in an age where data is growing in orders of magnitudes greater than ever before. By 2020, the amount of data on the planet is expected to reach 44 zettabytes (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 2). Furthermore, many individual companies are now processing quantities of data that, only a few years ago, seemed unimaginable. Alibaba, for example, generates tens of terabytes daily in online trading transactions, while Google processes hundreds of petabytes per month (Sakr, 2017, p. 34).

The need to process these ever-expanding volumes of information is driving radical changes in traditional data analysis techniques. For many use cases, conventional batch processing, which is oftentimes based on the MapReduce programming model, is no longer sufficient, with businesses now requiring that information be processed in real-time. This is because, for many applications, the value of data may decrease in proportion to the time that has passed since it was produced (Maarala, Rautiainen, Salmi, Pirttikangas & Riekki, 2015: 2855). As a result of this, a variety of open source tools for handling real-time data streams has become available in recent years.

Of the myriad frameworks available, four, in particular, have gained significant traction: Apache Flink, Apache Samza, Apache Spark and Apache Storm. While these tools fundamentally perform the same function, in that they process incoming streamed data, each takes a different approach to realise its goal. As a result of this, each has its own strengths and weaknesses. These tend to be directly correlated to the processing model used by the individual framework. Two approaches are taken in general: the micro-batching model and the record-by record model. The mirco-batching model tends to favour higher throughput, while the record-by-record model benefits from reduced processing latency.

1.2 Research Project/Problem

Latency, according to Chandramouli et al, is a significant user metric in many commercial real-time applications (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 255). Low latency, they argue, is important for avoiding stale decisions. High throughput, meanwhile, is important for ensuring that messages get processed in a timely manner. While both throughput and latency are of the utmost concern in real-time data stream processing systems, there are few studies that provide definitive metrics with regards to the performance of the numerous platforms available for handling streamed data. Hesse and Lorenz support this assertion, arguing that, in relation to throughput and latency, "there are no numbers that… create a clear ranking for these quantifiable attributes" (Hesse & Lorenz, 2015, p. 800).

Qian et al also argue that choosing a streaming platform is incredibly difficult. This is because of “the diversity of choices and the complexity of configurations for each platform” (Qian, Wu, Huang, & Das, 2016, p. 592). A comprehensive reference on how to choose a streaming platform, they argue, is still unavailable. In their work, focusing on Spark, Storm and Samza, they attempt to address this problem. They do so by performing a number of evaluations on their chosen stream processing frameworks. These evaluations, however, are done using specific version of the frameworks. Given the ever-increasing popularity of stream processing, many of the systems have undergone significant changes since Qian et al’s study was published. For example, Spark went from from version 1.6.1 to version 2.1.1 in a little over one year. The 2.0.0 release brought significant changes to the framework.

Pääkkönen acknowledges that framework releases can potentially result in significant performance improvements. He found that in previous benchmarking studies, the processing latency of Spark executing a word counting benchmark was between 2-3 seconds. However, the reported word count latency in his paper was “on a sub-second” range. The difference, he argues, could potentially be explained with the newer version of Spark, coupled with the different dataset. For reference, he used v1.3.1 in his study, whilst the other benchmarks used version v0.9. Interestingly, none of the benchmarking works examined for this study used Spark version 2.0.0 or greater. Thus, there exists a need to test the latest versions of the various open source Apache stream processing projects, to determine their current performance with regards to throughput and latency.

The specific research question that will form the basis of this work is as follows:

Does Apache Spark perform better than Apache Flink, Apache Samza, and Apache Storm in terms of throughput and latency when processing streamed data?

1.3 Research Objectives

The primary objective of this study is to evaluate the performance of Spark in relation to Flink, Samza, and Storm. The performance evaluation will be conducted in terms of two metrics: throughput and latency. These values will be obtained across a multitude of different scenarios, each of which will consist of common stream processing operations. In this way, it is hoped that the scope of the results will be wide enough to provide individuals and organisations who are looking to implement a streaming system with meaningful insights that will, ultimately, help inform their decision as to which framework to deploy.

1.4 Research Methodologies

Statistics related to the throughput and latency of each of the frameworks will be gathered as they execute various benchmarking scenarios. The configuration and implementation details from these scenarios will be presented, along with a qualitative analysis of the collected statistics. The mean and standard deviation will be calculated from these statistics for each framework/scenario combination. Various statistical tests will then be used to either confirm or refute the aforementioned research question.

1.5 Scope and Limitations

Nowadays, many Big Data applications are facilitated by Infrastructure as a Service (IaaS) cloud computing vendors, such as Google, Amazon, Microsoft (Li, Wu, Jiang, Li, & Wei, 2017, p. 792). In their 2014 work, Jiamin and Jun discuss the use of various Infrastructure as a Service (IaaS) cloud computing vendors in independent research studies that focus on the evaluation of software platforms (Jiamin & Jun, 2014, p. 152).

They posit that it is unrealistic to expect individual research groups – and even small enterprises – to purchase large numbers of commodity hardware for the purposes of academic or temporary analysis of platforms (Jiamin & Jun, 2014, p. 152). Thus, they argue that the use of cloud computing vendors is a reasonable expectation and compromise in such studies, especially when considering that many providers offer research grants that enable researchers to build up large-scale clusters free of charge (Jiamin & Jun, 2014, p. 152).

With the above said, it is important to highlight the limitations of such an approach to evaluating software platforms. Focusing on Amazon Web Services in particular - where various virtual machines are provided to end users via Amazon's Elastic Compute Cloud service - Jiamin and Jun highlight the fact that there is significantly more variance with regards to performance than is evident when physical clusters are used for testing. This variance, they argue, is partially the result of different types of processors being used on the hardware within the cloud centres (for example, AMD Opteron is 35% slower than the Intel Xeon) and it is – at least at the time their study was published – impossible for an end-user to choose the processor type in advance (Jiamin & Jun, 2014, p. 152). Furthermore, network performance can vary based on the geographical location of the cloud centre, particularly when the centre is in a different time zone, as it may be affected by the local working time and other such factors.

Chandramouli et al argue that capturing reliable metrics, particularly latency, is a difficult task. This, they argue, is because of the complexity of "dataflow plan[s] and the non-trivial interactions between the components of a distributed system" (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 256). Thus, a novel approach for capturing latency and throughput will need to be employed. This will be built upon knowledge garnered from the review of other benchmarking works. By taking this approach, it might transpire that the results collected in this study cannot be easily compared to the results found in other works.

Finally, the relative newness of the field of stream processing presents numerous challenges for this work. Firstly, many of the frameworks under evaluation are undergoing significant development as attempts are made to adapt to newly emerging requirements. Therefore, there is a risk that a new version for one or more of the frameworks will be released during the period that this study is conducted. This, quite evidently, could result in the findings of this study being outdated. As such, it is important for the readers of this work to note that the results of the evaluation are only valid for the versions of the frameworks outlined below. Furthermore - and also related to the issue of newness - some of the systems being tested are quite immature (in particular Samza). Qian, Wu, Huang and Das, for example, had difficulty when it came to benchmarking Samza; specifically, they were not able to effectively test Samza's latency. This, they argued, was due to the immaturity of the platform (Qian, Wu, Huang, & Das, 2016, p. 594).

1.6 Document Outline

This study will be structured as follows:

- Chapter 2 examines the existing literature surrounding stream processing frameworks.
- Chapter 3 outlines the design of a benchmarking suite that forms the basis of the various experiments that will be conducted in this study. It also discusses the approach that will be used for capturing the latency and throughput of the frameworks.
- Chapter 4 discusses the implementation of the experiments and the results collected during their running.
- Chapter 5 analyses the results from the experiments and determines whether Spark performs better than Flink, Samza, and Storm in terms of throughput and latency.

2 LITERATURE REVIEW

This chapter examines the existing literature surrounding stream processing frameworks. As distributed systems have a natural overhead associated with their use that makes them unsuited for all but the largest of processing tasks, it makes sense to begin the discussion by analysing and defining the concept of Big Data. Following on from this, traditional forms of data processing are discussed. A significant proportion of the chapter is then dedicated to the streaming platforms that form the basis of this study. This is because it is important for readers to be able to distinguish between the different processing models and terminology used by each framework. After this, the metrics of interest to the study will be discussed briefly. Finally, the chapter concludes with a review of existing benchmarking studies.

2.1 Big Data

Landset et al argue that the term Big Data is still the subject of disagreement. They posit that it "generally refers to data that is too big or too complex to process on a single machine" (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 2). Acharjya and P support this assertion (Acharjya & P, 2016, p. 511). They argue that there is no exact definition for Big Data. Rather, they suggest that it is "problem-specific" (Acharjya & P, 2016, p. 511). As such, the more widely accepted definitions of Big Data, Landset, Khoshgoftaar, Richter and Hasanin argue, tend to define it in terms of the challenges it presents. These challenges - oftentimes referred to as the "big data problem" - were first characterised in 2001 and fall into three categories: **volume**, **velocity** and **variety** (commonly referred to as the 3Vs) (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 3). In the years since this initial characterisation, additional aspects have been identified.

Hu, Wen, Chua and Li use a "4V" approach for distinguishing between Big Data and traditional data. These Vs are: volume, velocity, variety and value (Hu, Wen, Chua, & Li, 2014, p. 654). Volume refers to the sheer size of datasets. As an example, Hu et al point to Facebook, which reported 2.7 billon "likes" and comments registered per day by users in 2012. Velocity relates to the speed at which data is generated and collected (Hu, Wen, Chua, & Li, 2014, p. 654). Examples of such systems are radio-frequency identification data management applications, which can generate massive quantities of data every second. Variety refers to the format of data (Hu, Wen, Chua, & Li, 2014, p. 654). There are, Hu, Wen, Chua and Li posit, three types of data: structured, semi-structured and unstructured (Hu, Wen, Chua, & Li, 2014, p. 654). Traditional data is typically structured, whereas Big Data, such as user-generated content on YouTube and Twitter, is, for the most part, unstructured (Hu, Wen, Chua, & Li, 2014, p. 654). The final V is value. Big data tends to have a low value density, but, by employing a variety of mining methods, significant value can be derived when large enough datasets are analysed (Hu, Wen, Chua, & Li, 2014, p. 654).

Acharjya and P also use a 4V approach for defining Big Data (Acharjya & P, 2016, p. 511). Their Vs, however, differ slightly from Hu, Wen, Chua and Li's. While they agree on the original 3Vs - volume, velocity, and variety - as well as their definitions, they assert that the final V is veracity, not value (Acharjya & P, 2016, p. 511). Veracity, they argue, relates to the accountability of the data; essentially how trustworthy (or untrustworthy) the data is. Unlike both Hu, Wen, Chua and Li, and Acharjya and P, Landset, Khoshgoftaar, Richter and Hasanin are sceptical that that the addition of a fourth V – either value or veracity – adds to the overall understanding of Big Data (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 4). As such, they only focus on the original 3Vs in their 2015 work. This study will also take the same approach. The term Big Data, as used in this study, will refer to data whose challenges conform to the original 3Vs.

2.2 Traditional Data Processing

Traditional data management and analysis applications tend to use a *process-after-store* model for computing data and are usually built upon relational database management systems (RDBMSs) (Hu, Wen, Chua, & Li, 2014, p. 653; Stonebraker, Çetintemel, & Zdonik, 2005, p. 45). Under this model, incoming data is first persisted in the RDBMs - where it is potentially indexed – before being processed by the analysis application (Stonebraker, Çetintemel, & Zdonik, 2005, p. 45). Given the emergence of Big Data and its accompanying challenges, coupled with the need for real-time data handling from multiple services, these types of systems are no longer viable for many organisations.

The disconnect between the traditional approach and the emerging Big Data paradigm can be classified into two aspects: firstly, in terms of data structure, RDBMSs can only support structured data. When it comes to semi-structured or unstructured data, these systems offer little to no support (Hu, Wen, Chua, & Li, 2014, p. 653). Secondly, from the perspective of scalability, RDBMSs can only be scaled up with expensive hardware. Given the rates at which data production is growing, RDBMSs are simply not able to cope with the ever-increasing volumes of data being generated in today's world (Hu, Wen, Chua, & Li, 2014, p. 653). As a result of these limitations, new technologies began emerging that attempted to account for the challenges of Big Data.

2.2.1 Hadoop

The most established software platform for traditional Big Data analysis, according to Acharjya and P, is Apache Hadoop (Acharjya & P, 2016, p. 511). This assertion is also supported by Huang, Meng, Zhang, and Zhang (Huang, Meng, Zhang, & Zhang, 2017, p. 4). Hadoop was initially introduced in 2007 as an open source implementation of the MapReduce processing engine, which itself was introduced in 2004 by Google engineers (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 5, 9). MapReduce is a programming model for processing large datasets via a batch processing mechanism (Acharjya & P, 2016, p. 511). This model, evident in the name, is implemented in two steps, the map step and the reduce step. The map stage involves dividing the input into smaller subsets and distributing it to the various nodes in the cluster, while the reduce step involves merging the output from the processing of the nodes into a single dataset (Acharjya & P, 2016, p. 511).

Hadoop (and MapReduce), according to Patel, Sakaria and Bhadane, is not suitable for the ad hoc data exploration and real-time analytics required of stream processing frameworks (Patel, Sakaria & Bhadane, 2015, p. 50). This, they argue, is because Hadoop simply wasn't built for real-time processing (Patel, Sakaria & Bhadane, 2015, p. 50). This is evident in the fact that, with MapReduce, queries can take anywhere from several seconds to many hours - or even days - to complete.

Since its inception, Hadoop has evolved into a "vast web" of projects related to every step of a Big Data workflow (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 5).

The amount of projects that have been developed to either complement or replace the original elements of Hadoop has, thus, made a current definition of the project unclear (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 5). As a result of this, many people use the term *Hadoop Ecosystem* when referring to the project, instead of just Hadoop. The project currently consists of four core modules. The Common module contains a set of utilities needed by the other Hadoop modules. The MapReduce module, as mentioned, is the core data processing engine used by Hadoop. The Hadoop Distributed File System (HDFS) module is a file system designed to store large amounts of data across multiple nodes of commodity hardware.

The final component within the Hadoop ecosystem is YARN (Yet Another Resource Manager). Prior to Hadoop version 2.0, Hadoop and MapReduce were tightly coupled, with MapReduce responsible for both cluster resource management and data processing (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 5). YARN has now taken over these responsibilities and, by doing so, has improved upon many of the deficiencies present in the old MapReduce. YARN, for example, is able to run on larger clusters, more than doubling the amount of jobs and tasks it can handle before running into bottlenecks. It also allows for a more generalized Hadoop which makes MapReduce just one type of YARN application. This means it can be left out entirely in favour of a different processing engine (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 5). As such, YARN is used in conjunction with a number of the stream-processing frameworks under evaluation in this study. It is for this reason that we will discuss YARN in more depth in the next section.

2.2.2 YARN

YARN, as mentioned, forms the “architectural epicentre” of Hadoop (Patel, Sakaria & Bhadane, 2015, p. 50). It is primarily responsible for tasks such as job scheduling and the management of cluster resources (Hesse and Lorenz, 2015, p. 800). A central component of a YARN setup is the Resource Manager, which is realised as a daemon process running on a dedicated machine (Hesse and Lorenz, 2015, p. 800). The Resource Manager monitors the status of nodes within the cluster. It is also responsible for resource distribution amongst applications (Hesse and Lorenz, 2015, p. 800; Huang, Meng, Zhang, & Zhang, 2017, 4). It allocates resources in the form of “containers.” A container, which is usually a UNIX process, can be seen as a “logical bundle of resources” bound to a particular node (Hesse and Lorenz, 2015, p. 800; Huang, Meng, Zhang, & Zhang, 2017, 4).

Another component within YARN is the Node Manager (Hesse and Lorenz, 2015, p. 800; Huang, Meng, Zhang, & Zhang, 2017, 4). A Node Manager daemon runs on each node within the cluster. The task of the Node Manager is to keep track of the node’s resources, notifying the Resource Manager about failures. This communication is implemented using a heartbeat system. Node Managers may also communicate with each other, again via a heartbeat system. This message exchange, however, can only occur at the application level, specifically between so called Application Masters and their assigned containers (Hesse and Lorenz, 2015, p. 800; Huang, Meng, Zhang, & Zhang, 2017, 4).

Application Masters are created by the Resource Manager once an application has been accepted. The Application Master handles various aspects of program execution, including resource needs management and fault handling. YARN delegates the scheduling of applications to each individual Application Master (Huang, Meng, Zhang, & Zhang, 2017, 4). In order to acquire resources, an Application Master has to send a request to the Resource Manager. Prior to sending this request, the Application Master creates a logical execution plan for the application. Once the Resource Manager has allocated various resources, the Application Master is responsible for generating a physical execution plan based off the assigned resources (Huang, Meng, Zhang, & Zhang, 2017, 4). As soon as a resource lease on behalf of an Application Master is created, the corresponding container is pulled by the Application Master’s next heartbeat (Huang, Meng, Zhang, & Zhang, 2017, 4).

Huang, Meng, Zhang, and Zhang acknowledge that the centralised resource manager approach taken by YARN could potentially be viewed as a bottleneck in a Big Data system (Huang, Meng, Zhang, & Zhang, 2017, 4). This, they argue, is because Application Masters may often require large numbers of containers to process their data, and the Masters themselves have limited capacity when it comes to scheduling large amounts of tasks. However, many “sophisticated works,” they continue, have been proposed and implemented within YARN that ultimately enables it to overcome a number of limitations with regards to this approach.

2.3 Streaming and Stream-Processing

Stream-processing systems operate on continuous data streams (Kambatla, Kollias, Kumar & Grama, 2014, p. 2568). Hesse and Lorenzo define a data stream as a "continuous flow of incoming data records, comparable to a data feed" (Hesse & Lorenz, 2015, p. 797). The data within a stream, for example, could be sensor data, network data, stock prices or postings in social networks, to name just a few. Unlike with a traditional data analytics platform, incoming data is not usually persisted before query execution (Hesse & Lorenz, 2015, p. 797). As such, the stream processing model has an impact on the types of queries executed. Typically, so called continuous queries are used within a streaming system (Hesse & Lorenz, 2015, p. 797). As the name suggests, these queries are continuously executed using a dataset. The datasets, against which the queries are executed, are defined by a trigger function. This function specifies when query processing should occur. Generally, queries are executed based on a time interval or the number of newly approached data tuples (Hesse & Lorenz, 2015, p. 797).

At a high level, streaming applications can be represented by directed acyclic graphs (DAGs), where the vertices, called either processing elements (PEs) or tasks, represent operators, and the edges/links - the streams - represent the data flow from one processing element or task to the next (Anis Uddin Nasir, De Francisci Morales, Garcia-Soriano, Kourtellis, & Serafini, 2015, p. 137; Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 255; Li, Wu, Jiang, Li, & Wei, 2017, p. 784). Each task performs a predefined function on the data from an input stream (or streams) and then emits data to an output stream. A task without any incoming link is called a source, and one without any outgoing link is called a sink (Li, Wu, Jiang, Li, & Wei, 2017, p. 784). There can be multiple sources and sinks in a streaming application.

Hu, Wen, Chua and Li argue that under the stream processing paradigm, the value of data depends on data freshness (Hu, Wen, Chua and Li, 2014, p. 656). This assertion is supported by Hesse and Lorenzo, who argue that, as streaming systems tend to execute queries on "sliding windows" of data - where only the newest data is considered - it is generally easier to discover emerging trends, than if a database table was being evaluated, such as in a traditional data analytics platform (Hesse & Lorenz, 2015, p. 797). The window boundaries can be either time-based or count-based (De Matteis & Mencagli, 2016, p. 303).

There are, as mentioned in Chapter 1, two approaches to stream processing: micro-batching and record-by-record processing. With micro-batching, the stream is treated as a sequence of "small batch chunks of data" (Shahrivari, 2014, 123). Essentially, at specified intervals, the incoming data is grouped into single collections and sent for processing. With the record-by-record model, each incoming message is handled as a standalone event.

2.4 Streaming Architecture

Most traditional streaming systems, according to Patel, Sakaria and Bhadane, involve an "update and pass" methodology that handles a single record at a time (Patel, Sakaria & Bhadane, 2015, p. 57). There are two approaches for fault tolerance in these types of systems: replication, which tends to be fast, but entails significant hardware costs, and upstream backup/buffered records, which, in comparison to replication, is relatively slow but is significantly less expensive with regards to hardware costs (approximately half the price, in some cases) (Patel, Sakaria & Bhadane, 2015, p. 57). Importantly, neither of these approaches scales efficiently in large clusters that consist of hundreds or, even, thousands of nodes.

2.4.1 Scaling

Scaling refers to the ability of a system to adapt to increased data processing demands (Singh & Reddy, 2014, p. 3). In general, there are two approaches to scaling: horizontal scaling and vertical scaling. Horizontal scaling involves distributing the workload across ever-increasing numbers of servers. These servers are, oftentimes, "commodity" machines; machines that are relatively inexpensive to acquire (Singh & Reddy, 2014, p. 3). Vertical scaling, on the other hand, refers to the act of installing more processors, more memory and generally faster hardware within a single machine, with the goal of increasing processing power.

Both approaches to scaling offer numerous benefits and drawbacks. Vertical scaling, for example, can be quite costly initially. Single machines, given their nature, are limited by the number of expansion slots available. As such, users will need to buy hardware that is not only powerful enough to cover current needs, but also powerful enough to cover any foreseeable future needs (Singh & Reddy, 2014, p. 3). In contrast, horizontal scaling provides users with the ability to increase performance in small increments. This ultimately lowers upfront costs. There is also no limit to the amount that a system can be scaled out; that is, depending, of course, on the software being used for data processing (Singh & Reddy, 2014, p. 3). With that said, managing and upgrading a single machine is a relatively straightforward endeavour. Also, in a lot of cases, the power provided by current generation hardware might be sufficient for an organisation's data processing needs.

2.4.2 Lambda Architecture

Attempts have been made to merge the more traditional batch processing approach to Big Data analytics with the increasingly popular field of stream processing (Heidrich, Trendowicz, & Ebert, 2016, p. 113; Pathirage, Hyde, Pan, & Plale, 2016, p. 1627). Presented as a software design pattern, the Lambda Architecture unifies online and batch processing within a single framework (Kiran, Murphy, Monga, Dugan, & Baveja, 2015, p. 2786; Pathirage, Hyde, Pan, & Plale, 2016, p. 1627). Within the lambda architecture, there are three layers that each has a specific responsibility. The first layer, called the batch layer, is dedicated to analysing persisted data. Tools such as Hadoop or Spark are used at this stage. The second layer, called the speed layer, handles stream processing (Heidrich, Trendowicz, & Ebert, 2016, p. 113). Tools such as those being evaluated in this study are used at this layer. Finally, the last layer, called the serving layer, is responsible for responding to queries (Kiran, Murphy, Monga, Dugan, & Baveja, 2015, p. 2786).

Pathirage et al argue that there are a number of flaws with the Lambda Architecture. One of these relates to the fact that two codebases must be maintained within Lambda Architecture systems, the one that deals with the batch layer and the one that deals with the speed layer. Maintaining code that needs to produce the same result in two complex distributed systems is, they argue, "exactly as painful as it seems like it would be"

(Pathirage, Hyde, Pan, & Plale, 2016, p. 1627). Kiran et al also admit that the approach has shortcomings. Setting up jobs to execute and produce results using multiple projects, they argue, requires more skill from developers than when using simply one layer (Kiran, Murphy, Monga, Dugan, & Baveja, 2015, p. 2786). Pathirage et al suggest that the same results can be achieved using only stream processing. This is accomplished by "retaining the input data unchanged… and reprocessing the data through increased parallelism" (Pathirage, Hyde, Pan, & Plale, 2016, p. 1627). This unified model, which benefits from requiring the management of only a single codebase, is commonly referred to as the Kappa Architecture.

2.4.3 Streaming Challenges

One of the main challenges for streaming systems relates to the ordering of incoming data tuples (Hesse & Lorenz, 2015, p. 797). While there is a natural order based on the arrival time of the data values to the streaming system, this order does not necessarily correspond to the chronological order of the "value occurrences," or the creation time of the raw data (Hesse & Lorenz, 2015, p. 797). Due to varying factors, such as, for example, network latency or connection outages, the data sequence of incoming values could be upset, in relation to their creation time, with the potential for some values being lost entirely en route to the streaming system. If the occurrence of such scenarios is particularly high, the stream representing the source of data could be referred to as a "noisy" data stream (Hesse & Lorenz, 2015, p. 797). In such cases, a trade-off must be made for certain scenarios between performance and correctness, as it is possible to wait for missing values or, alternatively, execute queries with missing data (Hesse & Lorenz, 2015, p. 798).

2.5 Streaming Platforms

2.5.1 Apache Kafka

Wang, Liu, and Zhou describe Kafka as "a distributed, partitioned, replicated commit log service" that combines the "benefits of traditional log aggregators and messaging systems" (Wang, Liu, & Zhou, 2016, p. 364). Yadranjiaghdam, Yasrobi, and Tabrizi, on the other hand, define Kafka as a "distributed streaming platform that uses publishsubscribe messaging and is developed to be a distributed, partitioned, replicated service." Kafka's architecture is, according to Hesse and Lorenz, "comparatively simple," as it consists only of a set of Brokers (Hesse and Lorenz, 2015, p. 801). A Broker is essentially a server that has an instance of Kafka running on it. A Kafka cluster can be comprised of one or more Brokers.

Data streams with Kafka are defined by topics, which are divided into partitions that are distributed over Broker instances (Hesse and Lorenz, 2015, p. 801). Each partition holds a number of records. A record consists of a key, a value, and a timestamp (Yadranjiaghdam, Yasrobi, & Tabrizi, 2017, p. 331). Partitions, thus, are essentially sequences of records which are ordered and immutable, and may be continually appended to a commit log.

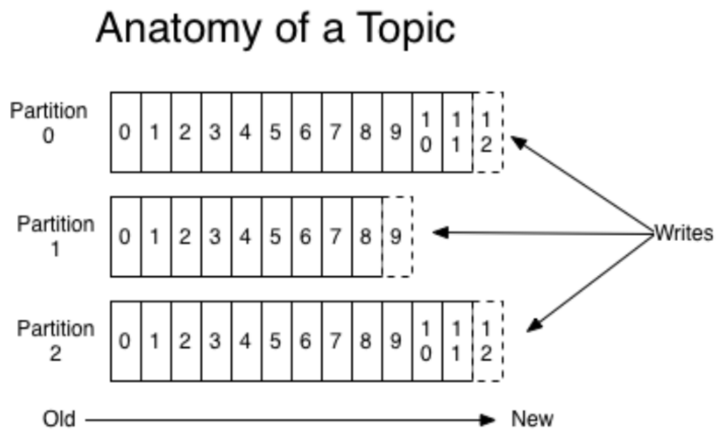


Figure 1. *Anatomy of a Topic*[1].

In terms of terminology within Kafka, the process that publishes messages to a topic is called a producer, while the process that subscribes to a topic to consume messages is called a consumer (Wang, Liu, & Zhou, 2016, p. 364). Kafka provides guaranteed message delivery with proper ordering, so messages sent by a producer to a particular topic will be delivered in the order they are sent (Solaimani, Iftekhar, Khan, Thuraisingham, & Ingram, 2014, p. 1). Topics are also multi-subscriber. As such, they

[1] See https://kafka.apache.org/intro

may have zero, one or many consumers who can access the data contained within (Yadranjiaghdam, Yasrobi, & Tabrizi, 2017, p. 331).

Kafka retains all published records, without consideration for their consumption. However, as Kafka's performance is effectively constant with respect to data size, storing data for a long time does not affect overall system performance (Yadranjiaghdam, Yasrobi, & Tabrizi, 2017, p. 331). With that said, Yadranjiaghdam, Yasrobi and Tabrizi suggest disabling this feature when persistence of records is not required, in order to avoid access latency issues with hard disks (Yadranjiaghdam, Yasrobi, & Tabrizi, 2017, p. 331).

2.5.2 Apache Flink

Flink's roots can be found in an open-source big data analytics research project called Stratosphere that was developed at the Technical University of Berlin (Hesse & Lorenz, 2015, p. 799; Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 15; Sakr, 2017, p. 39). In January, 2015, Flink became a top-level Apache project. Similar to Spark, Flink supports both batch processing and stream processing. Unlike Spark, the Flink streaming API is based on individual events, rather than the micro-batch approach that Spark takes (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 16). Various connectors are offered that allow for processing data streams from, among others, Kafka and RabbitMQ.

Flink is mainly written in Java and Scala, and offers client APIs for both languages (Hesse & Lorenz, 2015, p. 799). Flink's runtime makes use of a master-worker pattern (similar to Storm). There are two node types within a Flink cluster: the Job Manager (master) and one or more Task Manager(s) (the workers) (Hesse & Lorenz, 2015, p. 799). The Job Manager is the interface to client applications. It receives assignments from clients and, subsequently, schedules the work of Task Managers. It is also responsible for keeping track of the overall execution of jobs, as well as the state of every worker. It accomplishes this last task using a heartbeat mechanism.

Each Task Manager provides a certain number of processing slots to the cluster (Hesse & Lorenz, 2015, p. 799). These slots are used for parallelising tasks. The number of slots provided is configurable, but, according to, Hesse and Lorenzo, it is recommended to use as many slots as there are CPU cores in each Task Manager node. The degree to which a job is parallelised can be defined in multiple ways (Hesse & Lorenz, 2015, p. 799).

Flink's processing model applies transformations to parallel data collections (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 15). Such transformations, Landset, Khoshgoftaar, Richter and Hasanin argue, generalise map and reduce functions, as well as various other functions, such as join, group, and iterate . Flink also includes a cost-based optimiser which automatically selects the best execution strategy for each job (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 15).

Landset, Khoshgoftaar, Richter and Hasanin discuss various comparisons between Flink and Spark in their work (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 16). While Spark, in one comparison, was found to be superior in the areas of fault tolerance and the handling of iterative algorithms, Flink's advantages, they argue, were the presence of an optimisation mechanism and better integration with other projects (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 16). Flink, however used more resources, but was able to finish jobs in less time than Spark. It must be noted, however, that Landset, Khoshgoftaar, Richter and Hasanin do qualify their statement by saying that Flink, at the time of the study, was quite immature, and had undergone significant changes since their study was published that may affect the results of any new tests (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 16).

Hesse and Lorenzo claim that, with regard to performance, Flink is considered to have very low latency, but high throughput (Hesse & Lorenz, 2015, p. 799).

2.5.3 Apache Samza

Apache Samza is a distributed, "lightweight" stream-processing engine that was initially developed at LinkedIn before being open-sourced in 2013 (Feng, Zhuang, Pan, & Ramachandra, 2015, p. 2601; Riccomini, 2013; Zhuang, Feng, Pan, Ramachandra & Sridharan, 2016, p. 268). Samza is mainly written in Scala and Java (Hesse & Lorenz, 2015, p. 800). In a typical Samza setup, two other Apache projects are used: YARN and Kafka. YARN provides Samza's execution environment while Kafka provides the streaming layer. The architecture of Samza, Qian, Wu, Huang, and Das argue, makes it "simple and pluggable" (Qian, Wu, Huang, & Das, 2016, p. 592).

Samza's unit of deployment is a YARN job that can contain "hundreds of streaming tasks, all executing the same processing logic on messages from one or more stream partitions belonging to one or more streams" (Pathirage, Hyde, Pan, & Plale, 2016, p. 1628). Samza's stream primitive is a message rather than a tuple (Morshed, Rana, & Milrad, 2016, p. 1483). Samza divides streams into partitions. An individual partition represents an ordered sequence of read-only messages with distinctive ids. Samza uses a single thread internally to handle reading and writing messages, flushing metrics, checkpointing, windowing, etc.. (Feng, Zhuang, Pan, & Ramachandra, 2015, p. 2601). The Samza container creates an associated task to process the messages of each input stream topic partition.

Samza does not natively support DAG stream topologies, as Storm does (Pathirage, Hyde, Pan, & Plale, 2016, p. 1628). Instead, Samza's architecture encourages the formation of directed acyclic graphs through the connecting of multiple Samza jobs via intermediate Kafka streams. With this approach, a failure of a downstream job will not affect an upstream job that is producing data; it further facilitates sharing across stream processing stages, by allowing the addition of jobs that consume an intermediate stream (Pathirage, Hyde, Pan, & Plale, 2016, p. 1628).

Samza provides developers with a message serialisation and deserialisation API called Serde to support different message formats, such as JSON and AVRO (Pathirage, Hyde, Pan, & Plale, 2016, p. 1628).

Hesse and Lorenz claim Samza has "a relatively high throughput as well as a somewhat increased latency compared to Storm" (Hesse & Lorenz, 2015, p. 800).

2.5.4 Apache Spark

Spark began life as a research project at University of California Berkley in 2009 before becoming an Apache incubated project in 2010 (Acharjya & P, 2016, p. 516; Hesse and Lorenz, 2015, p. 799; Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 13; Sakr, 2017, p. 37). Spark is based on MapReduce, but addresses a number of MapReduces deficiencies (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 13). Particularly, it improves on speed and resource issues by utilising in-memory computation. The Spark project is mainly written in Java, Scala and Python, and provides APIs for each of these languages, as well as R (Acharjya & P, 2016, p. 516; Hesse and Lorenz, 2015, p. 799).

A core pattern used within Spark is the resilient distributed dataset (RDD) (Acharjya & P, 2016, p. 516; Hesse and Lorenz, 2015, p. 799; Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 13). Hesse and Lorenz define an RDD as a “read-only collection of Java or Python objects partitioned across a cluster” (Hesse and Lorenz, 2015, p. 799). In 2015, the RDD API was extended to Include DataFrames, which allow users to group a distributed collection of data by column, similar to a relational database table (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 13).

For covering special scenarios, Spark supports several libraries that are built on top of it, such as Spark SQL and Spark Streaming, which provides stream-processing functionality to the core Spark project and will be the focus of this study. These libraries extend Spark’s core package and most of the data structures used within are based on the same central data structure (RDD) as Spark’s core batch processing framework (Patel, Sakaria & Bhadane, 2015, p. 59).

Spark’s architecture consists of a Driver Program, a Cluster Manager and Worker Nodes (Acharjya & P, 2016, p. 516). The Driver Program represents the application that is executed on top of Spark (Hesse and Lorenz, 2015, p. 799). While there may be numerous applications distributed within a Spark cluster, only the main program creates a SparkContext. This object is responsible for coordinating all of the existing client processes. The Driver Program is typically connected to a Cluster Manager (Acharjya & P, 2016, p. 516; Hesse and Lorenz, 2015, p. 799). There are currently three cluster managers supported by Spark: the Spark Standalone manager, the Apache Mesos manager, and the Apache Hadoop YARN manager (Hesse and Lorenz, 2015, p. 799). The main purpose of the Cluster Manager is to provide executors to applications as soon as a SparkContext has established a connection.

Spark’s Worker Nodes are responsible for performing the calculations required as part of the Driver Program (Hesse and Lorenz, 2015, p. 799). One or more executors can run on a Worker Node and each executor, in turn, can contain one or more tasks (Acharjya & P, 2016, p. 516; Hesse and Lorenz, 2015, p. 799). Executor processes only work for one program at one time, and stay alive until the program has finished (Hesse and Lorenz, 2015, p. 799). This provides complexity reduction with regards to task scheduling, as each Driver Program can schedule the tasks of its exclusive executors independently. A downside to this approach, however, according to Hesse and Lorenz, is that data exchange between different Driver Programs can only be accomplished through "indirections," such as by writing data to a file system or a database (Hesse and Lorenz, 2015, p. 800).

In terms of stream processing, as mentioned, Spark Streaming builds upon Spark's core framework. Data streams represent the input for Spark Streaming, which, in turn, creates micro-batches out of the stream in the form of RDDs. These batches are then passed to the Spark Engine for processing (Hesse and Lorenz, 2015, p. 800). Spark also uses an abstraction for data streams called discretized streams (or D-Streams, for short). D-Streams consist of RDD sequences, whereby each RDD contains data pertaining to a particular stream interval. This approach, Hesse and Lorenz claim, was taken to provide better handling for faults and slow nodes within a cluster (Hesse and Lorenz, 2015, p. 800).

Calculations are structured as "a set of short, stateless, deterministic tasks instead of continuous, stateful operators" (Hesse and Lorenz, 2015, p. 800). These calculations, in the form of small batch processes, allow for the earlier identification of the above mentioned issues, and provides for exactly-once message processing. However, this redundancy, according to Hesse and Lorenz, comes at a cost, in terms of latency. With that said, Spark's throughput, they argue, is "comparatively" higher than other stream processing frameworks (Hesse and Lorenz, 2015, p. 800).

2.5.5 Apache Storm

Storm is a "real-time, fault-tolerant and distributed" stream processing framework that is mainly written in Java and Closure (Hesse & Lorenz, 2015, p. 798). It was originally developed by Nathan Marz and team at BackType (Hesse & Lorenz, Hesse & Lorenz, 2015, p. 798.; Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 14). In 2011, Twitter acquired BackType and the project was subsequently open-sourced the following year. To this day, Storm is still used to power various systems at Twitter (Patel, Sakaria & Bhadane, 2015, p. 60).

A storm cluster contains three node types. The first of these is the master node (called Nimbus, after the daemon that it is running on). The master node receives topologies from connecting clients and manages their execution (Acharjya & P, 2016, p. 516; Hesse & Lorenz, 2015, p. 798). This includes the distribution and scheduling of execution, as well as monitoring overall progress with regards to throughput. Regardless of the size of the cluster, the master node is always fixed (Patel, Sakaria & Bhadane, 2015, p. 52).

Next to the master node, Storm also contains a cluster of Apache Zookeeper nodes (Hesse & Lorenz, 2015, p. 798). Zookeeper, according to Hesse and Lorenz, can be seen as a service for coordinating processes within distributed applications. It essentially acts as a “transmitter of communication” between the other two node types (Hesse & Lorenz, 2015, p. 798). Zookeeper is also the only one of the node types within Storm that maintains any type of state. When a failed instance is restarted, processing continues from the last saved state, which is stored in and retrieved from Zookeeper (Hesse & Lorenz, 2015, p. 799).

The final node type within the Storm architecture is the worker node, also known as supervisor node (Acharjya & P, 2016, p. 516; Hesse & Lorenz, 2015, p. 798). Many supervisors can exist within a Storm cluster. The task of the supervisor is to spawn worker processes based on instructions received from the master node. The supervisor manages the status of the workers using a heartbeat mechanism (Hesse & Lorenz, 2015, p. 798). If a worker process terminates unexpectedly, the Supervisor can restart the process. The status of supervisors is also managed via a heartbeat mechanism. Each supervisor periodically sends a heartbeat, as well as information about free resources, to the master node to indicate that it is still operational.

Persistent queries in the context of Storm are called topologies (Hesse & Lorenz, 2015, p. 798). Hesse and Lorenzo define a topology as a “directed graph where the vertices represent computation and the edges represent data flow between computation components” (Hesse & Lorenz, 2015, p. 798). There are two distinct types of vertices:

spouts and **bolts**. A spout represents a source of data tuples that is used within the topology. A spout could, for example, be responsible for pulling messages from a message queue (Hesse & Lorenz, 2015, p. 798). In contrast to spouts, bolts contain most of the computation logic and are responsible for processing incoming data tuples.

Storm guarantees that each unit of data (tuple) will be fully processed at least once (Patel, Sakaria & Bhadane, 2015, p. 51). Spouts will keep messages in their output queues until the bolts acknowledge them (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 15). Messages will continue to be sent out until they are acknowledged, at which point they will then be dropped out of the queue (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 15).

Scaling storm is achieved by adding additional nodes to the cluster when more processing power is required (Patel, Sakaria & Bhadane, 2015, p. 52). In general, a single machine is used for development and in a development environment, while a collection of 3 - 5 nodes running in production can be considered a medium to large Storm cluster (Patel, Sakaria & Bhadane, 2015, p. 52).

Patel, Sakaria and Bhadane claim Storm can process “one million 100 byte messages” per second on each node in a Storm cluster (Patel, Sakaria & Bhadane, 2015, p. 50). It must be noted that they do not clarify what this processing entails. Hesse and Lorenz claim Storm “usually has a lower throughput as well as a lower latency” when compared to Spark Streaming (Hesse & Lorenz, 2015, p. 799). This, they qualify, depends on the versions and the algorithms used during comparison (Hesse & Lorenz, 2015, p. 799).

2.6 Benchmarking

Definitions of benchmarking, according to Anand and Kodali, can vary significantly (Anand, & Kodali, 2008, p. 258). Key themes within the literature include measurement, comparison, identification of best practices, implementation and improvement.

Of particular concern to this study, are *latency* and *throughput*.

2.6.1 Latency

In their 2011 work, Chandramouli et al posit that latency within a distributed event processing system (EPS) - equivalent to a stream processing system - denotes “the delay that is introduced by the EPS from the point of event arrival to result generation” (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 257). Unfortunately, they do not readily clarify what is meant by “result generation.” They argue that, in event-based models, each dataflow consists of a DAG of operators. Each operator “consumes events from one or more input queues, performs computation, and produces new events to be output or placed on the input queue of other operators.” Confusion arises as to whether “result generation” represents the end of computation or the stage at which the data has been output.

Landset et al seem to support the latter interpretation. They argue that latency can be defined as “the amount of time between starting a job and getting initial results” (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 10). Thus, under this interpretation, we can assume that latency represents the time from when a record arrives to the stage at which it has been output to, for example, a message queue. De Matteis and Mencagli also seem to support this. They define latency as “the time elapsed from the reception of a tuple triggering the operator internal processing logic and the delivering of the corresponding result” (De Matteis & Mencagli, 2016, p. 303).

Hesse and Lorenzo, on the other hand, seem to support the former interpretation. They define latency as “the time difference between consuming a message and the end of its processing” (Hesse & Lorenz, 2015, p. 798). Qian et al also support this assertion. They define latency as the “average time span from the arrival of a record till the end of processing this record” (Qian, Wu, Huang, & Das, 2016, p. 594).

The author of this study would tend to agree with both Hesse and Lorenzo’s and Qian, Wu, Huang, and Das’ definitions. Assuming latency represents the time difference between record arrival and the delivery of results introduces a layer of uncertainty into benchmarking studies. This is because the work becomes reliant on the performance of external systems. What if, for example, the output sink were a database and the database server were running slowly? This would adversely affect the results of the benchmarking test. However, for this study - as will be discussed in the Design and Methodology section - there is a need to rely on external systems, specifically a messaging system. So, with that said, the following definition of latency is applicable to this study is:

- *Latency is the time difference between the arrival of a record into the relevant messaging queue and the end of its processing within the framework under test.*

2.6.2 Throughput

There are a multitude of definitions available across the literature for throughput in the context of stream processing. Similar to latency, the deviation across these definitions tends to be somewhat pronounced. At a conceptual level, Landset et al define throughput as “the amount of work done over a given time period” (Landset, Khoshgoftaar, Richter & Hasanin, 2015, p. 10). Throughput in this regard, they argue, can be thought of as a measure of efficiency. They do not, however, elaborate on what exactly the term “amount of work done” encompasses. Nor do they give any indication of what a valid “time period” might be.

De Matteis and Mencagli provide a more grounded definition when they argue that throughput is “the number of results delivered per time unit” (De Matteis & Mencagli, 2016, p. 303). They focus on the value element of streaming frameworks with their definition, as value cannot be derived until results have been delivered. While, arguably, this may be true in a production environment, this definition is not ideal for a benchmarking study. Given that there are a multitude of possibilities for handling data that has gone through a stream processing framework (some of which will be discussed in the coming Design and Implementation section), if we were to take De Matteis and Mencagli’s definition literally, we would need to test each of these options in order to provide a complete picture of how each system interacts with the external systems to which there results are delivered. It is also worth mentioning that, as with Landset et al, De Matteis and Mencagli are also vague with regards to what constitutes a valid time period.

Qian et al provide a more refined definition in their work. They define throughput as representing “data size in terms of bytes processed per second” (Qian, Wu, Huang, & Das, 2016, p. 594). While this definition is certainly more focused, it is questionable as to whether there is truly any value in measuring the number of bytes processed per second. It is the individual messages arriving within a stream processing system that hold value, so the throughput should measure the processing of these messages, not the bytes that constitute the them. Given that messages may vary drastically in size, a throughput of 100MB/s could potentially represent the processing of only a single record. This assertion is supported by Hesse and Lorenz, who define throughput as "the number of external input messages that are completed per time unit" (Hesse & Lorenz, 2015, p. 798).

While Hesse and Lorenz's definition goes some way towards covering the issue of value, it still has several shortcomings, most notable of which is: what is meant by the term *completed*? Does it relate to when processing ends? Or is it when the results become available, as with De Matteis and Mencagli's definition? This author would argue that For the purposes of this study, it is a combination of the various definitions observed in the literature that will be form the basis of a definition of throughput. That definition is as follows:

- *Throughput is the number of external input messages that are processed per second.*

2.6.3 Existing Comparative Benchmarking Studies

Hesse and Lorenzo present a "conceptual survey" of various data stream processing frameworks in their 2015 work (Hesse and Lorenzo, 2015, p. 797). In this survey, a multitude of elements, or "dimensions," are considered during the comparison, including: the general architecture of each system, the languages the systems are written in, message treatment, and, of interest to this study, throughput and latency. However, in terms of throughput and latency, Hesse and Lorenzo do not provide any definitive metrics or information related to performed tests. This, they argue, is because "there are no benchmarks or measurements that compare all systems with each other" (Hesse and Lorenzo, 2015, p. 801). As such, they provide a general rating scale, consisting of *low* and *high*, when mentioning the performance of one framework in relation to another. They qualify this scoring system by citing other works in which the frameworks were tested independently (Hesse and Lorenzo, 2015, p. 797). They end their study by actively calling for work in the area of benchmarks for stream processing frameworks.

Several benchmarking systems have been created or proposed in recent years for the purposes of evaluating stream processing frameworks. One such system is the BiCEP benchmark suite (Mendes, Bizzaro, & Marques, 2013, p. 307). Mendes et al argue in their work that the performance requirements of the application domains to which stream processing frameworks are applied makes it “virtually impossible for a single benchmark, with a single metric” to be representative of the entire spectrum of applications. For this reason, they continue, BiCEP is composed of a number of “smaller, domain-specific” benchmarks, each with its own workload, dataset and metrics. One such example is the *Pairs* benchmark, which “exercises a wide range of features commonly found in most… processing application.” These features include: filtering, aggregation, detection of event patterns, etc… It is worth mentioning that Mendes et al do not disclose to which frameworks BiCEP has been applied.

StreamBench, created by Lu, Wu, Xie, and Hu, is another benchmarking system (Lu, Wu, Xie, & Hu, 2014, p. 69). StreamBench leverages a message system as a data feed that, in turn, streams data at a consistent rate into the stream processing framework under test (Lu, Wu, Xie, & Hu, 2014, p. 69). StreamBench consists of seven benchmarking programs that cover both basic operations, as well as common real-world use cases. Some of the programs contained within StreamBench include: *Wordcount*, a simple application that, as the name suggests, counts the number of words in a stream; *DistinctCount*, an application that “first extracts a target field from the record, puts it in a set containing all the words seen and outputs the size of the set”; and *Grep*, an application that searches for the presence of specific patterns in a stream (Lu, Wu, Xie, & Hu, 2014, p. 74).

Lu, Wu, Xie, and Hu applied StreamBench to Apache Spark and Apache Storm (versions 0.9.0-incubating and 0.9.1-incubating, respectively), and found that Spark had greater throughput than Strom, but also significantly higher latency (Lu, Wu, Xie, & Hu, 2014, p. 75). For example, in the *Grep* test, Spark’s throughput was around 20MB/s, whereas Storm’s was approximately 3.5MB/s. On the other hand, Storm’s latency for that same test was 5ms per data record versus 500ms for Spark (Lu, Wu, Xie, & Hu, 2014, p. 75.).

While Lu, Wu, Xie, and Hu's work produced very valid results in the tests they ran, it must be noted that, of the 7 benchmarking programs within StreamBench, only one has a non-text data type (the *Statistics* test that processes numeric values) (Lu, Wu, Xie, & Hu, 2014, p. 72).

In their 2016 work, Qian, Wu, Huang and Das "modify and extend" the definition of StreamBench and subsequently apply it, as mentioned previously, to Spark, Storm and Samza (Qian, Wu, Huang, & Das, 2016, p. 593). They took this approach, of modifying and extending it, because StreamBench, they argue, is not "mature enough" (Qian, Wu, Huang, & Das, 2016, p. 593). In their setup, they use Apache Kafka as a message broker to mediate between message generation and consumption. Interestingly, they use only two data sources to simulate "real-world cases" (Qian, Wu, Huang, & Das, 2016, p. 593). These are, firstly, web logs and, secondly, network traffic data, both of which cover text and numeric data. Overall, 7 benchmark programs are used in their experiments. The first of these is called *Identity* and serves as a baseline for other benchmarks. *Identity* does not perform any operation on the data stream; it merely takes an input stream and outputs that same stream. The rest of the tests involve basic operations, such as counting the number of distinct words in a stream and searching a stream for the occurrence of a particular string sequence.

In their experiments, Qian, Wu, Huang and Das used a six-node, two-cluster setup to test the various frameworks they were evaluating. One of the clusters consisted of a three Kafka nodes, while the final three nodes were responsible for handling the stream computation (Qian, Wu, Huang, & Das, 2016, p. 594). They found that the throughput of Spark was significantly higher than that of Storm (nearly 1,800 MB/s vs 100MB/s), and quite a bit faster than Samza (that was around 1000MB/s). Storm's latency, on the other hand, was far less than that of Spark's (less than 100ms vs around 25s in some scenarios). They did not collect any latency metrics for Samza.

The next chapter of this study outlines the design of a benchmarking suite that seeks to not only fill in the gaps identified in the analysis of other existing benchmarking works, but one that also seeks to answer the research question outlined in Chapter 1.

3 DESIGN AND METHODOLOGY

This chapter outlines the design of a benchmarking suite that forms the basis of the various experiments that will be conducted in this study. The aim of these experiments is to capture the latency and throughput of the frameworks outlined in Chapter 1 when executing common stream processing operations. This gathered information will then be used to answer the research question that forms the centre of this study. The chapter begins by discussing the environment upon which the experiments will be executed. It then presents the proposed architecture of the benchmarking suite and discusses the ways in which the latency and throughput will be captured. The chapter concludes by discussing the benchmarking applications that will form the individual experiments.

3.1 Experiment Environment

The experiment environment that will be used in this study is provided by Amazon Web Services (AWS). AWS is the "umbrella term" for the numerous remote services that make up Amazon's cloud-computing offering (Axelrod, 2015, p. 27). Cloud computing, for reference, can be defined as:

> "…a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Serrano, Gallardo, & Hernantes, 2015, p.30).

AWS was chosen because it is the "dominant player" in cloud computing, having been the first company to offer cloud services, back in 2006 (Serrano, Gallardo, & Hernantes, 2015, p.33). In their 2015 work, Serrano et al found that AWS had over 54% reported usage amongst interviewed companies. This is significantly greater than the 6% figure for Microsoft's Azure offering, or the 4% for Google Cloud. In 2017, AWS had a market share of 47.1%, with the nearest competitor being Microsoft Azure at 10.0%, followed by Google Cloud at 3.95% (Coles, 2018). As such, using AWS makes sense, as it most mirrors the production environment to which these frameworks will be deployed.

The specific service that will be used for the experiments in this work is the Amazon Elastic Compute Cloud, known as EC2. EC2 is a web service that provides "secure, resizable compute capacity in the cloud[2]" Essentially, it allows users to spin up virtual machines of varying sizes and purposes. There a number of benefits to using a cloud-based platform like EC2. In particular, compute nodes (called instances in EC2) can be started with the same software configuration, including installed tools, directories and users (Axelrod, 2015, p.27). This is because instances can be launched as direct clones of an existing virtual machine (otherwise known as an Amazon Machine Image in AWS). This not only saves time, it also reduces the likelihood of human error, such as misconfiguration, impacting the veracity of results.

There are a number of different instance types available on EC2. These are grouped firstly by "Family" and then by "Type." Family refers to the use case of the instance. There are, among others, the following categories within the family grouping: General Purpose, GPU Compute, Memory Optimised, and Storage Optimised. Within the individual family groups, there are a number of types. In General Purpose, for example, the types available range from the t2.nano option – with 1 virtual CPU and 500MB of RAM – to the m5.24xlarge – with 96 virtual CPUs and 384GB RAM. Each instance type is charged on a per hour basis. In the US West (Oregon) region, for example, the t2.nano is charged at a rate of $0.0058 per hour, while the m5.24xlarge is $4.608 per hour. For this study, the general purpose t2.large instance type will be used, running Ubuntu Server 16.04 LTS. This offering comes with 2 vCPUs and 8GB RAM (at a cost of $0.0928 per hour).

3.2 Proposed Architecture

The proposed architecture for this study can be seen in figure 2. This diagram presents a high-level overview of the setup. Kafka is being used to generate the streams of data that will be consumed by the frameworks during the benchmark experiments. To conform to best practices, the architecture consists of two separate clusters, one for running Kafka and one for running the stream processing frameworks. Each framework has its own set of dependencies, such as, for example, YARN. These are not indicated on the diagram. The frameworks are displayed in figure 2 as communicating with the

[2] See https://aws.amazon.com/ec2/

Kafka cluster as a whole, rather than with individual nodes. This is because Kafka consumers and producers (which, in this case, are the individual stream processing framework nodes) make use of all brokers in a cluster automatically. This architecture was chosen for several reasons, each of which will be discussed in the following paragraphs.

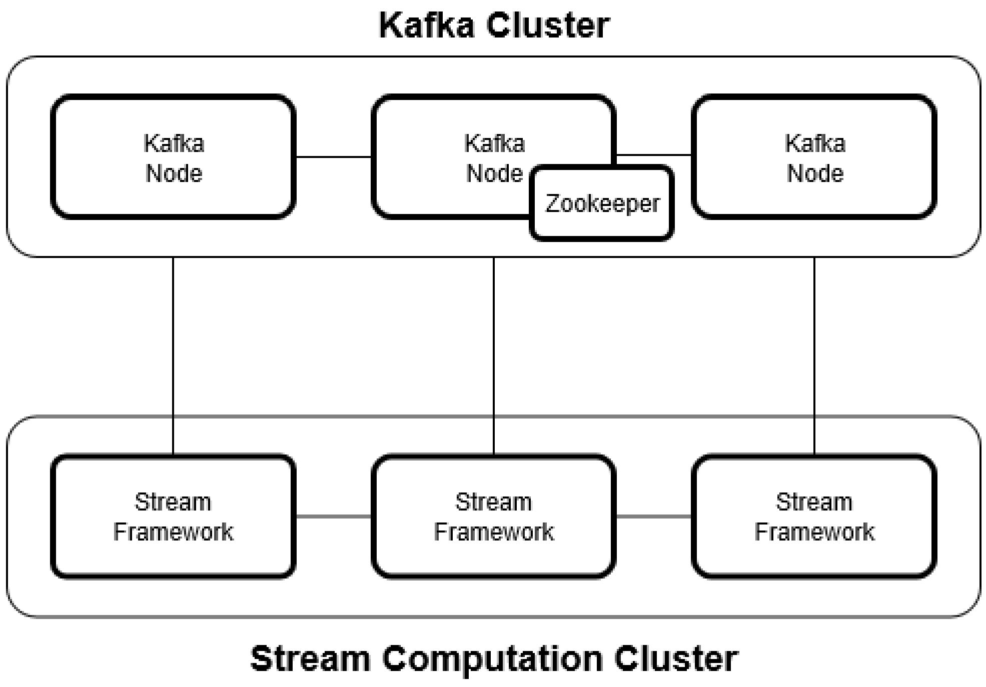


Figure 2. The proposed architecture for this study.

In their 2017 work, Imai et al present a "commonly used" stream processing environment that works for a multitude of frameworks, including those being tested in this study (Imai, Patterson, & Varela, 2017, p. 505). Figure 3 below details this setup. Data flows from left to right, starting with the data producer and ending in a data store. The producer sends events at a rate of λ, which are subsequently appended to message queues in Kafka. The consumer (in this case, a stream processing system) pulls data from Kafka at a throughput of τ (Imai, Patterson, & Varela, 2017, p. 505). After the consumer processes events, it optionally stores results in the data store. It could also, potentially, emit the records back to a Kafka queue for further processing downstream. In this setup, there can be multiple producers and consumers working simultaneously.

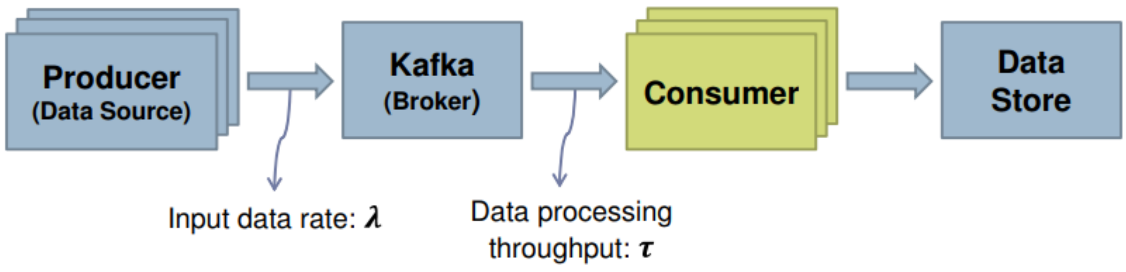

Producer
(Data Source)
Kafka
(Broker)
Consumer
Data Store
Input data rate: λ
Data processing throughput: τ

Figure 3. Common stream processing environment (Imai, Patterson, & Varela, 2017, p. 505).

Lu et al, in their 2014 work, also use a message system as a mediator for data generation in an architecture similar to that proposed by Imai et al (Lu, Wu, Xie, & Hu, 2014, p. 73). They justify this decision for two reasons. Firstly, this design, they argue, provides for a higher level of abstraction and decouples data generation from data consumption "for better feasibility" (Lu, Wu, Xie, & Hu, 2014, p. 73). Secondly, they posit that this approach is the actual usage pattern of many companies that deploy stream processing frameworks. For these reasons, it is this architecture that will be used for this study. It is important to note that there are some factors that should be considered when utilising this architecture, specifically: it is possible that the messaging system could become a bottleneck if data consumption speed is greater than data production speed; and the messaging system may have a negative impact on overall cluster performance.

In order to minimise the impact that the messaging system might have on the results of benchmarking tests, Qian, Wu, Huang, and Das suggest that data preparation should be done "offline." Offline data preparation, they argue, involves generating the streaming data first, and then using fetch and process steps to retrieve that data (Qian, Wu, Huang, & Das, 2016, p. 594). In this way, full network utilisation among the cluster can be achieved and the impact of the message system on computational performance is minimal (Qian, Wu, Huang, & Das, 2016, p. 594). For this study, the experiment environment will consist of two separate clusters. The first cluster will be a Kafka cluster, responsible for data/message generation, while the second will be a compute cluster that is responsible for executing the various streaming operations required for the benchmarks.

In terms of data production, Wang, Liu, and Zhou suggest that a single Kafka producer can reach speeds of 114MB/s (Wang, Liu, & Zhou, 2016, p. 364). To achieve these speeds, some fine tuning of Kafka’s configuration must be made. They find that two values in particular have significant effect on a producer’s throughput. The first of these is the producer.type value, which specifies whether messages are sent synchronously or asynchronously. When the value is set to synchronous, the throughput of a producer is 20MB/s. This increases to 100MB/s when set to asynchronously (Wang, Liu, & Zhou, 2016, p. 364). The second value is acks. This refers to the number of acknowledgements that the producer must receive before considering a request complete. Wang, Liu, and Zhou find that setting acks to 0 results in the highest throughput and greatest stability (Wang, Liu, & Zhou, 2016, p. 364). For this study, acks will be set to 0. With regards to producer.type, in the current version of Kafka, the send method of Kafka producers is asynchronous by default[3].

In terms of the number of EC2 instances that will be used for the experiments, six in total, split into two separate three-node clusters, should suffice. The first cluster will be dedicated to running Kafka (with one node also running Apache Zookeeper for coordination amongst the cluster), while the second will be responsible for stream computation. This setup is similar to the one used by Qian et al in their 2016 work (though, granted, the hardware used in their study is more powerful than the EC2 instances used here) (Qian, Wu, Huang, & Das, 2016, p. 595).

3.3 Metrics Gathering

For capturing the metrics relevant to this study, there are several different approaches that could potentially be taken. The first of these involves using the frameworks’ in-built metric-tracking functionality. Flink, for example, provides a mechanism that allows for the capturing and exposing of various metrics to external systems[4]. This is accessed via the RichFunction interface in Flink’s API. A number of metric types are supported. These are represented by the Counter, Gauge, Histogram and Meter interfaces. Of interest to this study is the Meter interface, which is used to measure the average

[3] See https://kafka.apache.org/10/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer .html

[4] See https://ci.apache.org/projects/flink/flink-docs-master/monitoring/metrics.html

throughput of a job. This is accomplished by registering occurrences of events with the relevant concrete Meter implementation. With regards to latency, a configuration property – latencyTrackingInterval – can be enabled in Flink's execution configuration that results in Flink periodically issuing a special record called a LatencyMarker.

While utilising each framework's included metrics-gathering functionality would, undeniably, simplify the process of capturing the required metrics, there are a number of limitations to this approach that, ultimately, make it an unviable option for this study. Firstly, each framework's implementation of metrics-tracking will differ. When measuring latency, Flink's LatencyMarker, for example, does not account for the time records spend in operators as they are bypassing them (Flink, 2018). The subtle differences between each framework in this regard, therefore, makes a fair comparison across them, when using internal systems, virtually impossible.

The second issue relates to resource utilisation. There is an overheard associated with using in-built metrics-tracking functionality. As the main goal of benchmarking is to produce a common reference for the evaluation of the tools under test, it is imperative that this study avoids adding any undue strain to the frameworks, where possible. To this end, the metrics should be gathered independently of the system and, where possible, the metrics-gathering functionality of the framework under test should be disabled, to prevent any resources being utilised by the framework, which would, consequentially, have a negative impact on the performance of the system. This, therefore, leads us to a second possible approach, one that involves using timestamps.

3.2.1 Capturing Latency

To calculate latency in a distributed event processing system, Chandramouli et al suggest using a "deterministic stimulus time." Essentially, each arriving event is assigned a new field, called stimulus time, which represents the "wall-clock" time of its arrival into the event-processing system (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 257). As the event progresses along the DAG path that represents the dataflow being executed at that time, multiple intermediate events may be produced. However, the stimulus time of an event *e* is the maximum timestamp across all source events in the lineage of *e*.

In conjunction with stimulus time, egress time must also be captured. Egress time represents the wall-clock time when the event exits the event processing system. Thus, for each output event *e*, its latency is the difference between the event's egress time and its stimulus time (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 257).

With the above in mind, coupled with the definition of latency used in this study – which states that latency is the time difference between the arrival of a message and the end of its processing - for this work, a timestamp will be added to each incoming event as it is appended to the Kafka topic. This timestamp will represent the stimulus time. Ideally, the timestamp should be added when the message is consumed by the framework under test. But, given the nature of this study, in which different computational processing models are compared, such an approach would not provide a true representation of latency. This is because of Spark's computational model.

In micro-batching streaming processing systems, the framework groups records into batches at predefined intervals (possible Spark settings and how they affect performance will be discussed in the next chapter). If the batching interval, for example, is set to two seconds, a timestamp can only be programmatically appended to a record once the spark RDD containing that record has been created, which will occur every two seconds. But that record may have been sitting in a Kafka topic for up to potentially two seconds prior to the creation of the RDD. Thus, to accurately measure latency, we need to factor in the time that it has been sitting in the queue waiting for processing.

Appending a timestamp to a Kafka record is a relatively straightforward procedure. In fact, each record already contains made a timestamp, along with a key and value. This timestamp, however, is, by default, the time the message was generated by the producer. Changing this value to the time the record is appended to a topic requires modifying a single configuration parameter. Specifically, log.message.timestamp.type on the various brokers that make up the Kafka cluster needs to be set to LogAppendTime (the default value is CreateTime, which, as mentioned, is the time the message was generated by the producer).

With stimulus time now covered, another timestamp, representing the egress time, will need to be added. This will be done as the record exits the test. The latency for that particular event will then be calculated as the final egress time minus the maximum stimulus time. It is worth mentioning here that the egress time will be a standard Java Date API value - rather than a value unique to each of the individual frameworks. Therefore, the overhead associated with its creation should not have any negative impact on the results of this study, as it will be common across all platforms.

In order to calculate the latency for each system, the results from the processing of individual messages will need to be output for analysis (which, as specified, will consist of subtracting the stimulus time from the egress time). There are multiple approaches that could be taken for outputting the data, such as writing to a database, a distributed cache, or even a local log file. Yahoo, for example, created a stream benchmarking tool, called Yahoo Streaming Benchmarks[5], that uses Redis, an in-memory data store, to keep track of metrics, before writing the final results to a .txt file on a local filesystem. One downside of this tool, however, is that the frameworks are responsible for the computation involved in producing the collected metrics, which, as discussed above, is not ideal.

For this study, the tools already available to hand will be leveraged to calculate latency, specifically Kafka. A new Kafka topic will be created, to which each stream computation node will publish its processed records. A custom-built lightweight Kafka Streams application, called metrics-writer, will then read each message from the topic, calculate the latency for that particular message, and output its results to a file in comma-separated format for subsequent manual analysis and visualisation[6].

3.2.2 Capturing Throughput

Throughput - as has already been defined - is the number of external input messages that are processed per second. A message can be considered processed, in the context of this work, once it reaches the stage at which the egress timestamp has been appended (as this represents the end of processing for that record within the stream framework under test).

[5] See https://github.com/yahoo/streaming-benchmarks

[6] See https://kafka.apache.org/10/documentation/streams/

As described in the previous section, all of the processed records will be written to an output file in comma-separated format. As such, measuring throughput is a relatively straightforward process. Essentially, starting at the very earliest stimulus time in the result dataset (as this represents the time that the experiment began), the records will be grouped into intervals of one second based on their egress time from this initial start period. A simple Java application was written to accomplish this goal. It takes, as parameters, the location of the result dataset and an initial start time. It then groups and counts the records as required.

Before proceeding to discuss the benchmarking applications that will be used in this study, it is worth mentioning that there is a caveat with taking the approaches mentioned here for capturing latency and throughput. When using timestamps within a distributed system, it is important, as Chandramouli et al highlight, to ensure that the internal clocks of the nodes on which the tests are being executed are synchronised using a standard protocol, such as Network Time Protocol (NTP) (see http://www.ntp.org/ for further information) (Chandramouli, Goldstein, Barga, Riedewald, & Santos, 2011, p. 257). If the clocks are not standardised, there may be errors within the delivered results. Fortunately, for this study, Amazon Web Service's EC2 uses the Amazon Time Sync Service to ensure a consistent and accurate time reference across all nodes in a cluster[7].

3.4 Benchmarking Applications

The majority of stream processing applications tend to consist of relatively basic stream operations that are chained together to produce a result – for example, **filter**, **map** and **reduce**. From reviewing the literature, it is evident that there are a number of common benchmarking applications that are applied to stream processing frameworks. However, some of these are not suited for this study. Lu et al, for example, argue that Word Count is "widely accepted as a standard micro-benchmark for big data" (Lu, Wu, Xie, & Hu, 2014, p. 72). This benchmark involves, as the name suggests, counting the occurrences

[7] See https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html

of words in a stream. Generally, there can be two types of Word Count; one that counts the words in an individual record, and one that counts the words across the entire stream.

In Lu et al's study, they take the second approach. Their implementation of Word Count, "first splits the words, then aggregates the total count of each word and updates an in-memory map," where the word represents the key and the current total count for that word is the value. Aggregative word counting can, generally, be represented with the below DAG. While Word Count is prevalent in many benchmarking works, there is an issue with using it in this study, one that pertains to Spark's micro-batching computational model and, specifically, the effects it has on capturing throughput and latency.

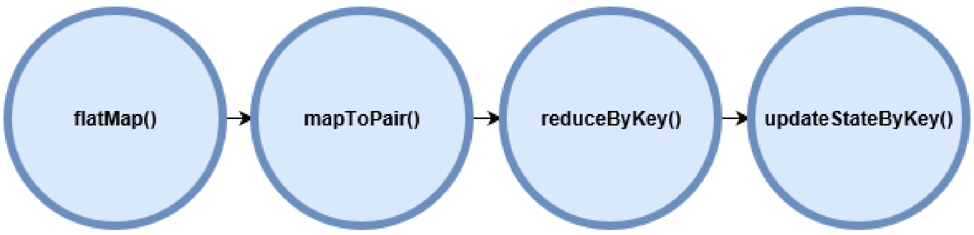


Figure 4. The directed acyclic graph of Aggregate Wordcount.

StreamBench, as set forth by Lu et al, measures throughput in MB/s, while latency is measured in milliseconds. The throughput metric, they argue, is "the average count of records as well as data size in terms of bytes processed." Latency, they continue, is "the average time span from the arrival of a record till the end of processing of the record" (Lu, Wu, Xie, & Hu, 2014, p. 72). With tuple-by-tuple stream processing models, measuring these values is relatively simple. However, with Spark's computational model, the task becomes much harder, if not impossible.

In the micro-batching model, all arriving records are periodically grouped into batches, where a batch is represented as an RDD. Actions are then taken on the entire batch, not single events. When calling each operation in the above Wordcount DAG, a new RDD is created. Therefore, once the first operation is called, it becomes almost impossible to track individual events as they go through the system. As a result of this, the only approach for identifying when an individual record's processing has been completed is to assume that the end of processing is the stage at which the entire batch completes. This, however, may not give an accurate representation of performance.

In StreamBench, it is entirely possible for the in-memory map to be updated with all of the words that constitute a single event early in the updateStateByKey() stage. Yet, if we assume that the batch end time is the end of processing for all records, the latency for that event may appear significantly higher than it otherwise would be, even though computation has been completed and the results delivered. It’s worth noting here that, as StreamBench’s source code is not publicly available and Lu et al do not go into detail about how they capture metrics, this may not be how they handle the issue of capturing the end of processing for a record. However, Wordcount, as described by Lu et al, is not a suitable benchmarking application for this work. With that said, it does provide insight into the types of applications that can and should be used.

3.4.1 Application Selection Criteria

Given the definitions of latency and throughput as set forth in this study - coupled with the methods being used to capture those metrics - there are several requirements that must be met by any applications used for testing. These primarily relate to results. The application must output its results to an external system (Kafka in this case). Using an in-memory map, as StreamBench’s Wordcount does, is not compatible with the metric-collection approach used here. With that said, a distributed in-memory data structure, such as Redis, would suffice, but as Lu et al do not elaborate on what is meant by “in-memory map” in the context of the StreamBench, we cannot assume that it refers to a distributed system.

A second requirement for is that the results must have a one-to-one mapping, such that a single input event has a corresponding output result. As specified previously, timestamps appended to events as they move through the stream processing frameworks is the primary way in which latency and throughput is measured. Operations that aggregate results from single events cannot be tracked as they progress through Spark. Granted, this excludes a number of common stream processing operations from this benchmarking study, such as **filter**, and also excludes, as mentioned, tests that are considered standards in the area of Big Data benchmarking, but, ultimately, this is the approach needed to fairly compare the frameworks' performance.

With the above in mind, the following sections will discuss the benchmarking applications that will be used in this study. Four, in total, have been selected, with each conforming to the above-specified criteria. These applications will be implemented in Java, as each of the frameworks provide a Java API. While this is less than the seven used by both Lu et al and Qian et al in their works, this study attempts to apply its applications to four frameworks, as opposed to Lu's two (Spark and Storm) and Qian's three (Samza, Spark and Storm). Before discussing the applications, the next section will briefly talk about the dataset that will be used for the experiments.

3.4.2 Experiment Dataset

The dataset that will be used in this study consists of a collection of Twitter messages that were sourced from the general Twitter stream over a period of three days in October, 2017. The dataset was downloaded from archive.org[8] as three separate archive files. Each archive consists of a number of individual JSON files that represent the data retrieved from the Twitter stream for a single minute of the day. Each of these JSON files is approximately 12MB in size and contains roughly 2200 Twitter messages. There are several message types that may be delivered via a Twitter stream, including maintenance messages, compliance messages and standard tweet messages[9]. From a cursory glance, it is evident that the majority of messages within the dataset are standard tweets, with some compliance messages also included.

Tweets, according to the Twitter developer documentation, are the "basic atomic building blocks of all things Twitter"[10]. The tweet object contains a number of root-level attributes, including *created_at*, which represents the timestamp of when the object was created, and *text*, which stores the actual UTF-8 text of the status update. Coupled with

[8] See https://archive.org/details/archiveteam-twitter-stream-2017-10

[9] See https://developer.twitter.com/en/docs/tweets/filter-realtime/guides/streaming-message-types

[10] See https://developer.twitter.com/en/ docs/tweets/data-dictionary/overview/tweet-object

this, there are also several child objects, such as, for example, the User object, which contains information about the poster of the tweet. The size of Tweet messages can vary greatly, ranging from around 2KB to 17KB. Status deletion notices, on the other hand - which are compliance messages that indicate a given Tweet has been deleted - contain simply a user id and the id of the deleted tweet. As such, they are relatively small, at about 150 bytes each.

It was originally intended that Kafka Connect would be used to feed the data into the relevant topics during the experiments. Connect is a framework included with Kafka that allows for the easy import and export of data to and from topics[11]. To achieve this, various *connectors* are offered out-of-the-box. These can fall into one of two categories: the first, source connectors, import data into Kafka, while the second, sink connectors, are used for export. For this study, the FileStreamSource connector was selected. This connector takes an input file location and an output topic name as parameters, and then, once started, streams the input file, line by line, into the configured topic.

Unfortunately, while the FileStreamSource connector worked as intended for smaller files, it was not able to handle larger files (those in the gigabyte range) and would throw an OutOfMemoryError shortly after starting. Upon further investigation, specifically of the *issues* section of the Apache website[12], it became apparent that this was a bug in the connector. A pull request to resolve the problem had been raised and closed[13], meaning the issue had been fixed, but, at the time of writing, it had not yet been added into the latest release of Kafka. As a result of this, the initial plan – which consisted of having one uber file (approximately 18GB in size) and a dedicated FileStreamSource connector living on each of the Kafka nodes - was no longer possible.

To compensate for FileStreamSource's shortcomings, a custom Java application was built. Called the kafka-file-loader, this Java application reads a file from the local file

[11] See https://kafka.apache.org/quickstart#quickstart_kafkaconnect

[12] See https://issues.apache.org/jira/browse/KAFKA-4335

[13] See https://github.com/apache/kafka/pull/4356

system and producers it, line-by-line to a specified Kafka input topic. At a glance, this new application seems to outperform the connector in terms of messages produced per second. This may be because the connector uses throttling to limit the rate of produced messages. Regardless, only a single instance of the kafka-file-loader is needed for the entire Kafka cluster.

Finally, before moving on to discuss the chosen applications, it is worth briefly mentioning why this dataset in particular was chosen. The justification is twofold. Firstly, social network data processing is one of the main use cases of stream processing frameworks. This is evident in the fact that, as mentioned, several of the frameworks under test either originated from, or were partially developed by, social networking companies. Secondly, the selected data represents the contents of an actual stream. Thus, its use here more closely aligns this study to real world use cases than those other works in which randomly generated data is used. With that said, the following sections describe, in detail, each of the benchmarking applications that will be run for the chosen frameworks.

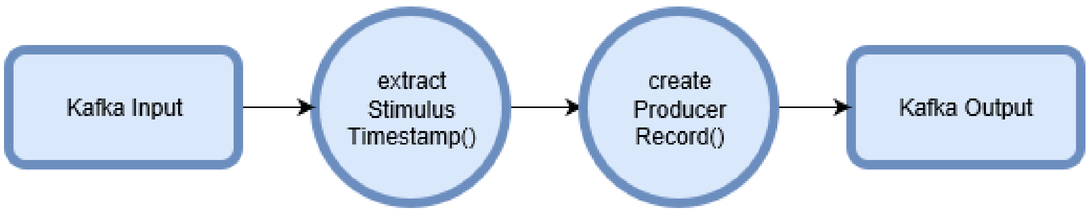
3.4.3 Identity

The first application selected for this benchmarking study is called Identity. Identity is based on the application of the same name from StreamBench. Its use here, however, differs slightly. In StreamBench, Identity reads an input and takes no operation on it (Lu, Wu, Xie, & Hu, 2014, p. 71). For this study, when the event arrives, it goes through a mapping operation that extracts the stimulus timestamp from the record's metadata. Incoming events, for reference, are represented by Kafka ConsumerRecord objects[14]. A ConsumerRecord is essentially a key/value pair object that contains some additional metadata, such as the name of the topic from where the record originated, the offset of the record (its position) in the corresponding Kafka partition, and also its timestamp. Extracting the stimulus time requires simply calling the ConsumerRecords timestamp() method.

[14] See https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/ConsumerRecord.html

Once the stimulus time has been extracted, the next step is to create the output event that will be emitted to the Kafka results topic. Output events are represented by Kafka ProducerRecords[15]. As with ConsumerRecords, the producer equivalent is a key/value pair object that contains some additional metadata. For our use case, there are only two fields of the ProducerRecord that are relevant: the first, topic, is used to specify the topic that the record should be sent to, and the second, value, is the value that is included with the record. It is worth clarifying here that, as the definitions of latency and throughput used in this study relate to the "end of processing," rather than the delivery of results, we do not need to emit the results of processing. As such, the value of the ProducerRecord – in not only this application, but also the others - consists simply of a list containing the stimulus time and the egress time.

Another point worth mentioning in relation to Kafka's record objects relates to the key element of the key/value pair combination. A key is purposely not included here, in the benchmarking applications, and also in the Java program that produces the input for these experiments. This is because of the way Kafka handles how it processes records. If no partition number or key is included in a record, Kafka will distribute the records in a round-robin fashion across the various partitions of a topic. If a partition number is not included, but a key is, Kafka will use a hash of the key to determine which partition to send the record to. Each partition has a single server that acts as the leader, which means it is responsible for handling all read and write requests to the partition. Thus, if we were to include a key, we could affect the load balancing across the Kafka brokers. While this may not have much impact on the results topic, it could have a negative effect on the performance at the input stage. Thus, the key here is left empty to ensure there is no impact on performance during benchmarking.



[15] See https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/prod-ucer/ProducerRecord.html

Figure 5. The various operations of the Identity benchmark.

3.4.4 Extraction

The next application used in this study is Extraction. Extraction, as the name suggests, involves extracting a field from an event. It - like all of the other applications in this study - builds upon the steps taken in the previous application. As with Identity, Extraction begins by reading an input from Kafka and then performing a mapping operation on the event. It, too, extracts the timestamp from the event's metadata, but it also needs to handle the event's payload, which, for this study, is a Twitter message object. To accomplish this, it first needs to deserialise the value into a more useable format. It does this by creating a JSON object of the event's JSON string payload.

Once the deserialisation process has been completed, the next step is to attempt to retrieve the 'text' field from the object. The word "attempt" is used here because, as mentioned previously, there are numerous different message types delivered via Twitter streams. Some of these messages do not have a text value, so, if the text field is not present, an empty string is returned instead. Regardless of the content of the text value, once the field has been extracted, the final step, as with Identity, is to create a ProducerRecord that includes the stimulus timestamp and the egress timestamp. The record is then emitted to the result topic.

While the operations for the Extraction application might appear similar to that of Identity's -in that both extract fields - there are several important extra steps being performed within the mapping stage, specifically deserialisation and extraction. Deserialisation of complex objects is a significant component of real world use cases of stream processing frameworks. Yahoo's Streaming Benchmarks, for example, uses a similar deserialization technique in its ad campaign application. This is important because their framework, they argue, was created to fill a gap in existing benchmarking works. Other benchmarks, they found, did not test "anything close" to real world use cases[16]. This, as a result, limits the usefulness of these frameworks. Thus, while the Extraction benchmark is relatively simple in its design, its use here adds additional value to the output of this study.

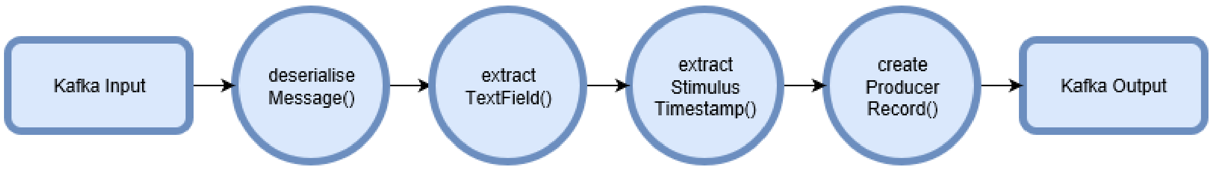


Figure 6. The various operations of the Extraction benchmark (order of execution may differ slightly across frameworks).

3.4.5 Grep

The third application comprising this study's benchmarking suite is Grep. Grep is based off the Unix command-line utility of the same name. Like its namesake, the Grep benchmark searches the text of incoming records for the presence of a specified regular expression, specifically, for this study, the hashtag symbol (#). This symbol plays an important role within Twitter. It is used to index keywords or topics, making it more likely for a Tweet to be shown in a Twitter search of the word[17]. A hashtag can be included anywhere within the text of a Tweet, but not all Tweets contain hashtags. In their 2016 work, Otsuka et al found that, of 8.3 million sampled Tweets, approximately 13% contained at least one hashtag (Otsuka, Wallace, & Chiu, 2016, p. 10).

Grep expands upon the Extraction benchmark, in that it not only extracts the 'text' field from incoming events, but also evaluates whether the field contains any hashtags. In a production environment, this type of check might be used to filter out messages from a data stream. However, given the need for a one-to-one mapping between input and output in this study for metrics collection, rather than filter the record, a mapping function is instead used. If the record is found to contain a hashtag, a boolean true value is returned. If no hashtag is found, or the event is of a type that does not contain a 'text'

[16] See Background section of README on https://github.com/yahoo/streaming-benchmarks

[17] See https://help.twitter.com/en/using-twitter/how-to-use-hashtags

field, false is instead returned. As with the previous application, the final step is to output an event containing the required timestamps.

Grep - and the hashtag symbol - was chosen for several reasons. Firstly, it is an application that is commonly present in Big Data benchmarks. This is because, as Lu et al argue, “it is a simple yet common operation,” one that is likely to be used in many real-world use cases (Lu, Wu, Xie, & Hu, 2014, p. 72). The second reason is because of the growing interest in the development of hashtag recommendation systems, as posited by Otsuka et al. They argue that, as tagging culture becomes more widely adopted - thanks in part to the development of the hashtag search feature in Twitter - many individual users and business marketers have started applying tagging to organise posts into related conversations. This has created a need to develop recommendation systems. And while Otsuka et al’s study uses Hadoop and MapReduce, as Twitter exposes its Tweets via a stream, using stream processing frameworks is the logical choice when it comes to handling the computation of any recommendation systems.

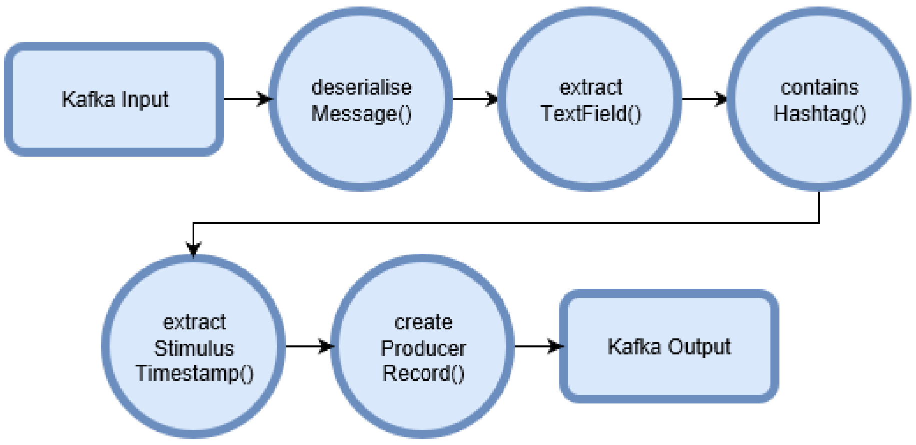


Figure 7. The various operations of the Grep benchmark (order of execution may differ slightly across frameworks).

3.4.6 WordCount

The final of the four applications comprising this study’s benchmarking suite is WordCount. Word counting, as mentioned previously, is a standard benchmark in the field of Big Data processing. Unlike the Wordcount used in StreamBench - which keeps an aggregate total of the words across an entire stream - the implementation used for this study counts the occurrence of words on a per record basis. This approach is taken to conform to the requirement of a one-to-one mapping of input and output events. With that said, many of the operations executed for the aggregate version of the benchmark are also used here. Indeed, in several of the frameworks, modifying the benchmark to run the aggregate version requires changing only several lines of code.

Of the four benchmarks in this study, WordCount is the most complex in terms of number of steps executed. WordCount, as with Grep, builds upon both the Identity and Extraction benchmarks. When a record arrives, the first step is to parse the event into a more usable JSON object. Once deserialised, the 'text' field, if present, is extracted. If there is no 'text' field, as with the previous applications, an empty string is returned. If one is present, the field is split into a stream of words using the regex \\W+, which splits on "non-word" characters[18]. Each value in this stream is then mapped to a key/value pair, where the word represents the key and the value is set to 1. The stream is then reduced by the key, with the value being summed for cases where there are two or more instances of the same word.

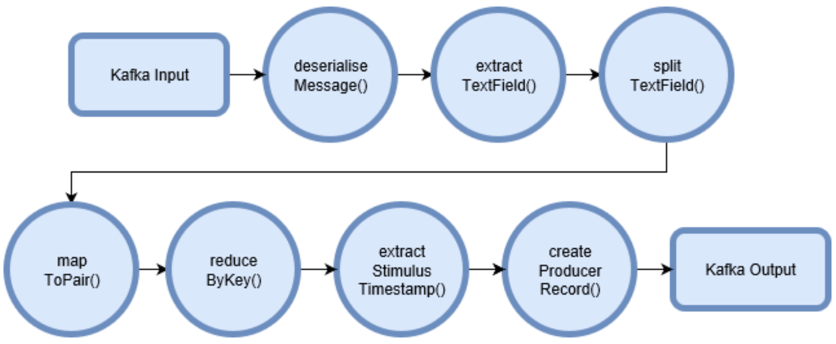


Figure 8. The various operations of the WordCount benchmark (order of execution may differ slightly across frameworks).

[18] See https://docs.oracle.com/javase/tutorial/essential/regex/pre_char_classes.html

4 IMPLEMENTATION AND RESULTS

This chapter discusses the implementation of the experiments and the results collected during their running. In total, 20 experiments were executed across three of the frameworks: Flink, Spark and Storm. Unfortunately, whilst implementing the various benchmarks for Samza, it quickly became apparent that it could not be tested using the approach proposed in this study. This is because of the way Samza is integrated with Kafka. Samza provides an interface, called *SystemConsumer*[19], that must be implemented by any systems wishing to integrate with Samza. This interface contains four methods that the implementing class must define: poll(), register(), start() and stop().

KafkaSystemConsumer[20] is the concrete implementation of the *SystemConsumer* interface that is included out-of-the-box with Samza. This class reads messages from Kafka and parses them into *IncomingMessageEnvelopes*[21]. These envelopes are then passed to *StreamTask*[22] implementations, which are the basic classes upon which Samza jobs are built. The issue pertaining to this study lies with the fact that the *KafkaSystemConsumer* implementation does not pass the Kafka timestamp into the *IncomingMessageEnvelope*s, which themselves do not actually contain any timestamp property to hold that value. Furthermore, the specific section of the *KafkaSystemConsumer* that is responsible for passing the Kafka message is private, meaning that it cannot be easily extended to modify the functionality.

The option of creating a new custom consumer was investigated. Consumers are configured in Samza using the systems.samza.factory value of the individual jobs'

[19] See http://samza.apache.org/learn/documentation/0.14/api/javadocs/index.html?org/apache/samza/system/SystemConsumer.html

[20] See https://github.com/apache/samza/blob/master/samza-kafka/src/main/scala/org/apache/samza/system/kafka/KafkaSystemConsumer.scala

[21] See http://samza.apache.org/learn/documentation/0.14/api/javadocs/org/apache/samza/system/IncomingMessageEnvelope.html

[22] See https://samza.apache.org/learn/documentation/0.14/api/javadocs/org/apache/samza/task/StreamTask.html

properties files. This parameter tells Samza which *SystemFactory*[23] it should instantiate. The *SystemFactory* is used by Samza to construct the relevant *SystemConsumer* (and *SystemProducer*). Thus, for implementing a custom consumer, a new factory implementation, coupled with a consumer implementation and a new message implementation (required to hold the timestamp), would need to be created. Whilst the factory and envelope implementations are relatively straightforward, the consumer implementation is far more involved.

Generally, according to the API's documentation (see footnote 10 on the previous page), there are three *SystemConsumer* implementation styles. These are: thread-based, selector-based and synchronous. Thread-based implementations use a series of threads to read from the underlying system asynchronously. The resulting messages are then put onto a queue, which is read from when the poll() method is invoked. Selector-based implementations, on the other hand, set up NIO-based[24] non-blocking sockets that can be selected for new data whenever poll() is called. Finally, synchronous implementations fetch directly from the underlying system upon poll() invocation.

Given the complexity of creating a custom consumer (dealing with multithreading and buffering, etc...), it was decided that the best approach would be to set aside the Samza experiments. This is because any results gained from testing would not be objective, as they would be overly dependent on this researcher's ability to implement an efficient consumer. Furthermore, the *KafkaSystemConsumer*, from some quick testing, appears to work as well as one might expect. Thus, there is no obvious reason - aside from the lack of the Kafka timestamp - why an organisation would not use this implementation in production. And as this study seeks to test production like use cases, this is a further reason not to implement a custom solution.

[23] See http://samza.apache.org/learn/documentation/0.14/api/javadocs/org/apache/samza/system/SystemFactory.html

[24] See https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

With the above clarified, the following section will discuss the experiment setup, as pertaining to Flink, Spark and Storm. Briefly, however - on a final note regarding Samza - as the framework is still relatively immature (being currently in version 0.14) and timestamps were only recently added to the Kafka platform[25], it is possible that coming Samza versions may handle the value. This is highly likely, especially when considering that the Amazon Kinesis[26] – a system similar to Kafka – integration currently contains a timestamp value in its *IncomingMessageEnvelope* implementation[27]. Thus, a follow-up study could potentially be conducted when the value is added, in which the Samza experiments, as proposed in this study, are run, and the relevant metrics collected.

4.1 Experiment Setup

The experiments, as mentioned in Chapter 3, were executed across two three-node clusters: one dedicated to running Kafka, and one to running the stream processing frameworks. The Kafka cluster's nodes were labelled as *kafka1*, *kafka2*, and *kafka3*, while the computational nodes were *computational1*, *computational2* and *computational3*. The following sections will briefly discuss the setup of these clusters and the configuration of relevant applications.

4.1.2 Kafka Cluster Setup

On each of the Kafka nodes, Kafka version 1.0.0, built for Scala version 2.11, was installed. On *kafka1*, Zookeeper version 3.4.11 was installed. *kafka1* also hosted the experiment dataset, along with the lightweight kafka-file-loader application that was used to feed the dataset into Kafka. Finally, *kafka2* hosted the metrics-writer application, which was responsible for consuming the results output by the computational nodes, calculating latency, and then writing the results to the local filesystem.

[25] See https://kafka.apache.org/0100/documentation.html#upgrade

[26] See https://aws.amazon.com/kinesis/

[27] See https://github.com/apache/samza/blob/master/samza-aws/src/main/java/org/apache/samza/system/kinesis/consumer/KinesisIncomingMessageEnvelope.java

Node	Installed Software Versions
kafka1	kafka_2.11-1.0.0 zookeeper-3.4.11 kafka-fileloader
kafka2	kafka_2.11-1.0.0 metrics-writer
kafka3	kafka_2.11-1.0.0

Table 1. Kafka cluster installed software versions.

In terms of Kafka broker setup, the following configuration was used: num.partitions, which represents the default number of log partitions per topic, was set to 30. Partitions essentially represent the unit of parallelism within Kafka. This is because Kafka always gives a single partition's data to one consumer thread. Therefore, the degree of parallelism for consumers is bounded by the number of partitions being consumed. According to information posted on the Confluent website[28] (which was founded by the team that built Kafka), more partitions can lead to higher throughput, but this comes at the cost of increased end-to-end latency, which is defined by "the time from when a message is published by the producer to when the message is read by the consumer."

As a rule of thumb, when trying to balance latency and throughput considerations, Confluent recommends limiting the maximum number of partitions per broker to 100 * *b* * *r*, where *b* is the number of brokers in a Kafka cluster and *r* is the replication factor. Given the setup of the computational cluster used in this study, coupled with the individual configurations of the frameworks under test (both of which will be discussed shortly), 30 partitions was identified as being an adequate number, providing the required throughput, whilst also maintaining a sufficiently low end-to-end latency.

Aside from num.partitions, the only other values changed from their default setting within Kafka were auto.create.topics.enable and log.message.timestamp.type, which, as discussed previously, was set to LogAppendTime. auto.create.topics.enable relates to

[28] See https://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/

Kafka's behaviour when a consumer or producer tries to read from, or write to, a topic that doesn't exist. This was set to true as a convenience, as each experiment requires the creation of two new topics, one for the incoming data feed and one for outgoing results. The approach of creating fresh topics was taken to ensure that no messages from previous experiments were inadvertently consumed by the following ones. Finally, Zookeeper, for reference, was left at its default configuration.

4.1.2 Computational Cluster Setup

On the computational cluster, Flink version 1.4.1 (bundled with binaries for Hadoop 2.8), Spark version 2.2.1 (prebuilt for Apache Hadoop 2.7 and later), and Storm version 1.2.1 were installed on each of the three nodes. It was intended that Flink and Spark would be tested in both standalone cluster mode and also running on YARN, so the latest release of Hadoop, version 3.0.0, was also installed across the cluster. Upon testing Flink and YARN, however, it emerged that there was a bug in version 1.4.1 (and also 1.4.0), which prevented jobs from running on YARN. The specific error related to vcore detection. Flink would throw an exception whenever a job was submitted, stating that the number of configured vcores for the job was greater than the number available on YARN.

Several proposed solutions were attempted to correct the issue, such as explicitly declaring the number of vcores in YARN's configuration (using yarn.nodemanager.resource.cpu-vcores), but the problem persisted. A quick test showed that Flink version 1.3.2 worked as expected, but as the benchmarking applications had been built using Flink's 1.4.1 API, they no longer functioned with the older version. Furthermore, version 1.3.2 is currently two versions behind the latest release. With version 1.5.0 on the horizon, it would make little sense to test the older version of Flink, just to see how it performs when running on YARN. As such, it was decided that only Flink's standalone cluster mode would be tested for this study.

In terms of node designation, *computational1* was configured as the master node for Flink and Spark's standalone cluster modes. It was also configured as the nimbus node for Storm, and the resource manager for YARN. *computational2* and *computational3*, meanwhile, were listed as slave/worker nodes for each framework. This was also the case for *computational1*, because, in relatively small clusters, such as the one used in this study, the work required of any master process is minimal. This, therefore, allows for the node to double as an additional worker.

Node	Installed Software Versions
computational1	flink-1.4.1 hadoop-3.0.0 spark-2.2.1-bin-hadoop2.7 apache-storm-1.2.1
computational2	flink-1.4.1 hadoop-3.0.0 spark-2.2.1-bin-hadoop2.7 apache-storm-1.2.1
computational3	flink-1.4.1 hadoop-3.0.0 spark-2.2.1-bin-hadoop2.7 apache-storm-1.2.1

Table 2. Computational cluster installed software versions.

With the setup of the experiment clusters now clarified, the following section will discuss some observations and details regarding the implementation of the benchmarking applications, the specific configuration of each framework, and any other points deemed noteworthy.

4.2 Implementation Details

The various applications created for this study can be found on GitHub[29]. Each framework's benchmarking applications are bundled into individual projects, which,

[29] See https://github.com/jonathan-curtis/stream-benchmark

when built, produce single Java executable jar files. The Factory Design Pattern[30] is used within these projects to determine, at runtime, which benchmark to select, based on passed in parameters. There are several implementation details that should be elaborated upon, pertaining mainly to the way in which the Kafka timestamp is extracted from incoming messages. There are also various configurations that must be discussed.

4.2.1 Extracting Kafka Timestamp

Each framework required a different approach for extracting the Kafka timestamp. For Storm, some minor development work was required, while for Flink, extra configuration needed to be set. Spark, on the other hand, provided the functionality for extracting the timestamp straight out-of-the-box. Beginning with Flink, Flink's documentation related to retrieving the Kafka timestamp is somewhat vague. Timestamp extraction is mentioned briefly on the Kafka connector section of the website[31]. The documentation claims that the *FlinkKafkaConsumer010* class will emit records with the timestamp attached if the "time characteristic" in Flink is set to event time. This requires calling setStreamTimeCharacteristic() on the StreamExecutionEnvironment, and passing in TimeCharacteristic.EventTime.

One might initially assume that calling the Kafka consumer, with the time characteristic set, returns a Kafka *ConsumerRecord* object, or something similar, that has the timestamp included. This, however, is not the case. Rather, calling the consumer returns a deserialised instance of the Kafka message's value. Accessing the timestamp requires additional knowledge of Flink's processing model.

The basic building blocks of streaming applications in Flink consist of events, state and timers[32]. Each record ingested into Flink has some metadata attached to it, which falls

[30] See https://github.com/jonathan-curtis/stream-benchmark/blob/master/flink-applications/src/main/java/com/curtis/benchmarking/BenchmarkFactory.java as an example

[31] See https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/connectors/kafka.html

[32] See https://ci.apache.org/projects/flink/flink-docs-release1.4/dev/stream/ operators/process_function.html

into the state and timers categories. This metadata is accessed via the low-level *ProcessFunction* operation, which exposes the data via a *Context* object. It is from this context that the Kafka timestamp must be retrieved. Thus, extracting the timestamp for this study required the implementation of a custom *ProcessFunction*[33], which was passed into the process() method as it was called on the stream returned by the *FlinkKafkaConsumer010*.

Storm's processing model warranted that a different approach be taken for extracting the timestamp. As mentioned in chapter 2, Storm consists of spouts and bolts, nodes that are responsible for ingesting and processing data, respectively. Storm passes data between these different nodes in the form of data tuples, which are essentially ordered lists of data. Storm's Kafka integration is provided via the *KafkaSpout* class[34]. This class requires that a *RecordTranslator*[35] be provided in order to deserialise the ConsumerRecord into a tuple. The default *RecordTranslator* implementation, *DefaultRecordTranslator*[36], does not extract the timestamp. Therefore, a custom extractor needed to be created[37]. This *TimestampRecordTranslator* is almost identical to the default implementation, so, therefore, should have no negative affect on Storm's performance.

4.2.2 Framework Configuration

In total, 20 benchmarking experiments were executed across 5 different configurations of the frameworks under test. These configurations were:

[33] See https://github.com/jonathan-curtis/stream-benchmark/blob/master/flink-applications/src/main/java/com/curtis/utils/ExtractKafkaTimestamp.java

[34] See https://storm.apache.org/releases/1.2.1/storm-kafka.html

[35] See https://github.com/apache/storm/blob/master/external/storm-kafka-client/src/main/java/org/apache/storm/kafka/spout/RecordTranslator.java

[36] See https://github.com/apache/storm/blob/master/external/storm-kafka-client/src/main/java/org/apache/storm/kafka/spout/DefaultRecordTranslator.java

[37] See https://github.com/jonathan-curtis/stream-benchmark/blob/master/storm-applications/src/main/java/com/curtis/kafka/TimestampRecordTranslator.java

- Flink running in Standalone Cluster mode
- Spark Standalone Cluster mode
- Spark running on YARN with default configuration
- Spark running on YARN with maximum requested executors
- Storm

As mentioned, when running Flink in standalone mode, the cluster was started with one JobManager (master) and three TaskManagers (slaves/workers). Spark Standalone was started with one master and three slaves. The configuration of these components, in both cases, was left at their default values. This was done to ensure that a fair comparison was conducted between them because, ultimately, there are far more configuration options than a single researcher could possibly test within a reasonable timeframe. Furthermore, each option could potentially have a drastic effect on performance. By not providing any configuration, aside from the most basic required for the cluster to function, this removes the possibility of errors being introduced that could affect the objectiveness of the results.

For Spark running on YARN, two different configurations were used. When initially setup on the cluster, YARN identified that there were 24 vcores available across the three instances, coupled with 24GB of memory. The concept of a vcore in YARN differs from that of AWS. YARN's vcore represents the "usage share of a host CPU[38]," not an abstraction of a physical core. For Spark on YARN's first configuration, the benchmarking applications were simply submitted to YARN, without overriding any values. This resulted in YARN assigning only two vcores to the application (one for the Spark driver process and one for the executor performing the application logic). For the second configuration, the *num-executors* property was overridden from its default value of 1. 20 executors were requested, but YARN would only assign a maximum of 11.

With regards to Storm's configuration, one nimbus node and three master nodes were started. In order to show a comparison between Storm and Spark running on YARN, Storm's default values were overridden in order to take full advantage of the resources on the cluster. Specifically, the number of workers was set to 3 programmatically within

[38] See https://www.cloudera.com/documentation/enterprise/5-14-x/topics/cdh_ig_yarn_tuning.html

the code, meaning that one worker process should, in theory, be instantiated for each of the supervisor nodes. Furthermore, the number of executors for the various components making up the topologies in the benchmarking applications was set to 6. According to the documentation[39], this means that, again in theory, each of the EC2 vcores should spawn between two and four threads, based on the topology being run. With 6 cores, that means there should be a maximum of 24 threads. This, thus, in some ways equates to YARN's maximum vcore value of 24. As a side note, while Storm can be run on YARN, the integration project[40] - at the time of writing - is self-described as a "work in progress." As such, it was excluded from this study.

4.3 Experiment Results

The experiments were executed over a three-day period in February, 2018. The execution time of individual experiments varied from a minimum of just under 12 minutes to a maximum of slightly over 34 minutes. Each experiment required a significant amount of preparation time beforehand, and some time afterwards, which was used to ensure that the results were extracted from the output topic. Some experiments needed to be executed multiple times because of failures or other issues. Each experiment was conducted against a dataset consisting of 13,124,700 individual records. It took, on average, approximately 5 minutes for the kafka-file-loader application to feed the data into the input topics on Kafka. This was verified by examining the metadata of the logs for the kafka-file-loader application (which in one instance, for example, showed a creation time of 07:54:24 and a final modified time of 07:59:04).

It is worth briefly clarifying that the approach taken in this study, unlike in others, does not limit the Kafka event generation rate (which is approximately 44,000 events per second). This is done to ensure that the maximum throughput can be identified. As a result of this approach, coupled with the method used for measuring latency, the

[39] See https://storm.apache.org/releases/1.2.1/Understanding-the-parallelism-of-a-Storm-topology.html

[40] See https://github.com/yahoo/storm-yarn

observed latency within this study might seem incredibly high in some cases, especially when compared to other works. This is because the latency is measured from when the event is added to the Kafka topic, not from when it is ingested into the framework. And as the records are added at a rate that is faster (as will be seen in the results below) than the rate at which the frameworks can consume them, for many of the benchmarks, the ingested records can be sitting in the message queue for a significant amount of time before being processed. This, however, demonstrates that the message generation process is not acting as a bottleneck for the system, something which was identified as a concern in the methodology section.

Arguably, the definition of latency used in this study is a more relevant interpretation of the value. This is because, in many cases – such as for targeted advertising systems – results are needed in real-time. As such, the time records spend in message queues waiting for processing should be taken into consideration. With that said, when the latency is measured in this study via the frameworks' internal reporting tools (where applicable), it appears to be more in line with what other works have reported when executing similar benchmarking applications.

Figure 9 below shows the reported processing time for Spark, which was taken from the Spark UI during the Identity benchmark (where the framework was running on YARN using the default configuration). It shows an average latency in milliseconds, rather than the seconds, and even minutes, that were observed for the metric using the definition defined in this study. In fact, the value of 599ms is almost identical to the value reported by Lu et al in their work (Lu, Wu, Xie, & Hu, 2014, p. 75). This makes sense, as they define latency as the average time span from the arrival of a record until the end of its processing. One thing worth noting, however, is that, as a consequence of the approach taken, the results of this work cannot necessarily be compared with those of other studies, unless a similar methodology has been used.

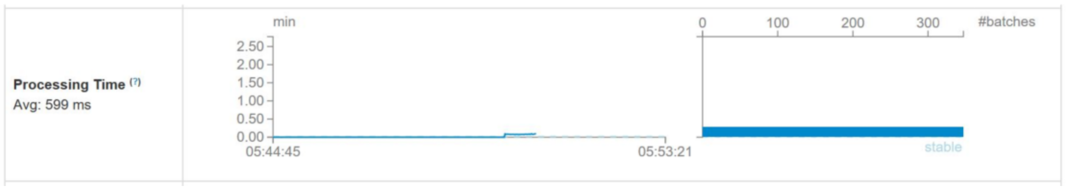


Figure 9. Spark’s processing time, which shows values in the milliseconds.

The following sections will now present the results that were obtained from executing the various experiments. The results are presented in two formats. The first is a line chart that plots the throughput per second captured while executing the experiment. The second is a table that shows a statistical summary of the values captured for both latency and throughput. Where applicable, certain results or configuration details may be discussed.

4.3.1 Identity

4.3.1.1 Flink

The mean throughput rate for Flink while executing the Identity benchmark was 15,928 events per second. All 13.1 million records were processed in a little over 800 seconds. Figure 10 below shows the throughput per second as captured throughout the entire length of the experiment. In terms of latency, table 3 shows a statistical summary of the observed values. The mean latency was 64,777ms, with a minimum of 0ms and a maximum of 337,939ms. The standard deviation for the latency was 104,111ms.

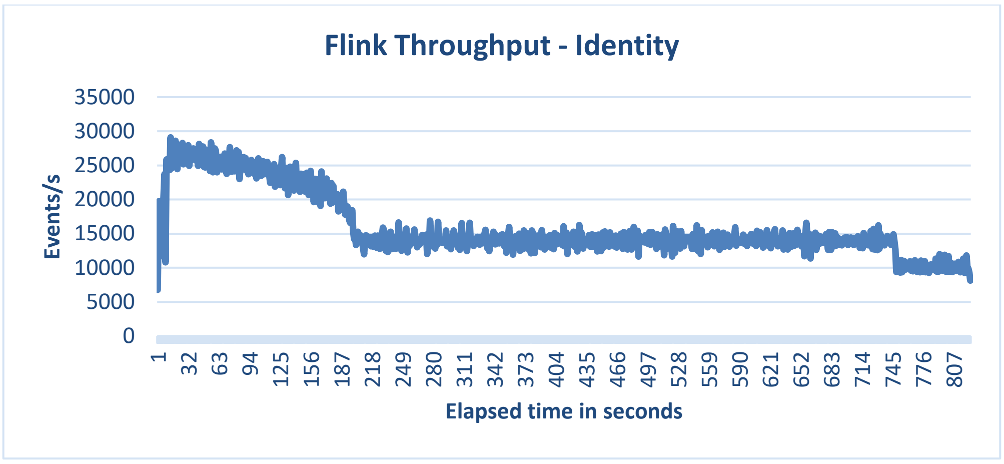


Figure 10. The throughput of Flink for the Identity benchmark.

Flink – Identity Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	824 (seconds)
mean	64777.22277 (ms)	15928.03398 (records)
standard deviation	104111.61869 (ms)	4752.17413 (records)
min	0 (ms)	6779 (records)
max	337939 (ms)	29092 (records)

Table 3. The statistical summary of the observed latencies and throughput collected from Flink during the Identity benchmark.

4.3.1.2 Spark Standalone Cluster

In contrast to Flink, Spark running in local cluster mode only managed to process a fraction of the total records. Of the 13 million records fed into the system, less than 5.5 million results were delivered to the output topic. As fault tolerance was not in scope for this study, the exact reason why so many records failed was not throughly investigated. From a brief search of the documentation, several possibilities appeared likely. For example, the problem may be related to a thread - or threads - not being given enough CPU time. Alternatively, it could possibly be caused by a timeout related to reading from the Kafka input topic. Regardless, the mean throughput rate was 7,825 events per second, with the experiment running for a total of 702 seconds. The mean latency was approximately 195,306ms, with a minimum of 1,342ms and maximum of 425351 (see figures X and X below for a more precise breakdown).

It is worth pointing out here that for each of the experiments conducted across all three of the Spark configurations, the batch interval was set to one second. This value was selected because the Spark documentation[41] advises that the interval should be set to a value where the batch processing time is less than the interval. As can be seen in figure 12, which shows the UI screen for the Spark standalone Identity benchmark, the processing time is significantly below the one second range. Therefore, this value provides an optimal balance between throughput and latency.

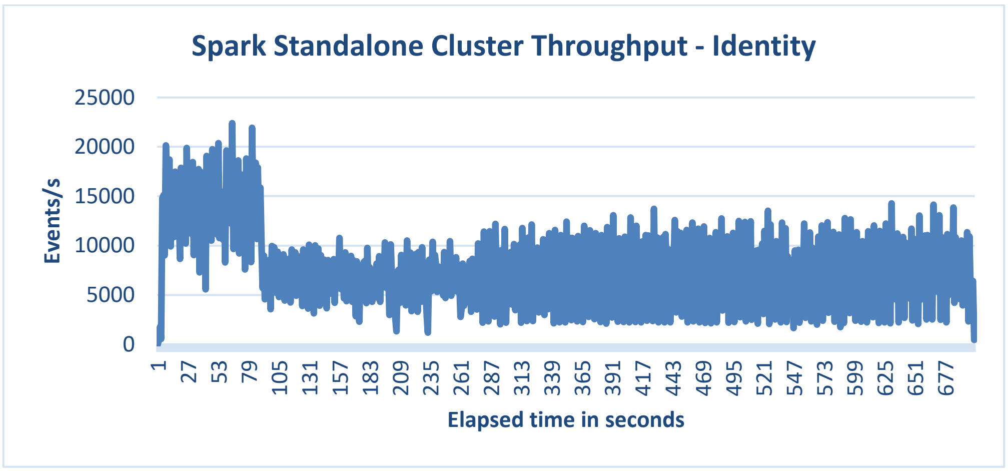


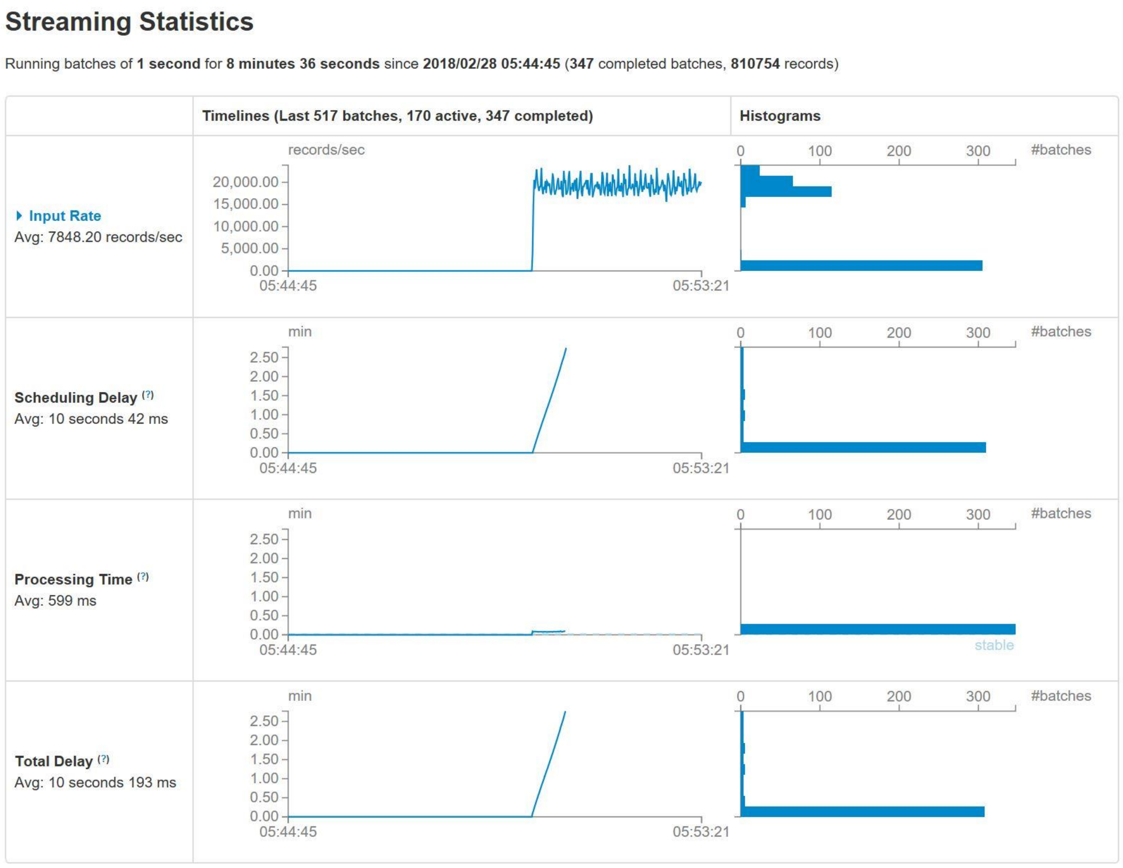
Figure 11. The throughput of Spark running in standalone cluster mode for the Identity benchmark.

[41] See https://spark.apache.org/docs/2.2.1/streaming-programming-guide.html#setting-the-right-batch-interval

Spark Standalone Cluster – Identity Benchmark

Summary	Latency Statistics	Throughput Statistics
count	5493085 (events)	702 (events/s)
mean	195306.04521 (ms)	7824.90883 (events/s)
standard deviation	130141.35115 (ms)	3954.18812 (events/s)
min	1342 (ms)	0 (events/s)
max	425351 (ms)	22394 (events/s)

Table 4. The statistical summary of the observed latencies and throughput collected from Spark (standalone) during the Identity benchmark.


Streaming Statistics

Running batches of **1 second** for **8 minutes 36 seconds** since **2018/02/28 05:44:45** (**347** completed batches, **810754** records)

	Timelines (Last 517 batches, 170 active, 347 completed)	Histograms
Input Rate Avg: 7848.20 records/sec		
Scheduling Delay (?) Avg: 10 seconds 42 ms		
Processing Time (?) Avg: 599 ms		
Total Delay (?) Avg: 10 seconds 193 ms		

Figure 12. The Spark UI screen showing the various parameters for the Identity benchmark being run on the Spark standalone cluster.

4.3.1.3 Spark YARN Default Configuration

In contrast to the standalone cluster mode, Spark running on YARN with default configuration performed significantly worse in terms of both throughput and latency. This, realistically, was to be expected, considering that the entire process ran via a single vcore within YARN. Like in standalone mode, it, too, also only processed a fraction of the overall dataset. The mean latency for the Identity benchmark was also the highest of all the framework/configuration combinations, coming in at 494,368ms.

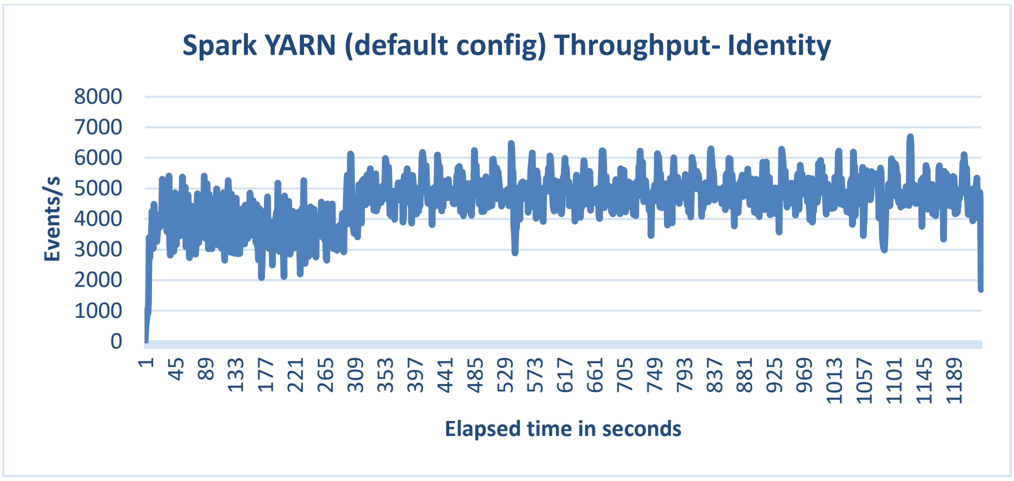


Figure 13. The throughput of Spark running on YARN with default configuration for the Identity benchmark.

Spark YARN (default config) Throughput – Identity Benchmark

Summary	Latency Statistics	Throughput Statistics
count	5715831 (events)	1230 (seconds)
mean	494368.41218 (ms)	4647.01707 (events/s)
standard deviation	259433.21035 (ms)	785.86049 (events/s)
min	1094 (ms)	0 (events/s)
max	932905 (ms)	6695 (events/s)

Table 5. The statistical summary of the observed latencies and throughput collected during the Identity benchmark from Spark running on YARN with default configuration.

4.3.1.4 Spark YARN Max Executor Configuration

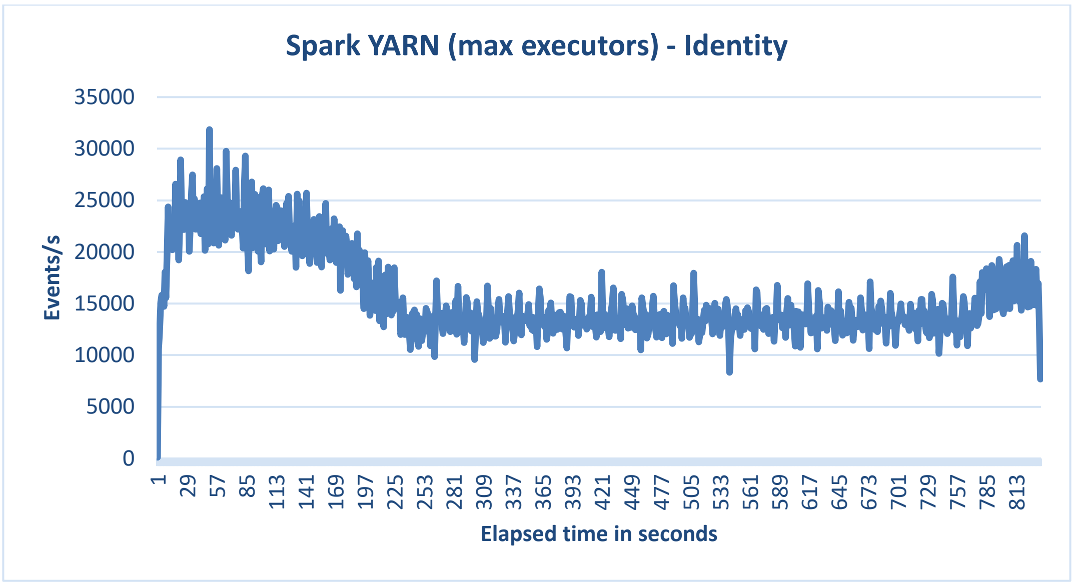


Figure 14. The throughput of Spark running on YARN with maximum requested executors for the Identity benchmark.

Spark YARN (max executors) Throughput – Identity Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	836 (seconds)
mean	38109.28682 (ms)	15699.40191 (events/s)
standard deviation	19954.36698 (ms)	4069.13669 (events/s)
min	363 (ms)	116 (events/s)
max	68394 (ms)	31859 (events/s)

Table 6. The statistical summary of the observed latencies and throughput collected during the Identity benchmark from Spark running on YARN with maximum requested executors.

4.3.1.5 Storm

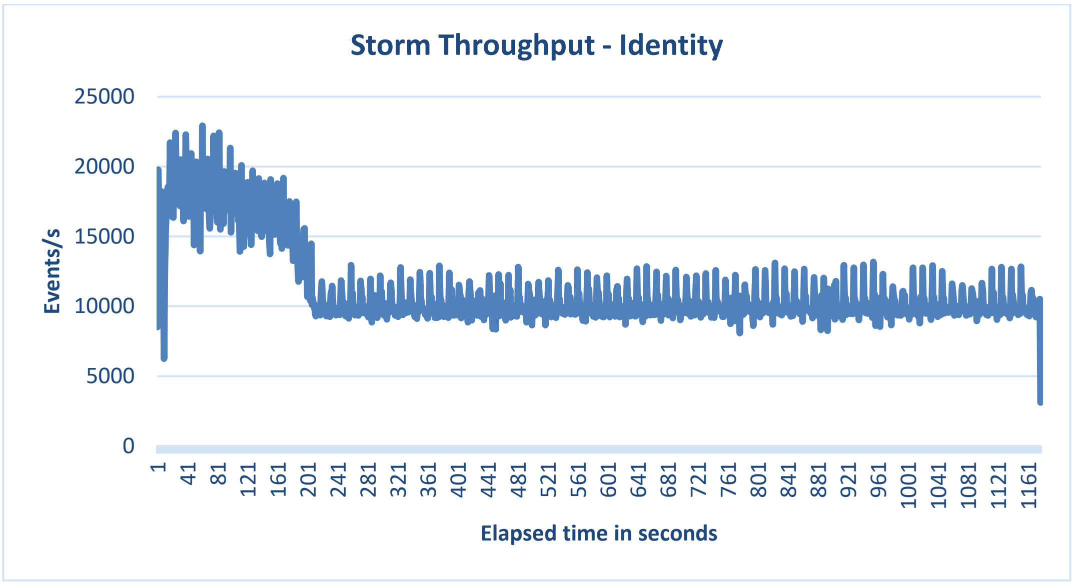


Figure 15. The throughput of Storm for the Identity benchmark.

Storm – Identity Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	1178 (seconds)
mean	57.68940 (ms)	11141.51104 (events/s)
standard deviation	111.03162 (ms)	2830.84451 (events/s)
min	0 (ms)	3113 (events/s)
max	1211 (ms)	22922 (events/s)

Table 7. The statistical summary of the observed latencies and throughput collected from Storm during the Identity benchmark.

4.3.2 Extraction

4.3.2.1 Flink

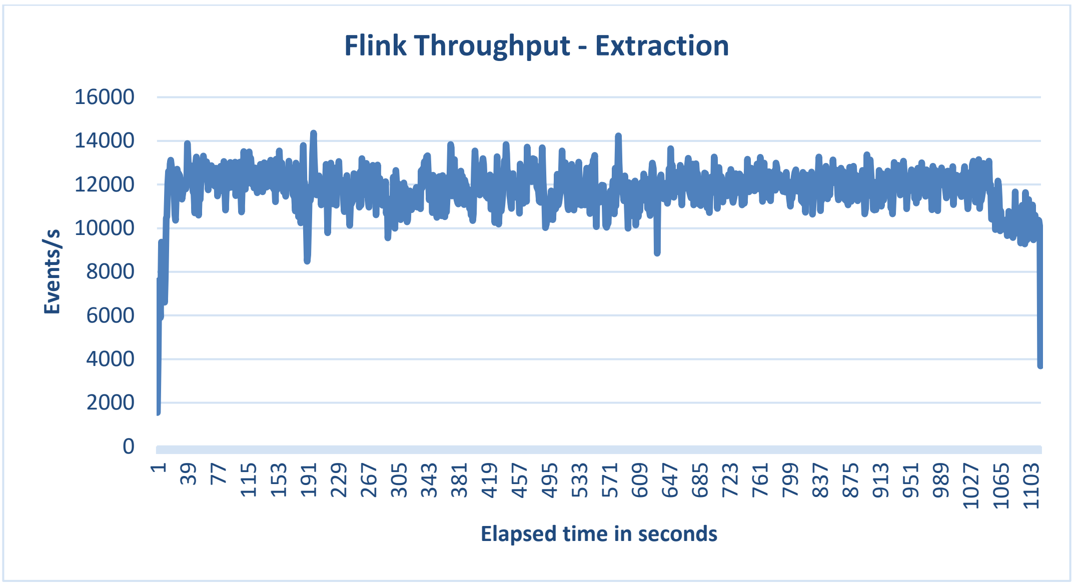


Figure 16. The throughput of Flink for the Extraction benchmark.

Flink – Extraction Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	1116 (seconds)
mean	288980.28997 (ms)	11760.48387 (records)
standard deviation	146899.08974 (ms)	1014.57978 (records)
min	18 (ms)	1550 (records)
max	551300 (ms)	14358 (records)

Table 8. The statistical summary of the observed latencies and throughput collected from Flink during the Extraction benchmark

4.3.2.2 Spark Standalone Cluster

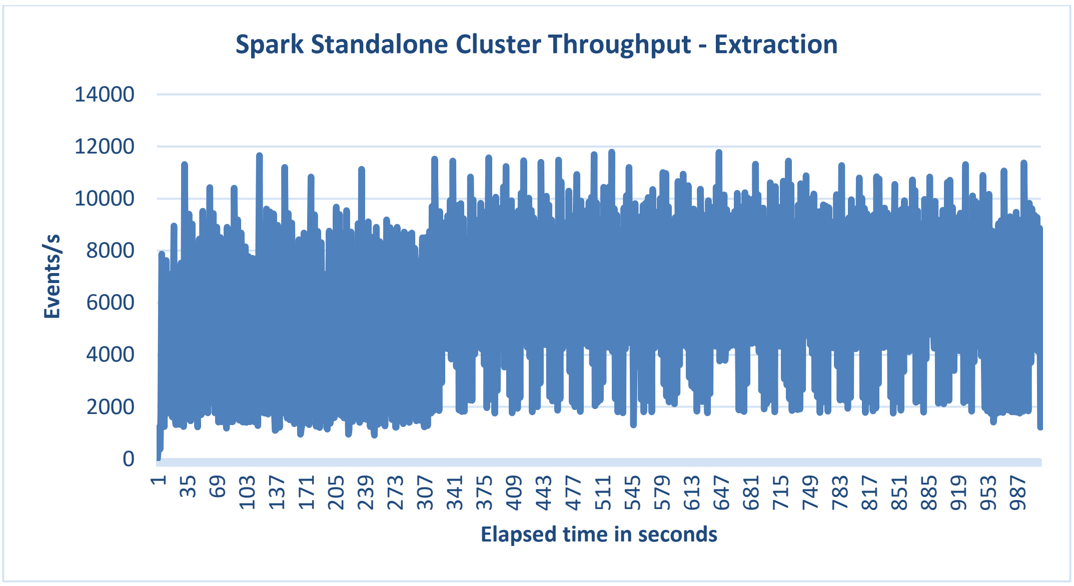


Figure 17. The throughput of Spark running in standalone cluster mode for the Extraction benchmark.

Spark Standalone Cluster – Extraction Benchmark

Summary	Latency Statistics	Throughput Statistics
count	5895172 (events)	1014 (events/s)
mean	384066.12264 (ms)	5813.77909 (events/s)
standard deviation	190177.01530 (ms)	3085.29367 (events/s)
min	1224 (ms)	0 (events/s)
max	701777 (ms)	11787 (events/s)

Table 9. The statistical summary of the observed latencies and throughput collected from Spark (standalone) during the Extraction benchmark.

4.3.2.3 Spark YARN Default Configuration

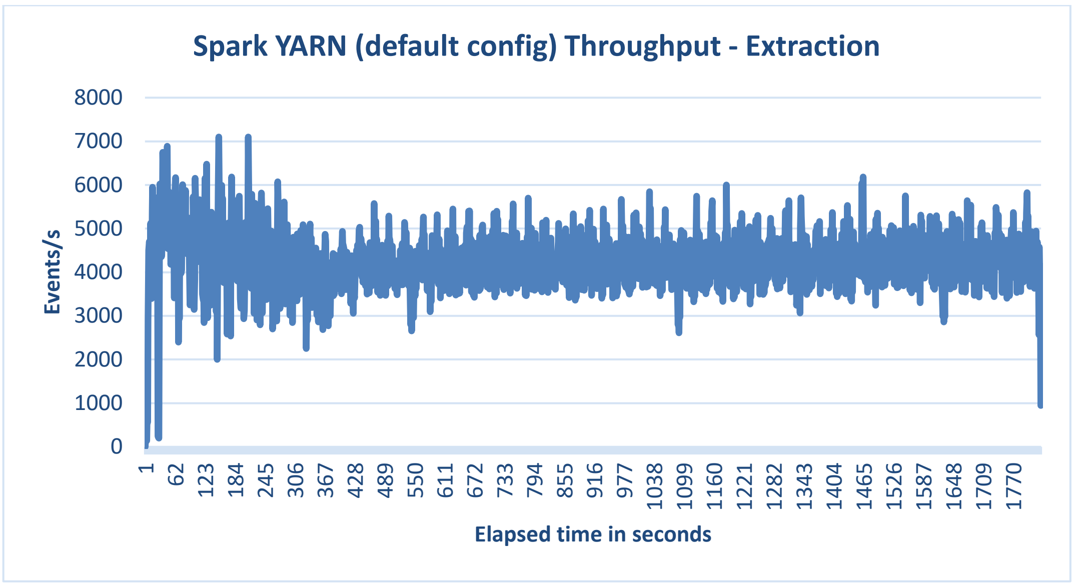


Figure 18. The throughput of Spark running on YARN with default configuration for the Identity benchmark.

Spark YARN (default config) Throughput – Extraction Benchmark

Summary	Latency Statistics	Throughput Statistics
count	7761199 (events)	1827 (seconds)
mean	734043.8456451123 (ms)	4248.056376573618 (events/s)
standard deviation	416098.89801168203 (ms)	684.9354304731373 (events/s)
min	922 (ms)	1 (events/s)
max	1446107 (ms)	7099 (events/s)

Table 10. The statistical summary of the observed latencies and throughput collected during the Extraction benchmark from Spark running on YARN with default configuration.

4.3.2.4 Spark YARN Max Executor Configuration

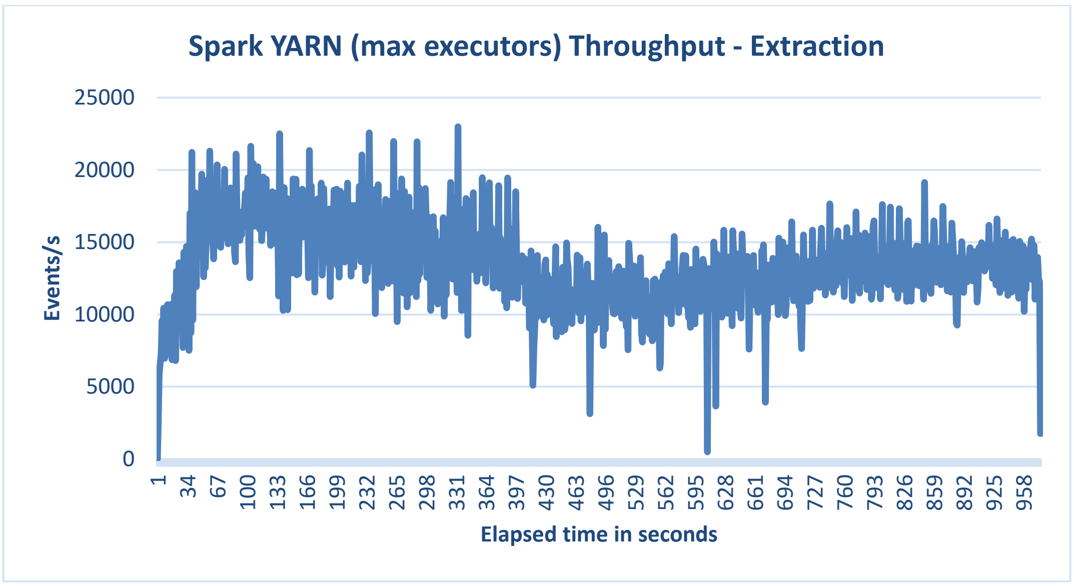


Figure 19. The throughput of Spark running on YARN with maximum requested executors for the Extraction benchmark.

Spark YARN (max executors) Throughput – Extraction Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	977 (seconds)
mean	155671.71113 (ms)	13433.67451 (events/s)
standard deviation	74724.56296 (ms)	2906.90378 (events/s)
min	1231 (ms)	0 (events/s)
max	254238 (ms)	22990 (events/s)

Table 11. The statistical summary of the observed latencies and throughput collected during the Extraction benchmark from Spark running on YARN with maximum requested executors.

4.3.2.5 Storm

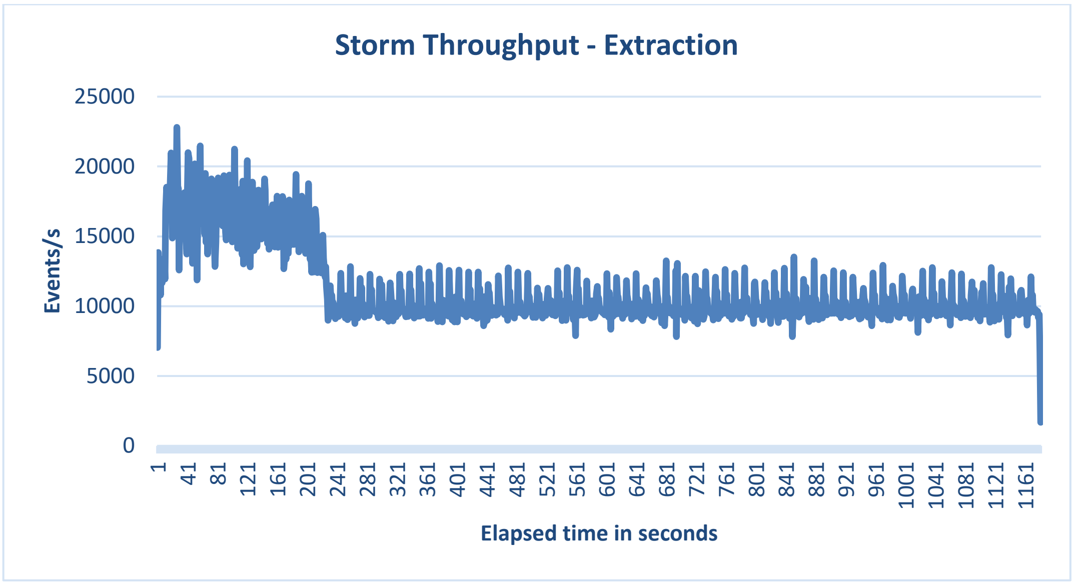


Figure 20. The throughput of Storm for the Extraction benchmark.

Storm – Extraction Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	1182 (seconds)
mean	51.812021 (ms)	11103.80711 (events/s)
standard deviation	101.25425 (ms)	2699.95988 (events/s)
min	2 (ms)	1683 (events/s)
max	1115 (ms)	22804 (events/s)

Table 12. The statistical summary of the observed latencies and throughput collected from Storm during the Extraction benchmark.

4.3.3 Grep

4.3.3.1 Flink

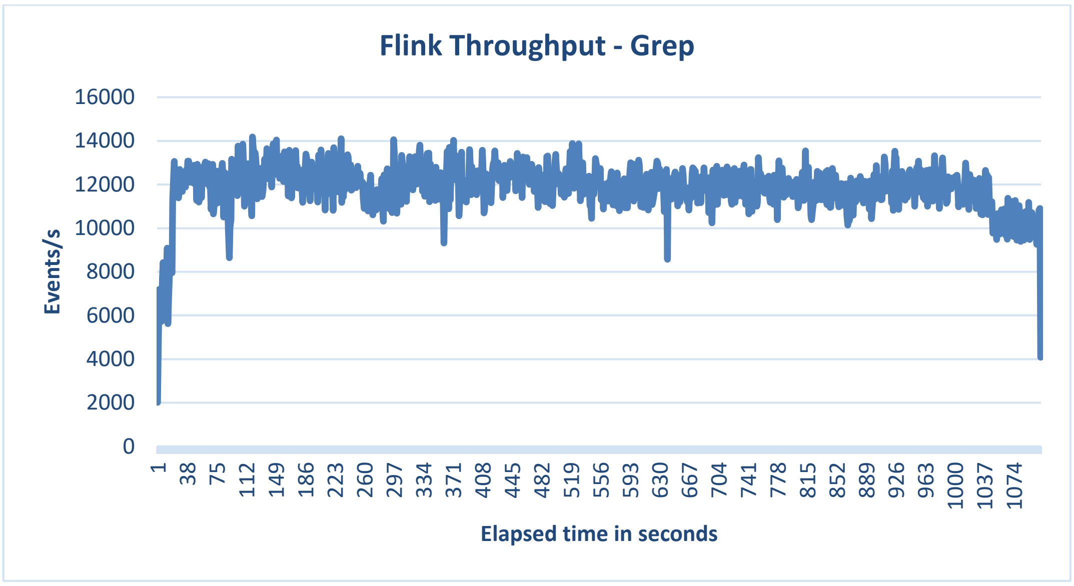


Figure 21. The throughput of Flink for the Grep benchmark.

Flink – Grep Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	1108 (seconds)
mean	286008.042914 (ms)	11845.39711 (records)
standard deviation	148620.72165 (ms)	1090.00641 (records)
min	30 (ms)	2015 (records)
max	576923 (ms)	14179 (records)

Table 13. The statistical summary of the observed latencies and throughput collected from Flink during the Grep benchmark.

4.3.3.2 Spark Standalone Cluster

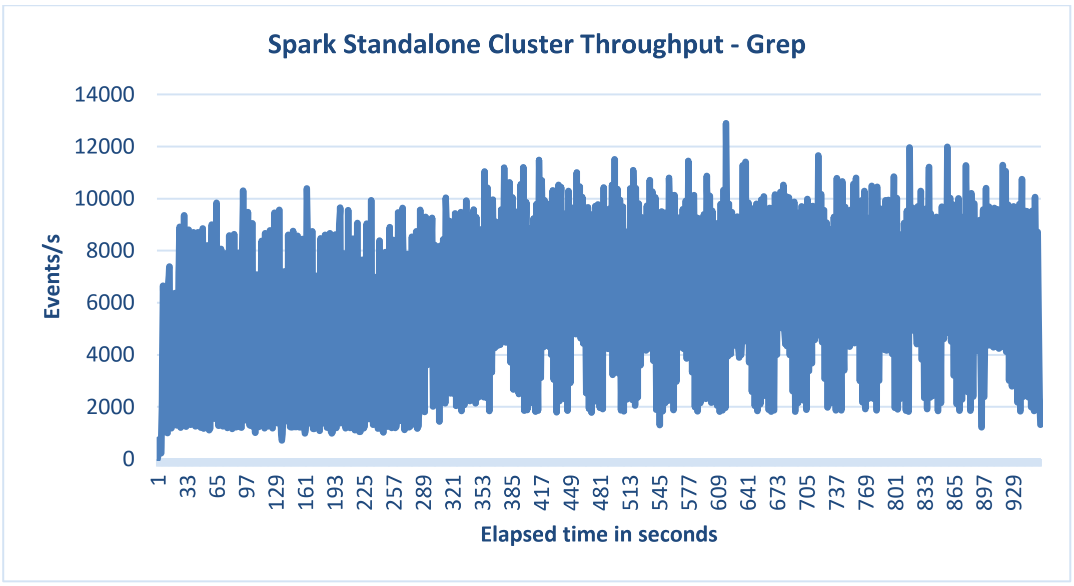


Figure 22. The throughput of Spark running in standalone cluster mode for the Grep benchmark.

Spark Standalone Cluster – Grep Benchmark

Summary	Latency Statistics	Throughput Statistics
count	5455689 (events)	959 (events/s)
mean	385411.54713 (ms)	5688.93535 (events/s)
standard deviation	178091.83558 (ms)	3211.756189 (events/s)
min	1247 (ms)	0 (events/s)
max	674593 (ms)	12891 (events/s)

Table 14. The statistical summary of the observed latencies and throughput collected from Spark (standalone) during the Grep benchmark

4.3.3.3 Spark YARN Default Configuration

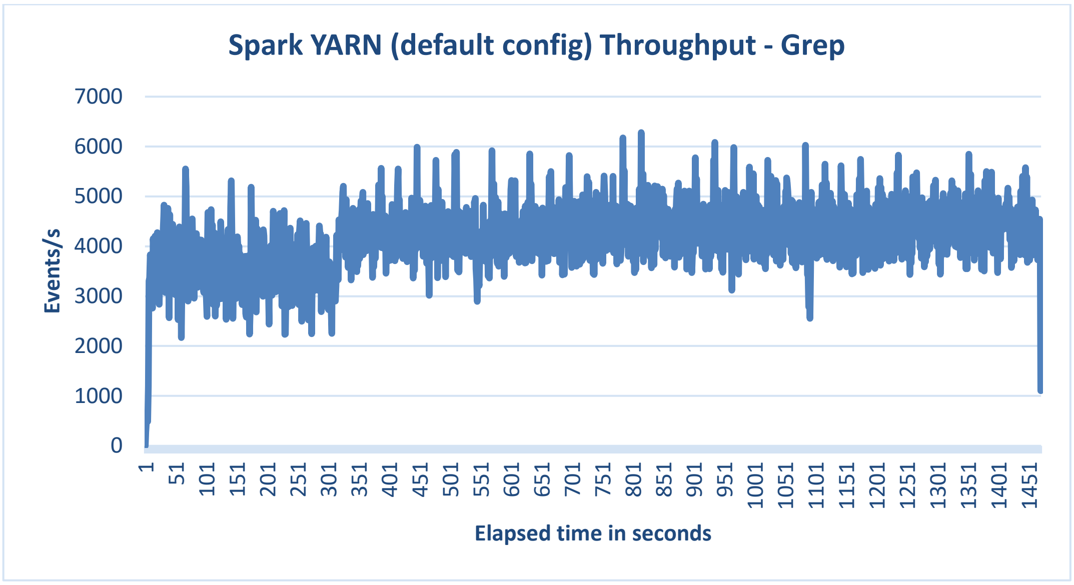


Figure 23. The throughput of Spark running on YARN with default configuration for the Grep benchmark.

Spark YARN (default config) Throughput – Grep Benchmark

Summary	Latency Statistics	Throughput Statistics
count	6129071 (events)	1471 (seconds)
mean	604132.30503 (ms)	4166.60163 (events/s)
standard deviation	324902.24041 (ms)	671.48590 (events/s)
min	1135 (ms)	0 (events/s)
max	1154017 (ms)	6281 (events/s)

Table 15. The statistical summary of the observed latencies and throughput collected during the Grep benchmark from Spark running on YARN with default configuration.

4.3.2.4 Spark YARN Max Executor Configuration

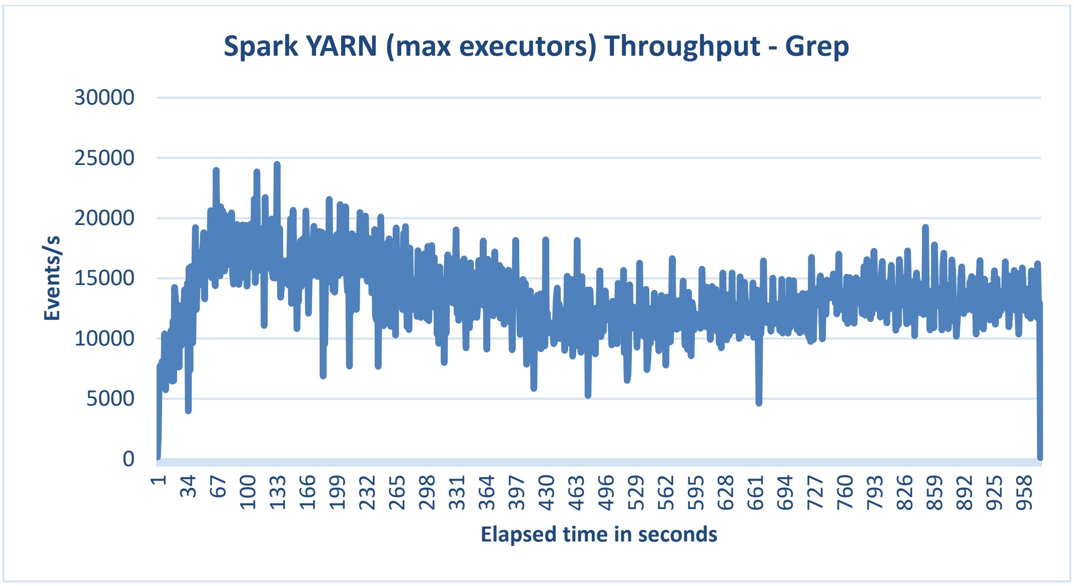


Figure 24. The throughput of Spark running on YARN with maximum requested executors for the Grep benchmark.

Spark YARN (max executors) Throughput – Grep Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	977 (seconds)
mean	152515.67566 (ms)	13433.67451 (events/s)
standard deviation	75051.73238 (ms)	2924.32218 (events/s)
min	478 (ms)	112 (events/s)
max	251879 (ms)	24472 (events/s)

Table 16. The statistical summary of the observed latencies and throughput collected during the Grep benchmark from Spark running on YARN with maximum requested executors.

4.3.2.5 Storm

When executing the Grep benchmark for Storm, some unusual behaviour was observed. Just past the midway point of the experiment, the latency increased dramatically, while the throughput decreased. The KafkaSpout then began reporting numerous failures (approximately 500,000). After several minutes, the problem resolved itself. This was initially assumed to be an issue with the cluster, so the experiment was rerun. Shortly into the new execution, a similar issue occurred. This one, however, did not resolve itself and resulted in an OutOfMemoryError being thrown, which ultimately ended the experiment.

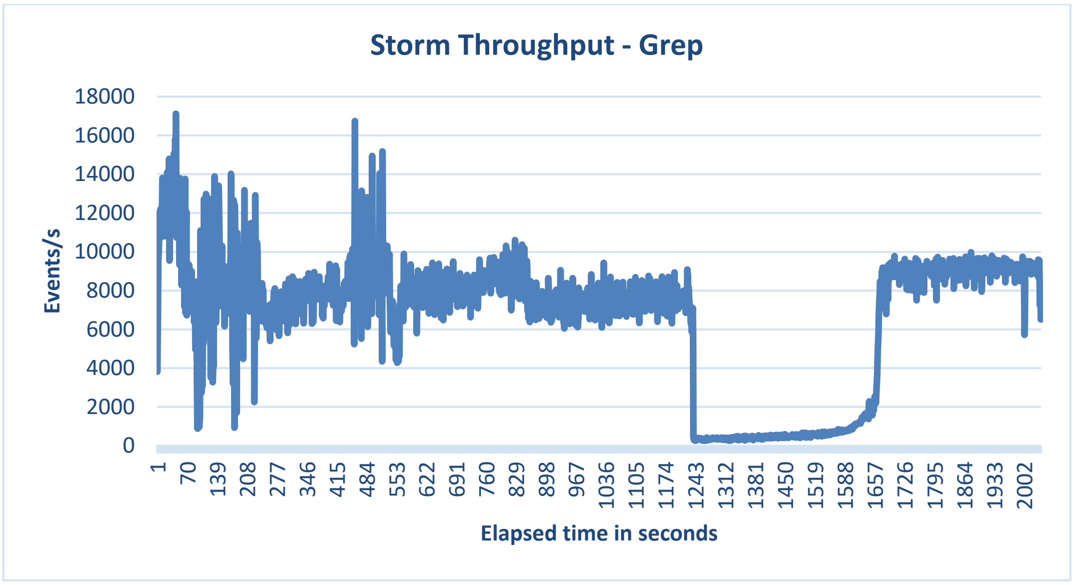


Figure 25. The throughput of Storm for the Grep benchmark.

The issue appeared to be related solely to the KafkaSpout. However, no additional configuration parameters had been passed to it during creation that might cause the problem. Furthermore, the Storm documentation[42] explicitly claims that the spout's default configuration has been shown to give good performance. From a quick search of a number of Storm forums, it quickly became apparent that many other users have experienced a similar issue.

---

[42] See http://storm.apache.org/releases/1.2.1/storm-kafka-client.html

Interestingly, the results show that by the time the experiment finished, more output events were received than input events. This is down to the way Storm guarantees message processing and handles errors related to messages[43]. When a tuple fails, depending on the configuration, Storm will retry that tuple. A tuple is considered failed when all of the messages that have originated from it fail to be fully processed within a specified timeout. This timeout, by default, is configured to 30 seconds. Therefore, what likely happened is that the sudden drop in throughput and increase in latency caused Storm to mark the messages during the period the issue occurred as failed, even though they were eventually delivered. This resulted in over half a million additional results being received.

Storm – Grep Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13679588 (events)	2041 (seconds)
mean	420701.02527 (ms)	6702.39490 (events/s)
standard deviation	508313.53364 (ms)	3450.19869 (events/s)
min	4 (ms)	256 (events/s)
max	1540202 (ms)	17118 (events/s)

Table 17. The statistical summary of the observed latencies and throughput collected from Storm during the Grep benchmark.

[43] See http://storm.apache.org/releases/1.2.1/Guaranteeing-message-processing.html

4.3.4 WordCount

4.3.4.1 Flink

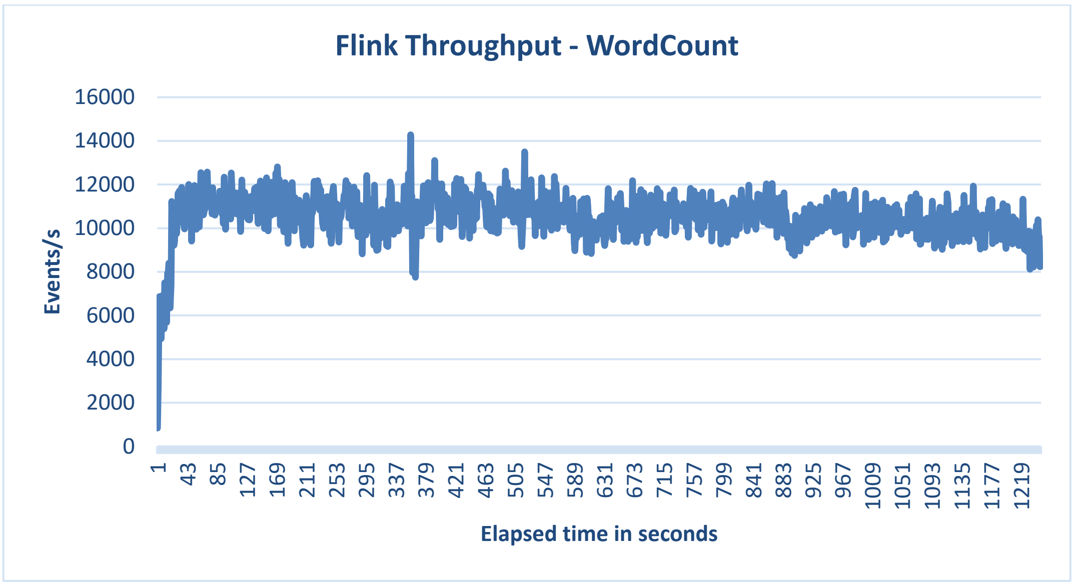


Figure 26. The throughput of Flink for the WordCount benchmark.

Flink – WordCount Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	1247 (seconds)
mean	381147.70771 (ms)	10525.02005 (records)
standard deviation	230201.81532 (ms)	980.68934 (records)
min	191 (ms)	838 (records)
max	786456 (ms)	14279 (records)

Table 18. The statistical summary of the observed latencies and throughput collected from Flink during the WordCount benchmark.

4.3.4.2 Spark Standalone Cluster

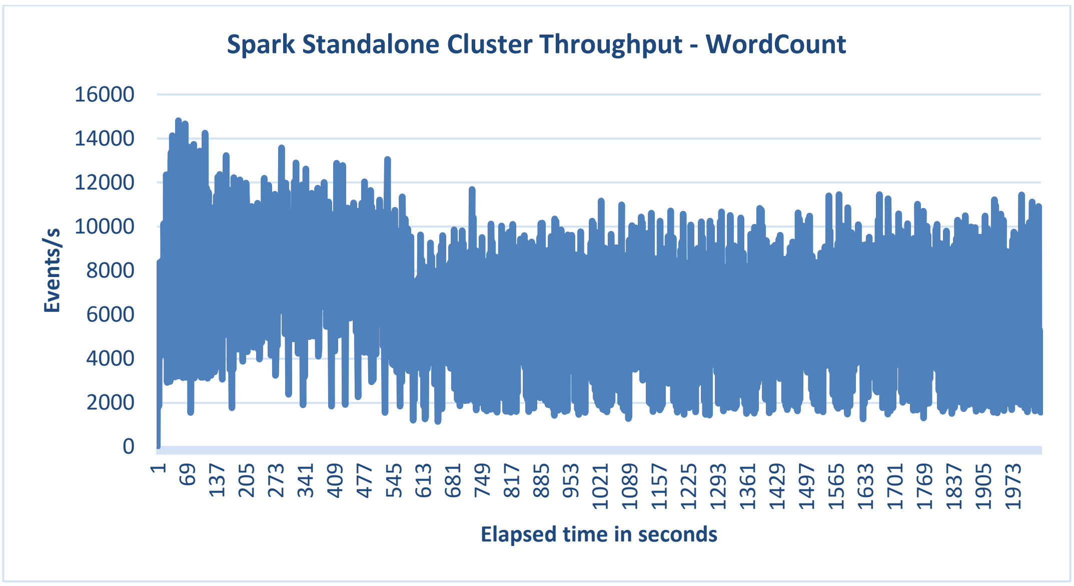


Figure 27. The throughput of Spark running in standalone cluster mode for the WordCount benchmark.

Spark Standalone Cluster – WordCount Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	2039 (events/s)
mean	657112.34765 (ms)	6436.83178 (events/s)
standard deviation	410260.88862 (ms)	2965.60391 (events/s)
min	858 (ms)	0 (events/s)
max	1389500 (ms)	14807 (events/s)

Table 19. The statistical summary of the observed latencies and throughput collected from Spark (standalone) during the WordCount benchmark

4.3.3.3 Spark YARN Default Configuration

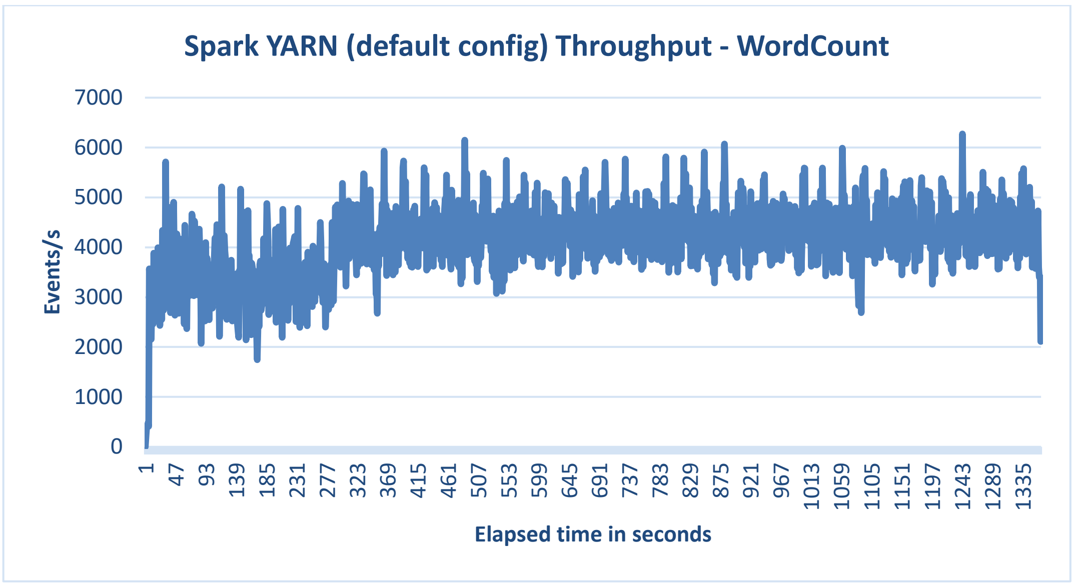


Figure 28. The throughput of Spark running on YARN with default configuration for the WordCount benchmark.

Spark YARN (default config) Throughput – WordCount Benchmark

Summary	Latency Statistics	Throughput Statistics
count	5590635 (events)	1363 (seconds)
mean	566854.65469 (ms)	4101.71313 (events/s)
standard deviation	299176.64160 (ms)	720.52493 (events/s)
min	1000 (ms)	0 (events/s)
max	1074834 (ms)	6273 (events/s)

Table 20. The statistical summary of the observed latencies and throughput collected during the WordCount benchmark from Spark running on YARN with default configuration.

4.3.2.4 Spark YARN Max Executor Configuration

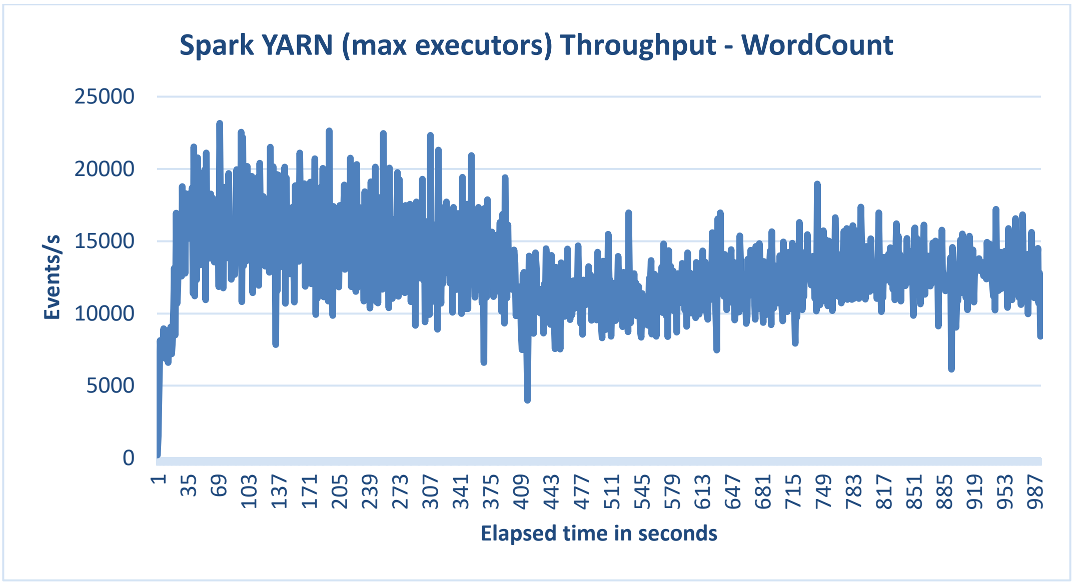


Figure 28. The throughput of Spark running on YARN with maximum requested executors for the WordCount benchmark.

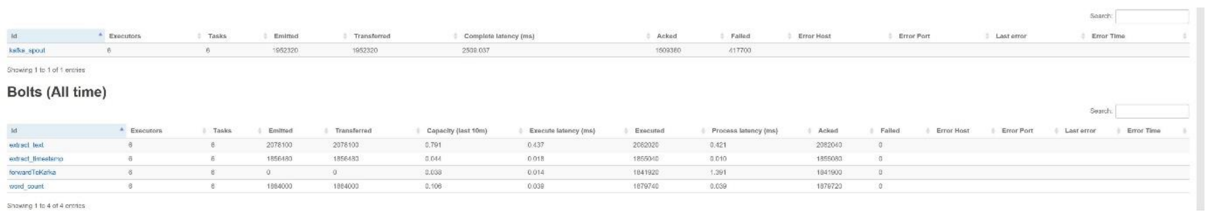
Spark YARN (max executors) Throughput – WordCount Benchmark

Summary	Latency Statistics	Throughput Statistics
count	13124700 (events)	994 (seconds)
mean	164852.14372 (ms)	13203.92354 (events/s)
standard deviation	79218.75152 (ms)	2884.82487 (events/s)
min	536 (ms)	206 (events/s)
max	274131 (ms)	23158 (events/s)

Table 21. The statistical summary of the observed latencies and throughput collected during the WordCount benchmark from Spark running on YARN with maximum requested executors.

4.3.2.5 Storm

As was the case with Storm's Grep benchmark, some unusual behaviour was seen surrounding the execution of WordCount. The Storm UI, figure 29, showed that the issue began relatively early in the process, with over 400,000 messages being marked as failed out of the first 2,000,000. For reference, the UI shows that no tuples have been emitted for the bolt responsible for forwarding the results to the Kafka output topic. The documentation is a little vague on this, but it appears the reason for this is because the emitted metric represents the number of times the emit() method is called on the *OutputCollector* class. This method is responsible for emitting new tuples to the default stream[44]. As these tuples are being forwarded to Kafka, rather than emitted to a stream for further processing, the emit() method is not called, resulting in the 0 evident in figure 29.



Search:

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
kafka_spout	6	6	1952320	1952320	2508.037	1509360	417700				

Showing 1 to 1 of 1 entries

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
extract_text	6	6	2078100	2078100	0.791	0.437	2062020	0.421	2082040	0				
extract_timestamp	6	6	1856480	1856480	0.044	0.018	1855040	0.010	1855080	0				
forwardToKafka	6	6	0	0	0.038	0.014	1841920	1.391	1841900	0				
word_count	6	6	1884000	1884000	0.106	0.039	1879740	0.039	1879720	0				

Showing 1 to 4 of 4 entries

Figure 29. Storm UI for the WordCount benchmark

[44] See https://storm.apache.org/releases/current/javadocs/index.html?org/apache/storm/task/OutputCollector.html

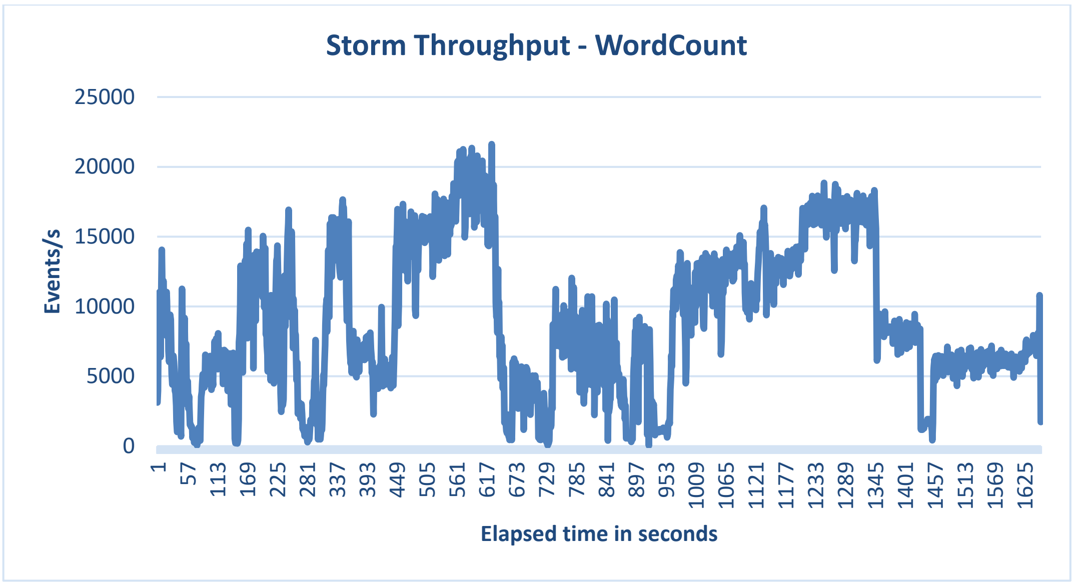


Figure 30. The throughput of Storm for the WordCount benchmark.

Storm – WordCount Benchmark

Summary	Latency Statistics	Throughput Statistics
count	15067104 (events)	1656 (seconds)
mean	408677.19104 (ms)	9098.49275 (events/s)
standard deviation	240276.15795 (ms)	5202.73881 (events/s)
min	6 (ms)	40 (events/s)
max	1169082 (ms)	21622 (events/s)

Table 22 The statistical summary of the observed latencies and throughput collected from Storm during the WordCount benchmark.

5 ANALYSIS

The purpose of this chapter is to analyse and discuss the results from the various experiments that were conducted. Two separate comparison groups have been identified. These are:

- Flink and Spark in Standalone Cluster mode
- Spark on YARN with maximum requested executors and Storm

While the experiments involving Spark on YARN using the default configuration were informative in their own right, it would not be objective to compare that configuration to any of the others. This is because of how limited the assigned resources were in comparison to the other setups. Had Flink on YARN worked as expected, it would have made an ideal candidate for a comparison. Both of the frameworks, in that case, could have been compared based off their default YARN configurations.

5.1 Flink and Spark Standalone

A comparison of means test was performed between Flink and Spark Standalone's mean latency values, as calculated for each of the benchmarking applications. The significance value was also calculated. In each case, a P-value of < 0.0001 was returned, indicating a highly significant difference between the two frameworks' latencies. A similar process was applied to the frameworks' throughput mean. Again, the results indicated that there is a highly significant difference between the two frameworks'. The below graphs and tables provide more details regarding the results. With regards to this study's research question, the measured values show that Flink outperforms Spark in terms of having both a lower latency and higher throughput.

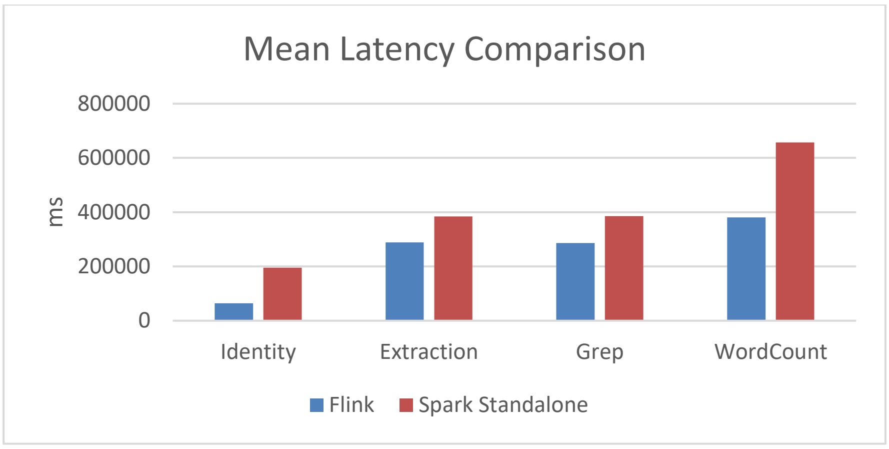


Figure 31. The mean latency of Flink and Spark Standalone for each benchmarking application.

Latency Summary

Results	Identity	Extraction	Grep	WordCount
difference	130528.822	95085.833	99403.504	275964.640
standard Error	57.129	80.101	80.407	129.853
95% CI	130416.8520 to 130640.7928	94928.8375 to 95242.8278	99245.9104 to 99561.0981	275710.1323 to 276219.1476
t-statistic	2284.816	1187.074	1236.262	2125.204
DF	18617783	19019870	18580387	26249398
significance level	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Table 23. Difference between the observed latency means for Flink and Spark Standalone. Also includes a significance value and 95% Confidence Interval.

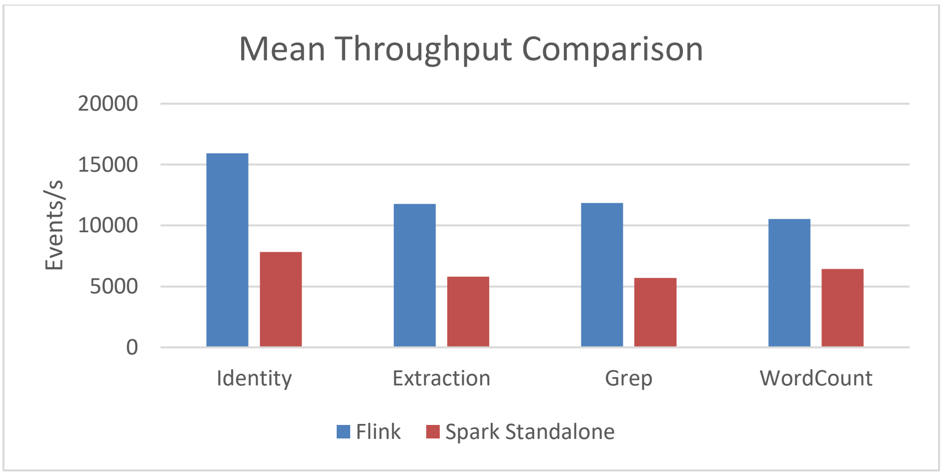


Figure 32. The mean throughput of Flink and Spark Standalone for each benchmarking application.

Throughput Summary

Results	Identity	Extraction	Grep	WordCount
difference	-8103.125	-5946.705	-6156.462	-4088.188
standard Error	226.155	97.695	102.704	86.748
95% CI	-8546.7325 to -7659.5178	-6138.2933 to -5755.1162	-6357.8768 to -5955.0467	-4258.2737 to -3918.1029
t-statistic	-35.830	-60.870	-59.943	-47.127
DF	1524	2128	2065	3284
significance level	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Table 24. Difference between the observed throughput means for Flink and Spark Standalone. Also includes a significance value and 95% Confidence Interval.

5.2 Storm and Spark on YARN

As was the case with Flink and Spark Standalone, a comparison of means test was performed between Storm and Spark's mean latency and throughput values. The significance value, again, was also calculated. In each case, a P-value of < 0.0001 was returned, indicating a highly significant difference between the two frameworks' latencies and throughputs. The latency for Storm was incredibly low for the Identity and Extraction benchmarks (so much so that the values do not appear on the chart below). The high latency seen in Grep and Workload was discussed previously in Chapter 4. In contrast, Spark on YARN's throughput was, in all cases, significantly higher than Storm's.

In terms of the research question, the results show that Spark performs better in terms of throughput then Storm. The issue of latency is more complicated. It is difficult to give a definitive answer as to which framework performed better. It is likely that there is a fix or workaround for the bug experienced with the *KafkaSpout*, but given the limitations of this study, it was not possible to investigate the issue further. For now, the results show that Storm's latency is significantly lower than Spark's, but there is the possibility that the value could spike under certain circumstances.

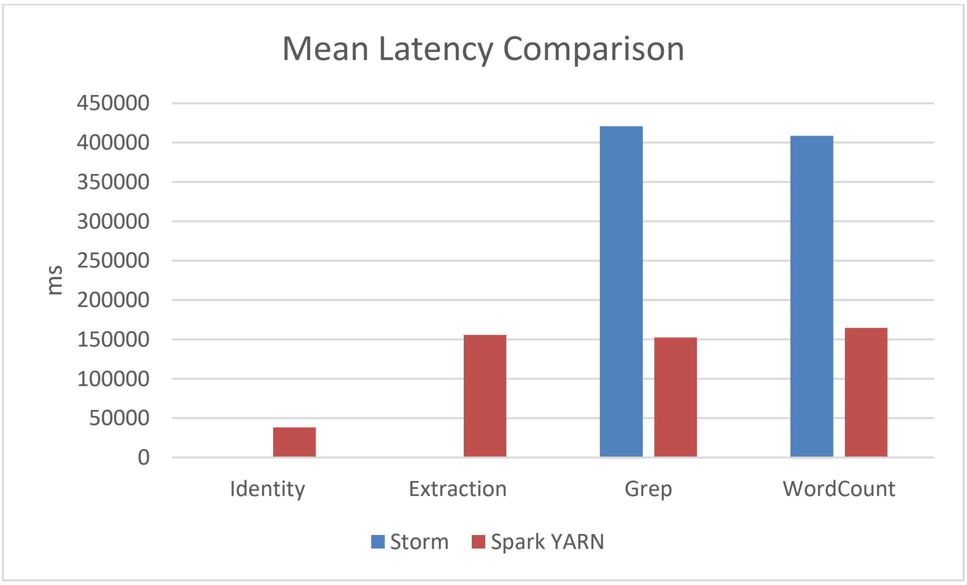


Figure 32. The mean latency of Storm and Spark on YARN for each benchmarking application.

Latency Summary

Results	Identity	Extraction	Grep	WordCount
difference	38051.597	155619.899	-268185.350	-243825.047
standard Error	5.508	20.626	141.769	69.392
95% CI	38040.8018 to 38062.3931	155579.4725 to 155660.3257	-268463.2122 to -267907.4870	-243961.0537 to -243689.0410
t-statistic	6908.327	7544.771	-1891.704	-3513.720
DF	26249398	26249398	26804286	28191802
significance level	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Table 25. Difference between the observed latency means for Storm and Spark on YARN. Also includes a significance value and 95% Confidence Interval.

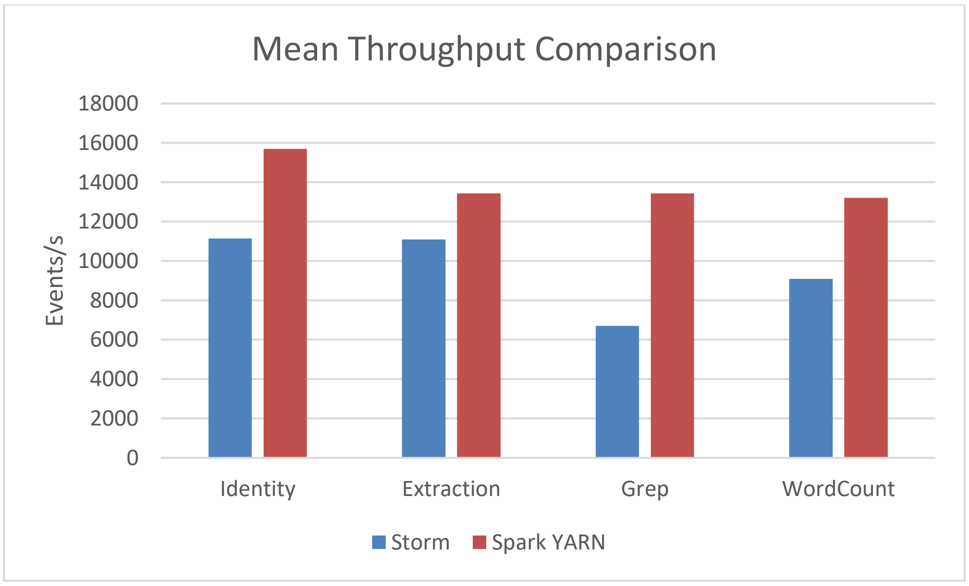


Figure 33. The mean throughput of Storm and Spark on YARN for each benchmarking application.

Throughput Summary

Results	Identity	Extraction	Grep	WordCount
difference	4557.891	2329.867	6731.280	4105.431
standard Error	153.754	120.873	127.963	179.611
95% CI	4256.3578 to 4859.4239	2092.8277 to 2566.9071	6480.3752 to 6982.1840	3753.2381 to 4457.6235
t-statistic	29.644	19.275	52.603	22.857
DF	2012	2157	3016	2648
significance level	P < 0.0001	P < 0.0001	P < 0.0001	P < 0.0001

Table 26. Difference between the observed throughput means for Storm and Spark on YARN. Also includes a significance value and 95% Confidence Interval.

6 CONCLUSION

6.1 Research Overview and Problem Definition

This research project examined the performance of various open source stream processing frameworks in terms of throughput and latency. Its goal was to answer the following research question:

Does Apache Spark perform better than Apache Flink, Apache Samza, and Apache Storm in terms of throughput and latency when processing streamed data?

In order to answer this question, a benchmarking suite was defined and implemented that used a novel approach for capturing the required metrics. In total, 20 experiments were executed across 5 different framework/configuration combinations. These consisted of:

- Apache Flink Standalone Cluster mode
- Apache Spark Standalone Cluster mode
- Apache Spark running on YARN with default configuration
- Apache Spark running on YARN with maximum requested executors
- Apache Storm

The experiments were conducted on a virtual environment provided by Amazon Web Services. The environment consisted of six t2.large node type instances, grouped into two three-node clusters. One cluster was responsible for running Apache Kafka, while the other ran the stream processing frameworks. This setup was chosen because it not only mirrored production-like environments, but it also allowed for the independent collection of metrics, thus preventing any undue strain from being put on the computational nodes

6.2 Findings

The results of the experiments show quite clearly that Flink, when running in standalone cluster mode, performs significantly better than Spark, running in its own standalone cluster mode. This is the case for both latency and throughput. This conclusion supports Hesse and Lorenzo's assertion that Flink's latency, in general, is relatively low, while its throughput is high (Hesse & Lorenz, 2015, p. 799). With that said, it is difficult to compare the findings of the study in this case to other works. This is because, firstly, no other academic study was found during the literature review that contained figures for Flink's performance and, secondly, the approach taken in this study for measuring latency and throughput differs to that seen in other works. This makes it difficult to conduct a comparative analysis of findings.

With regards to Spark and Storm, the results show that Storm's latency is many magnitudes of order lower than Spark's. This finding, in general, supports both Lu et al and Qian et al's assertions (Lu, Wu, Xie, & Hu, 2014, p. 74; Qian, Wu, Huang, & Das, 2016, p. 596). The issue related to the latency and throughput for Storm's Grep and WordCount benchmarks undeniably mars the final results slightly. With that said, if a reader of this study becomes aware of the issue before deploying Storm to production, then it will have been worthwhile.

6.3 Limitations

There are a number of limitations with this study. The first of these relates to the cluster size used for the experiments. While a six-node cluster is not necessarily small, it is by no means anywhere near the maximum number of nodes that these frameworks can execute on. Unfortunately, running the experiments on a larger cluster was beyond the scope of this work, but readers should be aware that the setup used here is definitely not pushing the frameworks to their limits.

Another limitation relates to the implementation of the benchmarking suite. While every effort was made to conform to best practices when coding the experiments, each framework has its own nuances and quirks that can only be learned after a substantial period of time. Looking at various forums related to the frameworks highlighted this fact. For example, retrieving the timestamp from Flink proved to be more troublesome than initially expected. It was only after many hours of investigation that the method that should be taken became apparent. Extra care was taken to highlight these details within this study. By doing so, it is hoped that any other researchers or concerned parties who want to implement elements of this study, can do so in as straightforward a manner as possible.

Given the range of configuration options available within each framework, the scope of this study is relatively small, in that it only tested 5 different configurations. Again, it was beyond the scope of this study to attempt to execute any more than was done initially. Readers need to be aware that this study and its results pertain only to the configurations specified in Chapter 4. As an example of how fine-grained users can get with these frameworks, Storm allows developers to specify programmatically the parallelism of individual spouts and bolts within a topology. In production environments, many hours might be dedicated to finetuning individual applications. The resulting configuration may only be applicable for that particular topology executing on that particular cluster. This, thus, highlights a significant flaw in benchmarking studies. The results cannot be easily generalised to fit other use cases.

6.4 Contributions

With the above limitations clarified, there are several contributions that are made by this study. The first of these relates to the benchmarking suite design. The approach for measuring latency in this study differs from those other benchmarking studies that were analysed. Whereas those studies measure latency from when the message is ingested into the framework, this study does so at the point were the message is added to the Kafka topic. This is done to ensure that a true value is captured for batch processing systems. Interestingly, Lu et al's study measured similar latency for Spark in the Identity benchmark as was observed in the Spark UI for Identity in this study. This, however, is not an accurate representation of latency, as the message, as was seen in this study, could have been sitting in the Kafka topic for many minutes before being consumed (Lu, Wu, Xie, & Hu, 2014, p. 75).

Another contribution relates to the results that were presented between Spark and Flink in local cluster mode and Spark on YARN and Storm. These results give a general overview as to how these frameworks are likely to perform on clusters similar to the one used in this study. The Flink and Spark comparison is particularly useful, because both frameworks were in their standalone modes using out-of-the-box configuration values. The results of these benchmarks might be useful for small to medium size business who are considering setting up a smallscale stream processing system. The configuration and implementation details mentioned in Chapter 4, should also help make that process easier.

6.4 Future Work

Given the small cluster size used in the experiments, it is recommended that the benchmarking applications defined in this study be executed on a larger cluster with more custom-tailored configuration. Coupled with this, additional benchmarking applications could be defined in order to expand the suite of executed tests.

Another possible area of future work relates to the issue of fault tolerance. This study did not factor in the possibility of failing messages. As was seen in the results, several of the frameworks either returned too few results at the end of processing or else returned too many. The benchmarking suite defined here could benefit from having a result validation module added to it, which is able to precisely identify how many records were lost or retried.

Finally, two of the framework configurations that were intended to be tested for this study were not. They were Flink on YARN and Samza. It is likely that the issues that prevented these frameworks from being added to the benchmarking suite will be resolved in the near future. Both Flink and Samza are gaining significant traction, so there is a need to see some definitive values around the performance potential of these two frameworks.

BIBLIOGRAPHY

Acharjya, D. P., & P, K. A. (2016). A Survey on Big Data Analytics: Challenges, Open Research Issues and Tools. International Journal of Advanced Computer Science and Applications, 7(2), 511-518. doi:10.14569/IJACSA.2016.070267.

Anand, G., & Kodali, R. (2008). Benchmarking the benchmarking models. Benchmarking: An International Journal, 15(3), 257-291. doi:10.1108/14635770810876593.

Anis Uddin Nasir, M., De Francisci Morales, G., Garcia-Soriano, D., Kourtellis, N., & Serafini, M. (2015). The power of both choices: Practical load balancing for distributed stream processing engines. Paper presented at the 31st International Conference on Data Engineering, 137-148. doi:10.1109/ICDE.2015.7113279.

Axelrod, A. (2015). Box: Natural language processing research using amazon web services. The Prague Bulletin of Mathematical Linguistics, 104(1), 27-38. 10.1515/pralin-2015-0011.

Chandramouli, B., Goldstein, J., Barga, R., Riedewald, M., & Santos, I. (2011). Accurate latency estimation in a distributed event processing system. Paper presented at the 27th International Conference on Data Engineering, 255-266. doi:10.1109/ICDE.2011.5767926.

Coles, C. (2018). Overview of Cloud Market in 2017 and Beyond. Available from: https://www.skyhighnetworks.com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws/. Accessed 12/08/2018.

De Matteis, T., & Mencagli, G. (2016). Proactive elasticity and energy awareness in data stream processing. Journal of Systems and Software, 127, 302. doi:10.1016/j.jss.2016.08.037.

Feng, T., Zhuang, Z., Pan, Y., & Ramachandra, H. (2015). A memory capacity model for high performing data-filtering applications in samza framework. Paper presented at the 2015 IEEE International Conference on Big Data, 2600-2605. doi:10.1109/BigData.2015.7364058.

Heidrich, J., Trendowicz, A., & Ebert, C. (2016). Exploiting big data's benefits. IEEE Software, 33(4), 111-116. doi:10.1109/MS.2016.99.

Hesse, G. & Lorenz, M. (2015). Conceptual Survey on Data Stream Processing Systems. Paper presented at the 21st International Conference on Parallel and Distributed Systems, 797-802. doi:10.1109/ICPADS.2015.106.

Huang, W., Meng, L., Zhang, D., & Zhang, W. (2017). In-memory parallel processing of massive remotely sensed data using an apache spark on Hadoop YARN model. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 10(1), 3-19. doi:10.1109/JSTARS.2016.2547020.

Imai, S., Patterson, S., & Varela, C. A. (2017). Maximum sustainable throughput prediction for data stream processing over public clouds. Paper presented at the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 504-513. doi:10.1109/CCGRID.2017.105

Jiamin, L., & Jun, F. (2014). A survey of mapreduce based parallel processing technologies. China Communications, 11(14), 146-155. doi:10.1109/CC.2014.7085615.

Kambatla, K., Kollias, G., Kumar, V., & Grama, A. (2014). Trends in big data analytics. Journal of Parallel and Distributed Computing, 74(7), 2561-2573. doi:10.1016/j.jpdc.2014.01.003.

Kiran, M., Murphy, P., Monga, I., Dugan, J., & Baveja, S. (2015). Lambda Architecture for Cost-effective Batch and Speed Big Data processing. Paper presented at the 2015 IEEE International Conference on Big Data, 2785-2792.

Landset, S., Khoshgoftaar, T. M., Richter, A. N., & Hasanin, T. (2015). A survey of open source tools for machine learning with big data in the hadoop ecosystem. Journal of Big Data, 2(1), 1-36. doi:10.1186/s40537-015-0032-1.

Li, H., Wu, J., Jiang, Z., Li, X., & Wei, X. (2017). Minimum backups for stream processing with recovery latency guarantees. IEEE Transactions on Reliability, 66(3), 783-794. doi:10.1109/TR.2017.2712563.

Lu, R., Wu, G., Xie, B., & Hu, J. (2014). Stream bench: Towards benchmarking modern distributed stream computing frameworks. Paper presented at the 7th International Conference on Utility and Cloud Computing, 69-78. 10.1109/UCC.2014.15.

Maarala, A. I., Rautiainen, M., Salmi, M., Pirttikangas, S., & Riekki, J. (2015). Low Latency Analytics for Streaming Traffic Data with Apache Spark. Paper presented at the 2015 IEEE International Conference on Big Data, 2855-2858. doi:10.1109/BigData.2015.7364101.

Mendes, M. R. N., Bizarro, P., & Marques, P. (2013). Towards a standard event processing benchmark. Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering, 307-310. doi: 10.1145/2479871.2479913.

Morshed, S. J., Rana, J., & Milrad, M. (2016). Open source initiatives and frameworks addressing distributed real-time data analytics. Paper presented at the 2016 International Parallel and Distributed Processing Symposium Workshops, 1481-1484. doi:10.1109/IPDPSW.2016.152.

Otsuka, E., Wallace, S. A., & Chiu, D. (2016). A hashtag recommendation system for twitter data streams. Computational Social Networks, 3(1), 1-26. 10.1186/s40649-016-0028-9.

Qian, S., Wu, G., Huang, J., & Das, T. (2016). Benchmarking modern distributed streaming platforms. Paper presented at the International Conference on Industrial Technology, 592-598. doi:10.1109/ICIT.2016.7474816.

Pääkkönen, P. (2016). Feasibility analysis of AsterixDB and spark streaming with cassandra for stream-based processing. Journal of Big Data, 3(1), 1-25. 10.1186/s40537-016-0041-8.

Riccomini, C. 2013. Apache Samza: LinkedIn's Real-time Stream Processing Framework. Available from: https://engineering.linkedin.com/data-streams/apache-samza-linkedins-real-time-stream-processing-framework. Accessed 07/12/2017.

Sakr, S. (2017). Big data processing stacks. IT Professional Magazine, 19(1), 34-41. doi:10.1109/MITP.2017.6.

Serrano, N., Gallardo, G., & Hernantes, J. (2015). Infrastructure as a service and cloud technologies. IEEE Software, 32(2), 30-36. 10.1109/MS.2015.43.

Singh, D., & Reddy, C. K. (2015). A Survey on Platforms for Big Data Analytics. Journal of Big Data, 2(1) doi:10.1186/s40537-014- 0008-6.

Shahrivari, S. (2014). Beyond batch processing: Towards real-time and streaming big data. Computers, 3(4), 117-129. 10.3390/computers3040117.

Solaimani, M., Iftekhar, M., Khan, L., Thuraisingham, B., & Ingram, J. B. (2014). Spark-Based Anomaly Detection over Multi-Source VMware Performance Data in Real-Time. Paper presented at the 2014 IEEE Symposium on Computational Intelligence in Cyber Security, 1-8. doi:10.1109/CICYBS.2014.7013369.

Wang, M., Liu, J., & Zhou, W. (2016). Design and implementation of a high-performance stream-oriented big data processing system. Paper presented at the 8th International Conference on Intelligent Human-Machine Systems and Cybernetics, 1 363-368. doi:10.1109/IHMSC.2016.64

Yadranjiaghdam, B., Yasrobi, S., & Tabrizi, N. (2017). Developing a real-time data analytics framework for twitter streaming data. Paper presented at the IEEE International Congress on Big Data. 329-336. doi:10.1109/BigDataCongress.2017.49.

Zhuang, Z., Feng, T., Pan, Y., Ramachandra, H., & Sridharan, B. (2016). Effective multi-stream joining in apache samza framework. Paper presented at the 267-274. doi:10.1109/BigDataCongress.2016.41.