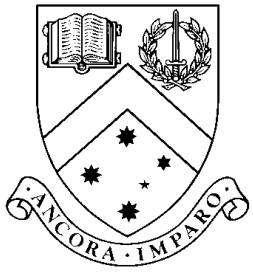
An Evaluation of Data Stream Processing Systems for Monitoring Heavy-Haul Railway Systems



Jonathan Poltak Samosir
Dr Maria Indrawan-Santiago
Dr Pari Delir Haghighi

Clayton School of Information Technology

Monash University

This dissertation is submitted for the degree of

Honours Degree of Bachelor of Computer Science

17 June 2015

I would like to dedicate this thesis to my mother who raised my sister and I single-handedly from birth, providing us with an environment to grow and learn about the world.

Declaration

I hereby declare that this thesis contains no material that has been accepted for the award of any other degree or diploma in any university or other institution. I affirm that, to the best of my knowledge, this thesis contains no material previously published or written by any other person, except where due reference is made in the text.

Jonathan Poltak Samosir
17 June 2015

Acknowledgements

I would like to acknowledge and express my gratitude to my supervisors, Dr Maria Indrawan-Santiago and Dr Pari Delir Haghighi, who have assisted and supported me throughout the duration of this research, providing great brains for me to bounce ideas off and providing me with much needed criticism on my work. The Faculty of Information Technology, at Monash University, have also provided much resources and support, for which I am greatly appreciative. I also would like to acknowledge Minh Quan Ngo and Weng Long Pang for their support in providing support, in the forms of proofreading and editing, for this thesis.

The shared office space provided by both the Faculty of Information Technology at Clayton and Caulfield campuses of Monash University has been an essential resource for me throughout the duration of this research and the writing of this thesis. This is also greatly appreciated, allowing me to have a space to work, think, and play.

Lastly, I would like to thank my friends at Monash University, also researching and writing their theses, for keeping me sane. Particularly, Minh Quan "Romy" Ngo, Jian Loong Liew, and Weng Long Pang.

Jonathan Poltak Samosir
17 June 2015

Abstract

Data processing, using big data technologies, is quickly becoming more commonplace in data-focused applications. In the Monash University Institute of Railway Technology, a project is being attempted which focuses on an existing railway project in the Pilbara region of Western Australia. This project is looking at the integration of big data processing technologies for use in processing readings from sensors monitoring the railway track conditions. Given planned future improvements in cellular networks in the Pilbara, the project team are also considering the use of realtime data steam processing technologies for the pre-processing, filtering, and live processing of the sensor readings acquired from the railways. These readings can be used to detect and monitor both track conditions and train car conditions, allowing abnormalities to be detected.

As many different realtime data stream processing technologies are becoming more mature and readily available, this study looks at forming a recommendation of the most suitable realtime data stream processing technology for use in the Monash University Institute of Railway Technology project. Candidate technologies include Storm, Samza, and Spark Streaming. To form this technology recommendation, a prototype pipeline will be designed and implemented in each of the technologies, performing similar processing methods that are intended to be used in the Institute of Railway Technology project.

Through the testing and evaluation of each candidate technology, using both quantitative and qualitative metrics, the study found that Storm is the most suitable technology for the case-study at hand. Storm has shown to exhibit positive performance, and processing features that are of great interest to the project.

The implications of this recommendation is that Storm will likely be used to implement a realtime data filtering pipeline for use in the overall railway project. Furthermore, the prototype pipeline, designed and implemented as part of this study, will be used as a baseline for the pipeline to be used in production in the project.

Further research is recommended to assess the scalability of Storm on differently sized and configured clusters. Furthermore, future research relating to Storm's ability at handling batch data processing would also be of much interest to the Monash University Institute of Railway Technology.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Currently, as a society, we are generating very large amounts of data from a large range of different sources. These sources include scientific experiments, such as the Australian Synchrotron [8] and The Large Hadron Collider [12], companies, such as Amazon [5], and also data generated by end users of products, such as social networks. The rate of data that is being generated is constantly increasing, presenting major challenges when it comes to the storage and processing of that data [29]. This is what is often referred to now as "Big Data". A further big data project, that will be looked into as the basis of the project outlined in this thesis, is the automated monitoring of railway tracks and cars by the Institute of Railway Technology at Monash University (IRT) [11].

Out of all of these data that we are faced with in such projects, often only specific parts of the data are of particular use for given purposes. Hence, rather than attempting to store all the new data that is being generated, an increasingly popular method of dealing with such data, in both academia and industry associated with big data, is the processing and analysis of data in realtime as it is received, before either being stored or sent on for further processing.

There are currently numerous realtime data processing systems that are in development and in production use, both in industry and academia. Examples of these realtime data processing frameworks include the widely used Storm project [3], developed at BackType and Twitter, Inc., and also the up-and-coming Spark Streaming project [14], developed at UC Berkeley's AMPLab [6], both of which are open-source projects. While there are a growing number of these projects being developed, often these projects are designed with a particular type of processing in mind. For example, the before mentioned Spark Streaming project, along with its mother project, Spark [7], was originally designed for highly parallelisable data with the use-case in mind of processing data in-memory using highly iterative machine learning algorithms related to data analytics [67].

What is proposed in this thesis is a realtime big data processing pipeline for the aforementioned railway monitoring system by the IRT at Monash University. This processing pipeline will allow data to be streamed from the railway and processed, in realtime, according to the team's given processing requirements. Multiple technologies exist in which this pipeline can be implemented, so a technology recommendation will be formed through the testing and evaluation of different implementations in different technologies.

This chapter will be structured as follows:
A brief outline of the big data processing domain, both batch and realtime processing, will be given in §1.1. Furthermore, an overview of the Monash University Institute of Railway Technology's project, upon which this project is based on, will be covered in the aforementioned section. In §1.2, the aims of the entire project will be given, as well as more specific goals and aims relating to the sub-project covered in this thesis. Research questions for this sub-project will be given in §1.3. Finally, the structure of this overall thesis will be given in §1.4, before concluding this chapter in §1.5.

1.1 Research Context

1.1.1 Big Data

Big data, as explained previously, is becoming commonplace in both industry and academia. Everyday companies are finding that they are generating large volumes of data and that their traditional relational database management system (RDMBS) solutions cannot scale to the epic proportions needed to handle this data in an efficient and robust manner [68]. Hence, companies and academics alike have started looking at alternative solutions designed with the goal of handling these massive datasets.

The most popular solution for this problem, up until recently, has been the MapReduce model of programming along with some type of scalable distributed storage system [27]. The MapReduce model was started at Google, Inc. with their own proprietary implementation along with their proprietary distributed file system, known as the Google File System (GFS) [48]. The use of this solution at Google allowed the company to easily handle all the data that was coming into their servers, including those related to Google Search, and perform the necessary processing operations that was needed at the time [43; 48].

1.1.2 Batch Data Processing

From the success of MapReduce usage combined with GFS, at Google, the open-source community responded swiftly with the development of the Apache Hadoop framework[1]. Hadoop originally offered an open-source implementation of MapReduce and their own open-source distributed file system known as the Hadoop Distributed File System (HDFS) [78].

Hadoop soon became the subject of mass-adoption in both industry and academia, being deployed at a fast rate. Development of the Hadoop framework also grew at a fast rate, with new applications related to HDFS and MapReduce being built on top of Hadoop, greatly benefiting the ecosystem as a whole. Some of these applications grew into widely adopted systems in their own right. For example, Hadoop applications such as Apache Pig [47] and Hive [90] allow for easy querying and manipulation of data stored on HDFS, both coming with the addition of their own query languages [74].

As further non-MapReduce model applications became of interest to the Hadoop community, Hadoop soon developed a further abstraction on top of the underlying resources (in most cases, HDFS). The goal of this was to facilitate the development and deployment of many different applications, varying in use-case, which could be run on the Hadoop ecosystem, without forcing developers to fit their application into the MapReduce model. This development was known as Apache Hadoop YARN: Yet Another Resource Negotiator, which can be thought of as an operating system-like abstraction sitting atop of the available Hadoop resources [94]. The abstraction provided by YARN facilitated the development of much more advanced, and non-MapReduce technologies which have since become widely used parts of the Hadoop ecosystem [54].

1.1.3 Realtime Data Processing

One of the major limitations of Hadoop, and the MapReduce model in general, soon became obvious: MapReduce was designed with the goal of being able to process batches of data, hence, given Hadoop's dominance, batched data processing was the focal point of the entire distributed data processing domain [61]. Essentially, batched data processing is where data gets collected first into large enough batches before being processed all-at-once. The point of processing in such a way is so there would be less overheads than attempting to process each individual datum as it arrives. For a lot of use-cases this was, and still is, fine as there were no other drawbacks apart from a high level of latency between the stages of when the data arrives and when it gets processed. However, for other applications, such as stock trading,

[1] https://hadoop.apache.org

sensor monitoring, and web traffic processing, a more low-latency, realtime solution was needed [61].

Soon, many solutions, with different use-cases and design goals, were developed in the area of distributed stream processing systems (DSPS). Given the Hadoop ecosystem that was already widely adopted, most of these DSPSs were built upon the still new YARN layer, ensuring overall compatibility with the Hadoop ecosystem, and the underlying HDFS. Some examples of such projects include the beforementioned Apache Storm, currently being used at Twitter, Inc. [92], among many other companies. Also up-and-coming projects, such as Apache Samza which is a recently open-sourced project, currently being used in production at LinkedIn Corporation [13].

1.1.4 Monash IRT Railway Project

The railway project that has been developed at Monash University's Institute of Railway Technology[2] uses numerous sensor technologies on certain train cars, such as the Track Geometry Recording Car (TGRC) and the Instrumented Ore Car (IOC), to monitor railway track conditions and detect track abnormalities [39; 40]. These train cars operate in the Pilbara region of Western Australia, continuously performing round trips from a port to a given loading point, where they are loaded with recently mined minerals and ores.

As is currently the case, data is received and processed using batch data processing technologies. Due to limited coverage of cellular networks in the Pilbara region of Western Australia, in which the trains currently operate, sensor data from a given trip is automatically transmitted in large batches to be received by remote servers once a train has concluded a round trip and arrives back in port from a loading point [88]. Given the current cellular network infrastructure in the region, this is the only feasible option for transmission of data, however Monash IRT have indicated that given future improvements in cellular network infrastructure, streaming the data back to remote servers in realtime is a likely possibility. This leads to the potential possibility of this research project's outcomes outlined in this thesis.

The form of batch handling and processing performed on the railway data currently leads to a number of limitations and problems. The data is currently stored in a relational database management system (RDBMS), and with the current implementation of the sensors, the data received is not consistently structured. In fact, the only sensor data guaranteed to be received in each batch is geographic location and time. Due to the highly structured nature of RDBMS technology, certain work-arounds need to be performed on the data so that it is

[2] https://platforms.monash.edu/irt/

compliant to RDBMS schemas, such as the insertions of default values in the case of missing attributes. Furthermore, low query performance has been noted as a problem plaguing the IRT team working with the railway data. They wish to resolve these problems by looking into non-relational models for their data storage and processing systems.

1.2 Research Aim

The main aim of the entire IRT research project is to develop a fully automated, non-relational big data pipeline to manage the data received from railway car sensors as a part of Monash University's Institute of Railway Technology project. The intention is to replace their current relational solution, with a non-relational big data system, offering at least the same capabilities at a larger scale and with higher performance. The scope of the overall project is relatively large, hence what is covered in this thesis refers to only a portion of the required work in the IRT.

The main aim of this thesis is to investigate possible solutions of dealing with the railway sensor data being streamed and processed in realtime, allowing such processing as the cleaning of noise from sensor readings. Based on identified candidate solutions, a recommendation will be formed which will be used by the IRT team to implement their required realtime processing logic.

To allow the possibility of realtime data streaming from the railway sensors, and forming a recommendation for the most suitable solution, appropriate data stream processing system (DSPS) technologies need to be looked at, tested, and evaluated. This makes up the core part of this sub-project's work. These DSPS technologies need to appropriately take, or accept, the data from some specified source, *e.g.* the sensors, apply any realtime processing logic that is required, *e.g.* pre-processing, then forward the data on for handling at another source, *e.g.* long term storage, such as HDFS.

A data filtering "pipeline", through which the sensor data flows, will need to be implemented in each of the candidate DSPS technologies, which can then be used for the testing and evaluation. By the end of this sub-project, we aim to have proof-of-concept implementations of the pipeline working on the National eResearch Collaboration Tools and Resources (NeCTAR) cloud services [10], making use of the candidate DSPS technologies, along with our overall DSPS technology recommendation for use in the IRT project.

1.3 Research Questions

The following research questions are the main focus points of this sub-project:

1. Which existing DSPS technologies can be used to build a data filtering pipeline for the Monash University IRT project?
2. What criteria-based qualitative and performance-based quantitative tests can be performed to **recommend** a particular DSPS technology for building the pipeline?
3. Can the data filtering pipeline be designed to be extensible, allowing for the addition of future realtime processing requirements?

Additionally, after answering each of these preliminary research questions, we will want to properly implement the theoretical discoveries from each stage. We do this with the goal of achieving a deployable pipeline that can be then be used in the testing and overall evaluation stages.

1.4 Thesis Structure

The proposed structure of the remainder of the thesis is as follows. Chapter 2, in §2, looks at prior research and work into the area of big data stream processing, performed both in industry and academia. A clear overview of the DSPS technologies chosen for this project, along with the design and implementation of the experimental systems to be used in the evaluation will be given in Chapter 3, in §3. An overview of the testing metrics, environment, results, and an overall evaluation discussion will be given in Chapter 4, in §4. Finally, Chapter 5, in §5 will conclude the thesis, highlighting the contribution made through this research project, along with highlighting any future work that have been made apparent as the result of this project.

1.5 Summary

This chapter has introduced the overall project, and in-turn the sub-project upon which this thesis will be based. It has briefly described the current state of the Monash University Institute of Railway Technology's project and the current problems that they are faced with, such as the need to deal with non-consistently structured data and low query performance. These problems are exacerbated given the non-realtime nature of the current data, where issues with sensors are only discovered much later after-the-fact. To overcome these issues, and to look forward into the hypothetical, but likely, possibility of having the technological infrastructure to support realtime data processing, this research aims to develop a realtime processing pipeline, taking the data straight from the railway sensors and perform any needed

realtime processing, such as data noise removal, and come up with a DSPS technology recommendation for the implementation of this pipeline.

This chapter also outlined a brief context of the research project and big data as a whole, the scope of this sub-project, and this sub-project's research questions. Finally it was concluded with an overview of the entire structure of this thesis. The following chapter will look into previous research that has been done on realtime big data processing and handling, along with going into more detail of the railway project's needs.

Chapter 2

Literature Review

To lay the ground work and context of this research project, this chapter goes into detail on existing research and work done in the areas of big data processing. Both more traditional batch processing of big data will be looked at, along with newer advances in the area of realtime stream processing. Different technologies enabling these processes will be discussed in detail, with the main focus being placed on the widely used open-source big data projects, such as those encompassed within the Hadoop ecosystem.

The realtime data processing of big data is now of more great importance to both academia and industry than it has ever been before. Advancements and progress in modern society can be directly attributed back to data and how we deal with it. The value of data has become increasingly more apparent, and data has become a sort of currency for the information economy [81]. Hence, those in society who realised the value of data early hold immense power over the entire economy, and in turn society, overall [66]. From seemingly inconsequential gains at the micro level, such as the ability to more accurately predict the rise and fall of airline tickets [41], to those of utmost importance for society as a whole, such as the prediction and tracking of the spread of the Swine Flu Pandemic in 2009 more accurately than the United States Centers for Disease Control and Prevention could using big data processing [69; 76]. It is applications of big data processing such as these that have been recognised by academics and organisations in industry alike, with the last decade seeing a major shift in research and development into new methods for the handling and processing of big data.

This review will be structured in two main sections. In §2.2, an overview and history for the existing major open-source batch processing systems for big data will be given. This will mostly surround the projects which fall under the umbrella of the Hadoop ecosystem. In §2.3, the current state of realtime data processing systems for big data will be discussed, including examples of the most widely used open-source realtime data stream processing

systems (DSPS). Comparisons will be drawn between the DSPS technologies and outlined in how they differ. Further discussion and analysis will be given in §2.4, including identifying the gaps in existing literature of which this research project aims to fill, before concluding in §5.

2.1 Big Data Processing Background

Much work has been done in the area of big data processing in the past decade, given the rise of large scale applications and services, a lot of the time involving large numbers of users and their given data. Data processing in general has been of large importance for many years, as no matter the business model, some sort of data has had to be dealt with. Traditionally, this has often been handled through the use of the relational model, as introduced by Codd [36]. This has been done through the use of relational database management systems (RDBMS), providing methods for the long term storage of data and functions for performing queries and manipulation on that data [25], such as through use of an implementation of structured query language (SQL) [33].

Given larger scales of data in newer projects, along with the needs to scale horizontally in terms of parallel and distributed processing, the more traditional relational forms of data management have become less attractive due to their complexity in terms of horizontal scaling [23]. Hence, companies dealing with these large scale use cases have started to look outwards to new directions of data management. This unattractiveness of existing technologies, in terms of scaling and parallelism, arguably led to the need to develop something new with these objectives in mind.

Note that in this chapter, we will refer to the processing of data in realtime as simply "realtime data processing". This term should be assumed to encompass the meanings that are also often represented through use of terms such as "data stream processing", "realtime stream processing", and "stream processing".

2.2 Batch Data Processing

Over the last decade, the main "go-to" solution for any sort of processing needed on datasets falling under the umbrella of big data has been the MapReduce programming model on top of some sort of scalable distributed storage system [27]. From a overly simplified functional standpoint, the MapReduce programming model essentially combines the common **map** and **reduce** functions (among others), often found in the standard libraries of many functional

programming languages, such as Haskell [65], Clojure [55], or even Java 8 [85], to apply a specified type of processing in a highly parallelised and distributed fashion [100].

The MapReduce data processing model specialises in batch mode processing on data which is already available. Batch data processing can be thought of where data needed to be processed is first queued up in "batches" Once ready, those batches get fed into the processing system and handled accordingly [38].

2.2.1 MapReduce and GFS

Dean and Ghemawat, in [43], originally presented MapReduce as a technology that had been developed internally at Google, Inc. to be an abstraction to simplify the various computations that engineers were trying to perform on their large datasets. The implementations of these computations, while not complicated functions themselves, were obfuscated by the fact of having to manually parallelise the computations, distribute the data, and handle faults all in an effective manner that met their performance requirements. The MapReduce model that was developed, enabled these computations to be approached from a more simple, high-level manner without the programmer needing to worry about optimising for available resources. In this way, it was more "declarative" while still allowing it to be written in a procedural language, such as Java. Furthermore, the MapReduce abstraction provided high horizontal scalability to differently sized clusters running on modest hardware.

While MapReduce conceptually offers many features that may be found in parallel database technologies, Dean and Ghemawat, in [44], emphasise the differences, and specifically flawed assumptions made about MapReduce in light of parallel databases. Issues in parallel databases such as fault tolerance when it comes to job failure, difficulties with input data from heterogeneous sources, and performing more complicated functions than those supported in SQL, are all fully supported and taken into consideration with MapReduce. Notably, while the programming model allows for further complicated functions, it also remains to be simple with only two major functions that need to be implemented: **map** and **reduce** [75].

As previously stated, the MapReduce programming model is generally used on top of some sort of distributed storage system. In the previous case at Google, Inc., in the original MapReduce implementation, it was implemented on top of their own proprietary distributed file system, known as Google File System (GFS). Ghemawat et al., in [48], define GFS to be a "scalable distributed file system for large distributed data-intensive applications", noting that can be run on "inexpensive commodity hardware". Note that GFS was designed and in-use at Google, Inc. years before they managed to develop their MapReduce abstraction, and the original paper on MapReduce from Dean and Ghemawat state that GFS was used to manage

data and store data from MapReduce [43]. Furthermore, McKusick and Quinlan, in [70], state that, as of 2009, the majority of Google's data relating to their many web-oriented applications rely on GFS.

2.2.2 Hadoop MapReduce and HDFS

While MapReduce paired with GFS proved to be very successful solution for big data processing at Google, Inc., and there was a considerable amount of notable research published on the technology, it was proprietary in-house software unique to Google, and availability elsewhere was often not an option [51]. Hence, the open-source software community responded in turn with their own implementation of Google's MapReduce and a distributed file system on which it runs, analogous to the role GFS played. The file system is known as the Hadoop Distributed File System (HDFS). Both of these projects, along with many others to date, make up the Apache Hadoop big data ecosystem. The Apache Hadoop ecosystem, being a top level Apache Software Foundation open-source project, has been developed by a number of joint contributors from various organisations and institutions such as Yahoo!, Inc., Intel, IBM, UC Berkeley, among others [53].

While Hadoop's MapReduce implementation very much was designed to be a complete functional replacement for Google's MapReduce, based upon published works by Google, HDFS is an entirely separate project in its own right. In the original paper from Yahoo! [78], Inc., Shvachko et al. present HDFS as "the file system component of Hadoop" with the intention of being similar to in external appearance to the UNIX file system, however they also state that "faithfulness to standards was sacrificed in favour of improved performance".

While HDFS was designed with replicating GFS' functionality in mind, several low-level architectural and design decisions were made that substantially differ to those documented in GFS. For example, in [30], Borthakur documents the method HDFS uses when it comes to file deletion. Borthakur talks about how when a file is deleted in HDFS, it essentially gets moved to a `/trash` directory, much like what happens in a lot of modern operating systems. This `/trash` directory is then purged after a configurable amount of time, the default of which being six hours. To contrast with this, GFS is documented to have a more primitive way of managing deleted files. Ghemawat, et al., in [48], document GFS' garbage collection implementation. Instead of having a centralised `/trash` storage, deleted files get renamed to a hidden name. The GFS master then, during a regularly scheduled scan, will delete any of these hidden files that have remained deleted for a configurable amount of time, the default being three days. This is by far not the only difference between the two file systems, this is simply an example of a less low-level technical difference.

HDFS, being designed as a standalone technology, is now used as a data storage component for many different big data technologies, not limited to only those within the Hadoop ecosystem [86] but also projects outside [97; 101]. HDFS was designed with general usage in mind and designed with horizontal scalability being a key consideration, hence runs on inexpensive clusters using modest hardware, while still delivering all its potential advantages [31].

2.2.3 Pig and Hive

Given the popularity of Hadoop, there were several early attempts at building further abstractions on top of the MapReduce model, which were met with a high level of success. As highlighted earlier, MapReduce was originally designed to be a nice abstraction on top of the underlying hardware, however according to Thusoo et al., in [89], MapReduce was still too low level resulting in programmers writing programs that are considered to be "hard to maintain and reuse". Thus, Thusoo et al., at Facebook Inc., built the Hive abstraction on top of MapReduce. Hive allows programmers to write queries in a similarly declarative language to SQL — known affectionately as *HiveQL* — which then get compiled down into MapReduce jobs to run on Hadoop [90].

Hive, along with HiveQL, provide a platform to run ad-hoc queries on top of HDFS stored data, which are run as batch MapReduce jobs. For example, all generated reports on Facebook to their paid advertisers are generated using batch jobs constructed using Hive, along with several other analytics use cases at Facebook [91]. An added benefit shown at Facebook from the use of Hive, over traditional MapReduce, was the users familiarity with SQL-like declarativeness in regards to querying and manipulating data [32].

Another commonly used abstraction, that was developed prior to Hive, was what is known simply as Pig. Like Hive, Pig attempts to be a further higher level abstraction on top of the MapReduce model, which ultimately compiles down into MapReduce jobs. what differentiates it more from Hive is that instead of offering a solely declarative SQL-like language for data manipulation and querying, it offers a language that takes more of a procedural and functional dataflow paradigm influenced approach [73]. This still allows for relational algebraic constraints to be specified on the data set in relation to defining the result [74]. Olston et al. describe Pig's language — known as *Pig Latin* — to be what they define as a "dataflow language", rather than a strictly procedural or declarative language.

Pig works by having the program first parsed into a directed acyclic graph (DAG), which then can be be optimised using DAG-related graph theory operations, before having that DAG being compiled down into native MapReduce batch jobs. The MapReduce stage is

also optimised through use of function ordering, before being submitted to run on a Hadoop cluster [82].

Hive, also being a high level abstraction on top of MapReduce, also enable many of their own optimisations to be applied to the underlying MapReduce jobs during the compilation stage [47; 90] as well as having the benefit of being susceptible to manual query optimisations, familiar to programmers familiar with query optimisations from SQL [52].

2.3 Realtime Data Processing

With HDFS being an open-source project, with a large range of users [96] and code contributors [53], it has grown as a project in the last few years for uses beyond what it was originally intended for: a backend storage system for Hadoop MapReduce. HDFS is now not only used with Hadoop's MapReduce, but also with a variety of other technologies, a lot of which run as a part of the Hadoop ecosystem, but also those which do not. Big data processing has moved on from the more "traditional" method of processing, involving MapReduce jobs, which were most suitable for the batch processing of data, to those methods which specialise in the realtime processing of streamed data. The main difference of which is that rather than waiting for all the data before processing can be started, in realtime data processing, the data can be streamed into the processing system and processed at any time with whatever data is made available.

The use cases for such realtime data processing differ much from batched processing in ways a lot of the time relating to the availability of data. With batch jobs, generally processing is done after-the-fact, or after data has been collected from various sources and stored in a system, similar to HDFS. With realtime data processing, data can be processed as soon as it is received from sources, without using a storage system, such as HDFS, to first hold it. For example, an array of sensors can stream their data directly to the data stream processing system (DSPS), without any intermediate step. From there, the DSPS will process the data according to how it is programmed, and either send it on to further systems, or store it for later batch processing.

2.3.1 Hadoop YARN

As previously looked at, the aim of the MapReduce model was performing distributed and highly parallelised computations on distributed batches of data. This suited a lot of the big data audience, and hence Hadoop became the dominant method of big data processing for many years [67]. However for some more specialised applications, such as the realtime

monitoring of sensors, as touched on earlier, stock trading, and realtime web traffic analytics, the high latency between the data arriving and actual results being generated from the computations was not satisfactory [61]. Furthermore, such use-cases were simply not suited for batch processing.

A recent (2013) industry survey on European company use of big data technology by Bange, Grosser, and Janoschek, noted in [26], shows that over 70% of responders show a need for realtime processing. In that time, there has certainly been a response from the open-source software community, responding with extensions to more traditional batch systems, such as Hadoop MapReduce, along with complete standalone DSPS solutions.

On the Hadoop front, the limitations of the MapReduce model were recognised, and a large effort was made in developing the "next generation" of Hadoop so that it could be properly extensible and used with other programming models not locked into the rigidity of the MapReduce model. This change in Hadoop became known officially known as YARN (Yet Another Resource Negotiator). According to the original developers of YARN, Vavilapalli et al. state that YARN enables Hadoop to become more modular, decoupling the resource management functionality of Hadoop from the programming model (traditionally, MapReduce) [94]. For Hadoop end-users, this decoupling essentially allowed for non-MapReduce technologies to be built on top of the Hadoop ecosystem, which was traditionally a part of MapReduce, allowing for much more flexible applications of big data processing on top of the existing robust framework Hadoop provides.

Examples of such systems now built, or in some cases ported, to run on top of Hadoop, providing alternative processing applications and use cases, include:

- Dryad, a general-purpose distributed execution system from Microsoft Research [58]. Dryad is aimed at being high level enough to make it easy for developers to write highly distributed and parallel applications.

- Spark, a data processing system, from researchers at UC Berkeley, that focuses on computations that reuse the same working data set in-memory over multiple parallel operations [103]. Spark, and in particular Spark Streaming, will be looked at further in §2.3.3.

- Storm, a realtime stream processing system [71, p. 244]. Storm allows specified processing on an incoming stream of data indefinitely, until stopped. Storm will be looked at further in §2.3.2.

- Tez, an extensible framework which allows for the building of batch and interactive Hadoop applications [87].

- REEF, a YARN-based runtime environment framework [35]. REEF is essentially a further abstraction on top of YARN, with the intention of making a unified big data application server.
- Samza, a relatively new realtime data processing framework from LinkedIn. Discussed further in §2.3.4.

These are just some of the more popular examples of applications built to interact with the Hadoop ecosystem via YARN.

With the emergence of YARN, the ability to build DSPS technologies on top of Hadoop led to a lot of interest in realtime processing solutions that was often looked over by previous Hadoop batch users.

2.3.2 Storm

One very notable DSPS technology developed independently of Hadoop, and that is gaining immense popularity and growth in its user base, is the Storm project. Storm was originally developed by a team of engineers lead by Nathan Marz at BackType [84]. BackType has since been acquired by Twitter, Inc. where development has continued. Toshniwal et al. [92] describe Storm, in the context of its use at Twitter, as "a realtime distributed stream data processing engine" that "powers the real-time stream data management tasks that are crucial to provide Twitter services" [92, p. 147]. Since the project's inception, Storm has seen mass adoption in industry, including amongst some of the biggest names, such as Twitter, Yahoo!, Alibaba, and Baidu [9].

While Storm does not run on top of YARN, there is currently a large effort from engineers at Yahoo!, Inc. being put into a YARN port for Storm, aptly named *storm-yarn* [49; 64]. This YARN port will allow applications written for Storm to take advantage of the resources managed in a Hadoop cluster by YARN. While still in early stages of development, storm-yarn has begun to gain attention in the developer community, through focus from channels such as the Yahoo Developer Network [95] and Hortonworks [45].

The programming model for Storm consists of a Storm topology. This topology is made up of various components, which either "transform" the data in some specified way (known as a *bolt*), or input the data from a specified source before outputting it to one of the given bolts (known as a *spout*). The layout of spouts and bolts are then constructed in a separate topology definition, defining the way the data streams (represented as tuples) flow through the topology [59]. An example Storm topology is shown in Figure 2.1.

Storm, while predominantly written in Clojure, running on the JVM, allows topologies to be defined and written in any available programming language through the use of a Thrift

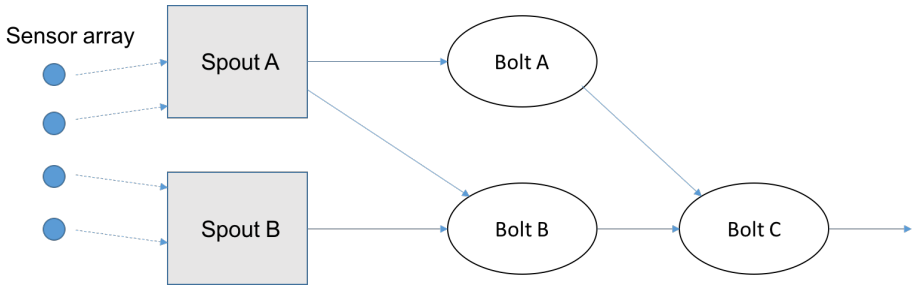


Fig. 2.1 An example Storm topology showing the source of the tuple stream from spouts A and B, received from an array of sensors, which then forwards the stream on for processing at bolts A and B. Bolts can forward streams onto other bolts.

definition for topologies [4]. Thrift is a software library, developed at Facebook, Inc., for the purpose of defining datatypes and interfaces for use in multiple programming languages [79]. Storm provide example Storm projects written in Java, Clojure, and Python, however most official Storm documentation, at present, use Java.

2.3.3 Spark StreaminS

Spark is another popular big data distributed processing framework, offering of both realtime data processing and more traditional batch mode processing, running on top of YARN [103]. Spark was developed at UC Berkeley, and is notable for its novel approach to in-memory computation, through Spark's main data abstraction which is termed a *resilient distributed dataset* (RDD). An RDD is a set of data on which computations will be performed, which can be specified to be cached in volatile memory across multiple machines. What this then allows is multiple distributed operations being performed on this same dataset in parallel without such overhead, as secondary storage IO operations, which is noted by Spark as a performance limitation of MapReduce [42]. Spark is designed with highly iterative computations in-mind.

The Spark project has experienced a large growth in terms of project contributions over the past few years, now encompassing a number of sub-projects running on top of the Spark engine, all suited for different use cases. These include:

- **Spark SQL**: An SQL-like engine that runs on top of Spark, allowing Spark users to reap the benefits of using a declarative SQL-like language on top of Spark's functional API [24]. (Previously known as Shark [98]).

- **Spark GraphX**: A graph abstraction on top of the Spark engine, allowing for the efficient expression of distributed graph structures and computations on those structures [97].

- **Spark MLlib**: A distributed scalable machine learning platform on top of Spark, allowing for efficient machine learning capabilities [80].

- **Spark Streaming**: A DSPS platform built on top of the Spark engine, allowing for the processing of data in realtime [104].

Given these different platforms built on top of Spark's in-memory computation engine, Spark offers solutions for many different big data processing use cases. We are most interested in Spark Streaming, given its realtime DSPS offerings. Spark Streaming uses a different programming model than that which is offered in the base Spark platform, that involves what is labelled as *DStreams* (discretised streams). DStreams essentially allows for a series of deterministic batch computations be treated as single a realtime data stream [104]. The DStream model is specific to the Spark Streaming system — the original batch mode Spark system continues to use the previously mentioned RDD abstraction — and the creators claim performance improvements of being $2-5\times$ faster than other DSPS technologies, such as S4 and Storm, given its use of in-memory stream computation [105]. However, this is a topic of debate, with opponents claiming that these Spark performance claims are very much dependent on external factors, such as available system resources [50].

Both Spark and Spark Streaming have started to gain notable usage in both industry and research projects in academia in the last few years. Online video distribution company, Conviva Inc., report to be using Spark for the processing of analytics reports, such as viewer geographical distribution reports [60; 102]. The Mobile Millennium project at UC Berkeley [93], a traffic monitoring system that uses GPS through users' cellular phones for traffic monitoring in the San Francisco Bay Area, has been using Spark for scaling the main algorithm in use for the project: an expectation maximisation (EM) algorithm that has been parallelised by being run on Spark [57].

Spark Streaming, while originally written in Scala, supports official APIs for Java, Scala, and Python. Note that the Python API for Spark has only recently been introduced and, as of yet (version 1.3.1), does have have support for all the data sources which Java and Scala offer [21].

2.3.4 Samza

Samza is a relatively new realtime big data processing framework originally developed in-house at LinkedIn, which has since been open-sourced at the Apache Software Foundation [77]. Samza offers much similar functionality to that of Storm, however instead the running of Samza is highly coupled with the Kafka message broker, which handles the input

and output of data streams. Essentially, Kafka is a highly distributed messaging system that focusses on the handling of log data [62], integrating with the Hadoop ecosystem.

While Samza is lacking in maturity and adoption rates, as compared to projects such as Storm, it is built on mature components, such as YARN and Kafka, and thus a lot of crucial features are offloaded onto these platforms. For example, the archiving of data, stream persistence, and imperfection handling is offloaded to Kafka [28]. Likewise, YARN is used for ensuring fault tolerance through the handling of restarting machines that have failed in a cluster, among further resource management [28].

Samza processes streams of data through pre-defined jobs which perform any specified operation on the data streams, much like bolts in Storm. The source of streams in Samza comes from Kafka, which can be also used as an output destination for jobs. How the data is received from the actual source (such as a sensor), then sent through Kafka is a matter of further implementation on the part of the programmer [99]. Note that a layout of jobs in Samza differs considerably from defining a Storm topology layout, in that each individual job defines its input and output sources, rather than jobs being linked together in a separate definition file.

Samza is mainly programmed using Scala, running on the JVM, and allows programming of jobs through a set of Java interfaces [18]. It is officially supported and recommended to program Samza jobs in Java, however it would also be possible to implement the provided interfaces using any JVM language, such as Scala or Clojure.

2.3.5 S4

S4 (Simple Scalable Streaming System) is another realtime big data processing framework that originated at Yahoo!, Inc. that has since been open-sourced [72]. It is a relatively old project compared to the before-mentioned projects, with development becoming less of a priority in the last few years, with the last update being released in 2013. S4 was highly influenced in design by the MapReduce programming model.

Much like what was noted about Samza in §2.3.4, S4 attempts offload several lower level tasks to more mature and established systems specialising in those areas. The logical architecture of S4 lays out its jobs in a network of processing elements (PEs) which are arranged as a directed acyclic graph. Each of these PEs entail the type of processing to be done on the data at that point in the network. Each of the PEs are assigned to a processing node, a logical host in the cluster. The management and coordination of these processing nodes is offloaded by S4 to ZooKeeper [61]. Much like the before-mentioned Kafka, ZooKeeper in itself is its own complex service used as a part of many different big data infrastructures,

including Samza. ZooKeeper specialises in the high-performance coordination of distributed processes inside distributed applications [56].

Note that a number of performance issues have been noted in S4, including non-linear performance scalability, bad resource management, low expectations on fault tolerance, and high reliance on the network [34]. As mentioned earlier, development on S4 has stagnated. Adoption of S4 over DSPS platforms, such as Samza, Storm, and Spark Streaming, has not been seen to be major, and literature in big data research have shown a significant lack of focus on S4 when looking at the topics of DSPS technologies.

2.4 Discussion and Analysis

From the previously covered literature, it is rather difficult to provide a reasonable comparison for all the different realtime data processing projects. A lot of the claims made in original literature relating to the projects cannot be quantified impartially, as comparisons or tests have not been carried out relating to other projects. Instead, Stonebraker, Çẽntintemel, and Zdonik proposed what they claim to be the eight requirements for realtime data processing systems [83], which can be used to given an impartial comparison of the previously covered projects. The requirements were defined a number of years prior to the creation of the four main realtime data processing systems that were covered (2005), however are highly cited as being the defining features that the current generation of realtime data processing systems have strived to meet. The requirements, put forward by Stonebraker et al., are summarised as follows:

1: Keep the data moving - This requirement relates to the high mobility of data and importance of low latency in the overall processing. Hence, processing should happen as data moves, rather than storing it first then processing what is stored.

2: SQL on streams - This requirement states that a high-level SQL-like query language should be available for performing on-the-fly queries on data streams. SQL is given as an example, however it is noted the language's operators should be more oriented to data streams.

3: Handle stream imperfections - Given the high degree of imperfections in data streams, including factors such as missing and out-of-order data, this requirement states that processing systems need to be able to handle these issues. Simply waiting for all data to arrive if some is missing is not acceptable.

4: Generate predictable outcomes - This requirement relates to the determinism associated with the outcomes of specified processes to be applied to data. A realtime processing system should have predictable and repeatable outcomes. Note that this requirement is rather hard

to satisfy as in practice data streams are, by character, rather unpredictable. However, the operations performed on given data, and thus the outputs, are required to be predictable.

5: Integrate stored and streamed data - This requirement states that a realtime processing system should provide the capabilities to be able to process both data that is already stored and data that is being delivered in realtime. This should happen seamlessly and provide the same programming interface for either source of data.

6: Guarantee data safety and availability - This requirement states that realtime processing systems should ensure that they have a high level of availability for processing data, and in any cases of failures, the integrity of data should remain consistent.

7: Partition and scale applications automatically - This requirement states that the partitioning of and processing of data should be performed transparently and automatically over the hardware on which it is running on. It should also scale to different levels of hardware without user intervention.

8: Process and respond instantaneously - This requirement relates to delivering highly responsive feedback to end-users, even for high-volume applications.

The previously covered literature has been used to determine whether or not the beforementioned realtime data processing systems adhere to these requirements. The outcome of this is shown in Table 2.1.

Note that in Table 2.1, those cells with "N/A" as the value simply mean that the literature is inconclusive on whether or not they adhere to the particular requirement.

Table 2.1 Realtime data processing systems compared

	Storm	**Spark Streaming**	**S4**	**Samza**
1: Data mobility	Fetch	Micro-batch	Push	Fetch (Kafka)
2: SQL on streams	Extension (Trident)	Yes (SparkSQL)	No	No
3: Handling stream imperfections	User responsibility	N/A	No	Yes (Kafka)
4: Deterministic outcomes	N/A	N/A	Depends on operation	N/A
5: Stored and streaming data	Yes (Trident)	Yes (Spark)	No	No
6: High availability	Yes (rollback recovery)	Yes (checkpoint recovery)	Yes (checkpoint recovery)	Yes (rollback recovery)
7: Partition and scaling	User responsibility	N/A	Yes (KVP)	Yes (Kafka topics)
8: Instant response	N/A	N/A	N/A	N/A

From this table, we can note that literature covering both deterministic outcomes and instant response from each of the DSPS technologies are not consistent. This may not be a problem for the project at hand, however proves to be a factor for further research. Furthermore, the possibility of performing batch computations on stored data are not available on

both S4 and Samza. Given the dominance of batch processing shown to be needed in many areas of big data processing, this would be an important factor of adoption of DSPS technologies in many existing applications. Existing applications may want to introduce realtime data processing on-top of their existing batch processing logic. With technologies affording both batch and realtime data processing possibilities, both Storm and Spark Streaming would be very appealing options.

2.5 Conclusion

The choosing of appropriate realtime data processing platforms for the processing of a given application and dataset is an important, and often confusing, problem. Most platforms compare themselves in terms of performance with other frameworks, often which are disputed by members in other platform "camps" [46; 50]. Hence, providing an informed recommendation based on the features each platform offers, and an impartial performance evaluation is a much needed and important contribution to address this gap in qualitative and quantitative evaluations of the available platforms. These evaluations will be looked at further in Chapter 4.

From looking at existing literature in the area of realtime data stream processing, we have identified four candidate data stream processing technologies for use in the evaluation:

- Storm
- Samza
- Spark Streaming
- S4

However, due to the noted issues in S4 relating to lack of development, lack of adoption, and performance issues, in relevant literature, we have made the decision to ignore S4 for use in the overall evaluation. Hence, evaluation will be constrained to the candidate DSPS technologies of Samza, Spark Streaming, and Storm.

This research will further the field of realtime big data processing in addressing these shown gaps, and making the decision process far more streamlined for researchers and developers of DSPS technologies alike.

Chapter 3

Pipeline Design and Implementation

This chapter describes the design and implementation of the prototype data filtering pipelines to be built for testing and evaluating our candidate data stream processing system (DSPS) technologies. This will be performed with the intention of arriving at a recommendation system for the case study of the Monash University Institute of Railway Technology (IRT) project. Each of the candidate DSPS technologies have been detailed in-depth in previous chapters, and all exhibit the capabilities needed to implement the intended prototype pipelines.

This chapter will first go into the design of the data filtering pipeline, in §3.1, before detailing the testing environment which was used to implement and test each pipeline, in §3.2. In §3.3, the implementation of the prototype pipelines will be detailed for each of the candidate DSPS technologies, as well as looking at the programming possibilities afforded by each technology. Finally, the chapter will be concluded in §3.4.

3.1 Data Filtering Pipeline Design

The overall design of the data filtering pipeline will be designed in such a way that readings from sensors can be fed into the pipeline, processed in some specified manner, then output for further use, including storage for batch processing, or discarded in the case of noisy data. Thinking of the pipeline in such a way, we can decompose the design into three main components:

1. Sensor data input component
2. Sensor data processing component
3. Results output component

Each component would also need to connect somehow to form the entire pipeline design. The components are listed in order of their precedence of their role in the pipeline. Hence, the first component's output would be the input to the second component, of which the output would be the input to the third component. This creates the dataflow behaviour exhibited in pipelines. This basic design that we have so far is illustrated in Figure 3.1.

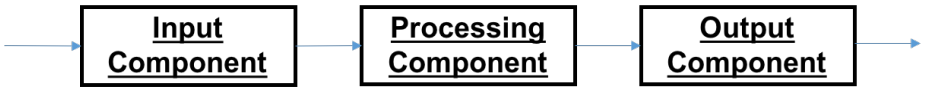


Fig. 3.1 A simplistic overview of our pipeline design. Note the dataflow behaviour exhibited with each component's output being an input of the following component.

We now have a simple design of the pipeline, however we are left with the questions of the output of the **Output Component**, the input of the **Input Component**, and the contents of each component.

The output of the **Output Component** will simply be whatever extensions are needed to be made to the pipeline. Hence, rather than acting as a sink, this component will be an abstract interface which any extension needed will be able to adhere to extend the pipeline. Designing the pipeline in this manner directly addresses our third research question touched upon in Chapter 1, in §1.3. This allows the output data to be dealt with in any manner required by the IRT team, whether it be stored for later batch processing, or forwarded on for further realtime processing.

The input of the **Input Component** will be the data read from the sensors to which the pipeline is connected to. The sensors generally would not be directly connected to the pipeline, so we could have another abstract interface here that any sensor connects to, then the interface sends the data to the actual component. This allows data to be fed into the pipeline from any type of sensor that is able to adhere to the interface.

The contents of each component is a larger task. The **Processing Component** will be made up of an arbitrary number of tasks, each of which are tasked to process the data in some manner before forwarding it onto whatever the next specified task is. This, in itself, acts as a mini-pipeline where the core of processing will be done. Here we will say that the **Processing Component** consists of n sequential processing tasks. For the **Input Component**, the contents will consist of a task-like component tasked with passing data onto the processing tasks which make up the **Processing Component**. The contents of the **Output Component** will be as described previously: an abstract interface allowing for the extension of the pipeline.

This leaves us with a final pipeline design as illustrated in Figure 3.2.

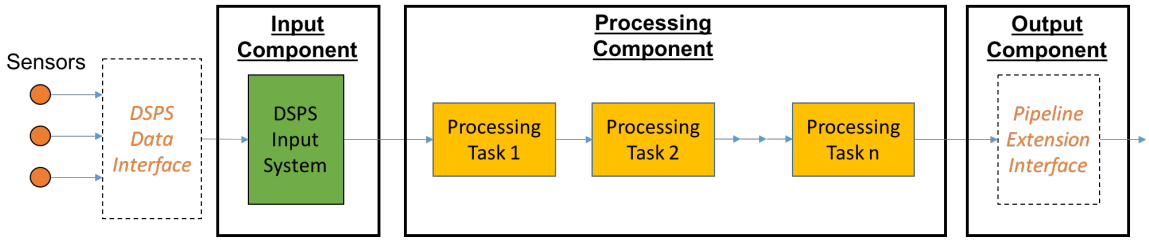


Fig. 3.2 A complete overview of our pipeline design.

Note that this diagram illustrates an overall abstract pipeline design. However, for implementation of the pipeline which will be used in the evaluation parts of the project, we need to implement concrete components for testing. In this project, we do not have access to any sensors, or any such hardware, for sending readings to the pipeline for processing. Hence, for evaluation we have had to simulate the sensor component of the overall pipeline design in some way. Note that this does not affect the pipeline in any way, since the actual sensor hardware part of the design is abstracted away from the software pipeline itself. The function of the sensors is to send data to the pipeline at any arbitrarily specified interval through use of the DSPS Data Interface. To simulate this functionality multiple methods exist. The most basic of such is arguably reading data from a file and placing that onto a stream for the pipeline to process. However, this very much mimics batch processing methods, and hence is not ideal for simulating realtime data. Furthermore, placing data from a file onto the pipeline does not accurately reflect the client-server networking functionality that will exist in a real sensor-pipeline connection.

A better way to simulate this sensor-pipeline realtime data connection is to develop a data entry interface to the pipeline upon which data can be sent to at any time. This data entry interface can then be used to simulate arbitrarily specified intervals upon which data can be sent, thus mimicking the functionality of a sensor. To do this while still adhering to the client-server connection design, we can use a socket connection, allowing data to be sent over a specified port to be placed upon the pipeline. Each candidate DSPS technology either has native support for server socket connections, given their commonplace as mediums upon which data can be sent over, or support from the programming language standard libraries used to implement realtime processing logic.

Hence, the **Input Component** will be implemented as a server socket connection, and the **Processing Component** as a filtering task. Speed sensor readings will be passed in along the socket connection using a tool allowing interface to network ports, such as `netcat`[1], and filtered to make sure they are valid speed values, between the limits of 0 and 90. Any speeds

[1] http://netcat.sourceforge.net/

outside of these limits can safely be deemed to be either bad, or noisy, sensor readings. This results in a concrete design as shown in Figure 3.3.

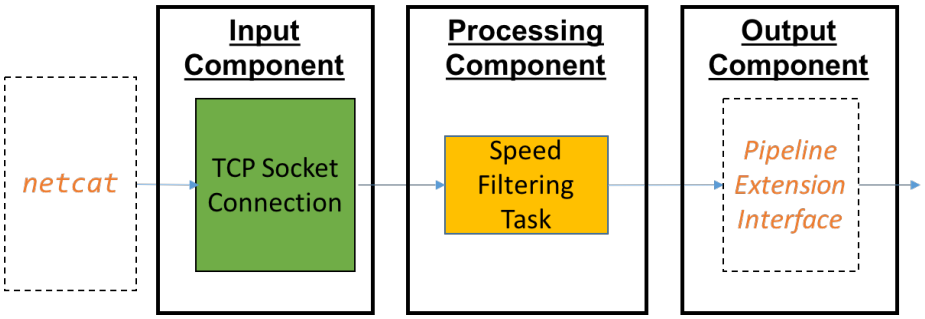


Fig. 3.3 A concrete implementation of our pipeline design.

3.2 Testing Environment Details

3.2.1 Details on Host System

The testing environment on which the prototype pipelines were implemented using each of the candidate DSPS technologies is the same National eResearch Collaboration Tools and Resources (NeCTAR) cloud computing environment touched upon briefly in Chapter 1. This cloud service platform is provided as an initiative by the Australian Government for researchers that require the infrastructure[2].

Our environment consists of a single NeCTAR cloud instance of which the details are specified in Table 3.1.

Table 3.1 Control system used for testing pipelines.

Distribution	Ubuntu GNU/Linux 14.04.2 LTS
Kernel	Linux 3.13.0-36-generic
Architecture	x86_64
Virtual CPUs	16
Available RAM	64 GB
Available Disk Space	490 GB

The NeCTAR cloud instance utilises a single node upon which all processing is performed. In a real deployment, it is common to use a full cluster upon which the pipeline would be deployed, where processing is shared and distributed between nodes. Due to limited testing resources, all testing has been performed on a single node cluster, however this remains a

[2]https://www.nectar.org.au/about-nectar

constant feature between all implementation and tests performed in the project. Furthermore, the pipelines built upon each of the candidate DSPS technologies are freely configurable to be deployed to run on arbitrarily sized clusters without requiring changes to the underlying source code.

3.2.2 Details on Data Stream Processing System Technologies

The DSPS technologies that have been chosen to be focused on in this sub-project include the following:

- Samza
- Storm
- Spark Streaming

Literature concerning these DSPS technologies have been covered in §2.3, however we will look into more depth into the systems regarding their usage from the programmer's point-of-view. Note that in the previous chapter, a further DSPS technology, S4, was covered, however due to its noted decline in usage and development in the chapter, it has been decided to omit the use of the technology from this project.

Version 0.9.0 of Samza was used for implementation of the prototype pipeline in Samza. At the time of implementation, this was the major stable version of Samza, accepted as a top-level Apache project. Java, the only official programming language supported by the Samza project, was used, using JDK 1.8.0 as the target implementation.

Version 0.9.4 of Storm was used for implementation of the prototype pipeline in Storm. As with Samza, this was also the major stable version of Storm available. Java 8 was used as the implementation language, also compiling for the JDK 1.8.0.

Version 1.3.0 of Spark and the Spark Streaming extension were used for the Spark Streaming implementations. Prototype pipelines were implemented in both Scala 2.9.2, targeting the JDK 1.8.0, and Python 2.7, which is made possible through the relatively new (as of Spark 1.2.0) PySpark subsystem. Implementations for the Spark Streaming pipeline were built targeting both Spark JVM and PySpark due to numerous undocumented claims that performance differs on both Spark implementations. These claims are addressed further in the following chapter.

The specific JDK implementation for all tests was the Intel 64-bit OpenJDK 1.8.0 release 45 built and packaged for the GNU/Linux distribution of Ubuntu 14.04.2 LTS.

3.3 Implementation of Pipelines in Data Stream Processing System Technologies

Here, we will look at how the each of the candidate DSPS technologies were used for implementation in relation to our testing pipeline design. We will look at the possibilities each technology affords in terms of programmability, and then what was required to be implemented for our testing and evaluation to arrive at a DSPS recommendation.

For each DSPS technology, we will look at what was needed to be implemented for the following components from our design:

- Input component
- Processing component
- Output component

3.3.1 Samza

Samza defines itself as a realtime data processing system, which allows the processing of streams made up of immutable messages of similar type or category. Streams in Samza exist independently of other concepts, allowing themselves to be *consumed* from or *produced* to by various supported systems. The concepts of consuming and producing refer to the actions of getting data from a stream and placing data onto a stream, respectively. Samza defines the systems that can be used for consuming and producing to streams to be any piece of software that implements the stream abstraction. For example, Kafka, a popular distributed messaging queueing system, is often used to consume and produce to Samza streams. Systems such as these can be "plugged" into a Samza project to work with the same streams which native Samza applications are written to work with [19].

For a Samza programmer, the most important concepts of the overall Samza project to understand are the concepts of the **system** and **task**. In a Samza project, a system exists to produce a particular stream from some source of data which then can be processed by a task which consumes it. A task will always perform some processing before producing the results as output to another stream. Systems and tasks can then be arranged in a pipeline-like structure allowing a specified type of processing on the messages which travel on those streams. Hence, it is easy to think of a system as the source of data for an overall Samza project. A very simple example of such a Samza project is shown in Figure 3.4.

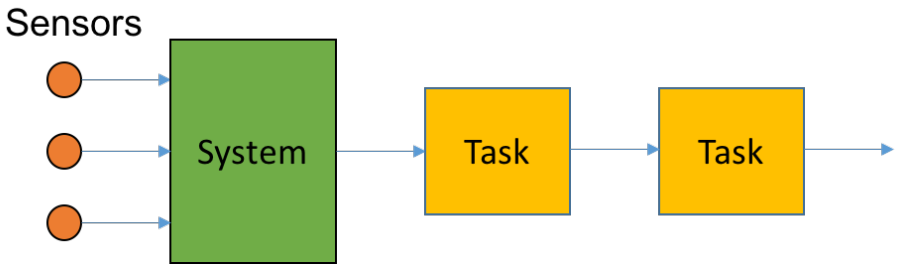


Fig. 3.4 A simple Samza pipeline showing the roles of the system (handling input data), task (handling data processing), and stream concepts. Streams are represented as arrows.

3.3.1.1 Programming Overview

When programming a Samza project, interfaces exist for both the system and task concepts through the `org.apache.samza.system.SystemConsumer` and `org.apache.samza.task.StreamTask` Java interfaces [19]. Default systems exist in the Samza project, and may be used to produce streams which can then be processed by an implemented `StreamTask`.

The `StreamTask` interface is arguably the main interface in which the most important processing will be done in a Samza project. It requires implementing a single method, `process`, which passes in an immutable message from the stream on which the task is configured to consume. These messages are then processed as specified by the programmer, before the result of the processing is wrapped in a message envelope and produced to a stream, also specified by the programmer [19].

The streams on which `StreamTask` implementations consume are specified in a task specific configuration file, independent to that of the implementation source code. These configuration files offer a range of parameters for the execution of tasks, each of which are documented[3], however can result in quite verbose and hard to understand configuration files. For each task configuration, a system needs to be defined that produces to the stream that the task consumes from. These can either be a self-implemented system, or an existing pluggable system, such as Kafka [19].

For self-implementation of systems, the `SystemConsumer` interface requires much more understanding and effort to implement than that of the `StreamTask` interface. Methods, such as `start`, `stop`, `register`, and `poll` need to be implemented, allowing for the producing of messages onto a particular stream. The `start` and `stop` methods simply connect and disconnect the system to the underlying system, while the `poll` method takes care of producing any messages from the system. The `register` method is much more complicated in the way

[3] https://samza.apache.org/learn/documentation/0.9/jobs/configuration-table.html

that it requires registering the implemented system with lower-level components of Samza to integrate the system into Samza [19].

Note that, due to Samza's relative infancy, compared with other DSPS technologies, many of these interfaces either completely lack or offer insufficient documentation. Furthermore, for the same reason, example Samza projects are hard to find. This is a key factor that will be touched upon in the next chapter on evaluating the DSPS technologies.

3.3.1.2 Input Component

For implementation of the input component design in Samza, this revolves around the `SystemConsumer` interface. As we want the Samza system to produce to a stream based on values received over a server socket connection, we can use an instantiation of the `java.net.ServerSocket` class to do this. This can be set up in the system's constructor and then, in the system's `start` method, attempt to poll any incoming values using a `java.io.InputStream` generated from that connection. If any values are encountered on that connection, we can produce them to the default speed value stream, which will consumed by the filtering task.

The system's `stop` method simply will be used to close the server connection, while the `register` method will be used to register the system with the previously mentioned default speed value stream.

The source code of this Samza system can be seen in Appendices A.1 and A.2.

3.3.1.3 Processing Component

For the implementation of the processing component design in Samza, this revolves around the `StreamTask` interface. This Samza task will get its input by consuming from the same stream on which the previously detailed Samza system implementation produced to. The main `process` method can then be used to consume a value from the stream, then check if the value received is between the acceptable speed limits. Depending on the outcome of this test, the message is either flagged for being noisy, or not, and then produced to a stream for valid values, or a stream for noisy data. This allows the filtering of data, with flagged data being produced to a separate stream to those data that are deemed to be valid.

The source code of this Samza task is available in Appendices A.3 and A.4, while the task configuration is available in Appendix A.5.

3.3.1.4 Output Component

For the output component of our designed pipeline, no implementation needs to be done. Samza, by design, affords extension by implementing a new `StreamTask` that is configured to consume from an existing stream in the Samza project.

3.3.1.5 Deployment

Samza projects are generally compiled and distributed as JAR files, which may or may not contain project dependencies, which then can be run directly on installed Samza distributions. Assembly of these JAR files are conventionally performed through use of Java build automation tools, such as Maven or Gradle.

For the implementation of our designed pipeline, Maven was used to assemble a JAR file containing compiled bytecode, which can be deployed on different Samza installations.

3.3.2 Storm

Storm defines itself as a distributed realtime computation system that provides primitives for performing realtime computations. Storm compares itself to existing big data systems, such as Hadoop MapReduce, which similarly provides primitives for performing batch data computations.

Like Samza, Storm also is built around the key concept of a stream, however Storm's definition of a stream slightly differs to that of Samza's. In Storm, a stream is defined as an unbounded sequence of immutable tuples, rather than a sequence of immutable messages as they are defined in Samza. A tuple is defined as a dynamically typed list of values, in which the values can be of any type. Rather than consuming and producing to streams, Storm approaches computations on streams through *stream transformations* using the previously mentioned realtime computation primitives.

3.3.2.1 Programming Overview

The realtime computation primitives that Storm offers are the components of Storm that are of most interest to the programmer. These primitives are known as the **spout** and **bolt**, previously looked at in §2.3.2. To compare these components to those present in Samza, the spout acts as a source of streams, much like Samza's systems, while the bolts act as consumers of streams, which process and emit new streams, much like Samza's tasks. Storm offers interfaces for implementation of custom spouts and bolts through the `backtype.storm.spout.ISpout` and `backtype.storm.task.IBolt` interfaces, respectively [2].

As explained in §2.3.2, spouts and bolts are arranged in a particular way to make up an overall Storm topology, which is used to transform streams in some specified manner. A simple example of such a topology is shown in the previous chapter, in Figure 2.1.

Basic spouts require implementation of multiple methods, `open`, `nextTuple`, and `declareOutputFields`, which act as a spout initialiser, handler for each tuple to be emitted to a stream, and a tuple type declaration, respectively. Implementation of a simple spout is a much easier task in Storm than a system in Samza, due to different provided interfaces depending on the type of configuration needed for a particular spout. For example, a very basic spout can be made, using a lot of default configuration options, by implementing the `backtype.storm.topology.base.BaseRichSpout` interface [2].

Bolts also require implementation of three main methods, `prepare`, `execute`, and `declareOutputFields`, allowing bolts to be initialised, process received tuples in a specified manner, and declare tuple value types, respectively [2].

One major point of difference with these components, relative to their analogous components in Samza, is that the spout and bolt implements are completely independent to the overall Storm topology layout. In Samza, systems and tasks require the output streams to be hardcoded within their respective implementations, however Storm has a further separate source file for defining the topology layout. This topology layout is generally defined in a separate file to all spout and bolt implementations using the provided `backtype.storm.topology.ToplogyBuilder` class. This class exposes methods for defining the topology layout, in terms of spout and bolt placement, along with inputs and outputs for each spout and bolt implementation, as well as overall topology configuration [2]. An obvious downside to this over Samza configuration files is that it requires a recompilation in the case that configuration options need to be tweaked, while Samza configuration files are not part of the compiled source code.

The source code for the Storm topology built and configured for our prototype pipeline can be viewed in Appendix B.3.

3.3.2.2 Input Component

For the implementation of the input component design in Storm, this revolves around the `BaseRichSpout` interface. Much like the Samza implementation, this implementation also uses `java.net.ServerSocket` and `java.io.InputStream` for listening on a server socket connection for incoming speed values. The `open` method is used for the creation of such a server socket connection, while the `nextTuple` method is used for attempted to read any values off the connection. If a value is encountered, it is constructed into a Storm tuple containing the speed value, and emitted onto the configured stream. The `declareOutputFields`

method is then used to declare the speed value type as a single value of type Double to be expected to make up a valid tuple.

The source code for this Storm spout is available in Appendix B.2.

3.3.2.3 Processing Component

For the implementation of the processing component design in Storm, this revolves around the `BaseRichBolt` interface. The `execute` method is used to test the data value in the received tuple in the same way as tested in the Samza implementation. Based on the outcome of the test, a new tuple is made with two values: the speed value along with a noise flag, being flagged if the speed value fails the test, unflagged otherwise. The newly constructed tuple is then emitted to the output stream for further processing.

The source code for this Storm bolt is available in Appendix B.1.

3.3.2.4 Output Component

As with the Samza implementation, the Storm implementation of the designed pipeline does not require any concrete implementation of the output component. This is as Storm, by design, allows any extension to be made to a topology simply by implementing a further bolt which receives its tuples from an existing stream.

3.3.2.5 Deployment

Storm topologies are generally compiled and distributed in the same way as Samza projects; as JAR files containing compiled bytecode, assembled using Java build automation tools.

For the implementation of our designed pipeline, Maven was used to assemble a JAR file containing compiled bytecode, which is free to be deployed on different Storm installations.

3.3.3 Spark Streaming

For Spark Streaming, we have implemented the designed pipeline in both Python and Scala, running on the PySpark and Spark JVM systems, respectively. However, as the programming interface is mostly the same between both projects, the two implementations remain sufficiently similar, apart from being written in two different programming languages, that it is not needed to write about them separately. For parts of the implementation that differ between the two systems, they will be noted.

While many analogies can be made between the key components of Storm and Samza, due to the two project's similarities in design and usage, Spark Streaming presents itself

differently. Spark Streaming defines itself as an extension of its parent project, Spark, a big data batch processing system, which affords scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be streamed into Spark Streaming using a variety of supported sources, from complicated data systems, such as Kafka, to low-level sources, such as server sockets.

3.3.3.1 Programming Overview

Internally, Spark Streaming processes data quite differently to Samza and Storm, due to being an extension to the base Spark batch processing engine. Data streams are fed into Spark Streaming, which automatically partitions the streams into small batches which are then sent to the underlying Spark batch processing engine. The Spark batch processing engine then outputs a stream of results made up of multiple batches of data. A diagram illustrating this process is shown in Figure 3.5.

Fig. 3.5 Illustration [22] depicting how Spark Streaming runs on-top of the Spark batch engine.

This way of processing streams as a sequence of batches is all enabled through Spark Streaming's stream abstraction, discretised streams, or DStreams. It differs to how streams are conceptualised in Samza and Storm in the way that while streams are still, in essence, an unbounded sequence of data structures, being tuples in Storm, and message envelopes in Samza, these data structures are the same data structure which is used to represented data batches in Spark. These are, of course, the same resilient distributed dataset (RDD) data structure as explained in the previous chapter, in §2.3.3. Hence, Spark Streaming DStreams can be decomposed into a sequence of RDDs, making them fully compatible with batch-mode Spark [1].

For programming a Spark Streaming job, there are not a set of provided interfaces expected to be implemented, like in Samza or Storm. Instead, all of the computations to be performed on streams are performed on an instantiation of the `org.apache.spark.streaming.dstream.DStream` class returned from a DStream producing function from the `org.apache.spark.streaming.StreamingContext` class [1]. Processing performed on

DStreams are generally programmed in a dataflow fashion, with each function performed on the DStream returning a new DStream.

This way of operating on DStreams in a Spark Streaming program leads to all processing being contained in a single file, whereas in Storm or Samza processing would be split between tasks or bolts written in different files. This leads to more straightforward control over DStream transformations when programming, however also leads to much more tightly coupled logic. This will be a topic of interest later when evaluating the different DSPS technologies.

3.3.3.2 Input Component

For the implementation of the input component design in Spark Streaming, Spark Streaming provides a native server socket input system built-in on the `StreamingContext` class, available through the `socketTextStream` method. Hence this has been used to open a connection on a particular server socket for inputting of speed data in the same manner as in the Samza and Storm implementations.

3.3.3.3 Processing Component

For the implementation of the processing component design in Spark Streaming, a function has been written to test a single speed value, testing whether or not it is between the specified speed limits. Depending on the outcome of the test, a tuple is returned containing the original speed value, and a noise flag that is flagged if the data value is outside of the speed limit.

This speed test function then gets mapped to the existing socket connection DStream, effectively running the function for every speed value that is received over the connection.

3.3.3.4 Output Component

Unlike the Storm or Samza implementations, Spark Streaming offers no simple way to extend an existing streaming processing project through means of extra processing tasks that operate on existing streams. However, it does afford extensibility in the way of adding further processing on the DStream that is produced from previously performed operations, such as the previously mentioned speed test function mapping.

The source code for our Spark Streaming implementation can be seen in Scala in Appendix C.1, and in Python in Appendix C.2.

3.3.3.5 Deployment

Spark Streaming programs are compiled and distributed in the same fashion in which Samza and Storm projects are. However, for Spark Streaming programs written in Python using their relatively new PySpark API, compilation is unnecessary and an entire Spark Streaming Python project can be distributed and submitted and run on different installations of Spark Streaming straight from the Python source file. This greatly eases the distribution and deployment aspects of Spark Streaming, when using PySpark, in relation to Storm topologies and Samza projects.

For our Spark Streaming pipeline implementation written in Scala, SBT[4] was used to assemble a JAR file containing compiled bytecode, for ease of use over Maven with Scala projects. For the Spark Streaming pipeline implemented in Python, the Python source files were simply used for deployment. Both methods enable the deployment of the Spark Streaming pipelines on different Spark installations.

3.4 Conclusion

In conclusion this chapter has touched upon all details of implementation for each of the prototype pipelines. We have detailed how each prototype data filtering pipeline has been built for each candidate system, ready for subsequent tests and evaluation, of which will be discussed in the following chapter. Furthermore, the overall design of the pipeline that has been implemented has been detailed along with the details of the overall testing environment upon which the pipelines are to be implemented.

[4] http://www.scala-sbt.org/

Chapter 4

Evaluation and Discussion

Evaluation of each of the chosen DSPS technologies will be performed using both qualitative and quantitative testing methods of evaluation. The quantitative evaluation methods used will focus on the benchmarking of various features that are common to each of the technologies, using the pipelines implemented, as described in the previous chapter. The qualitative evaluation methods will focus on looking at the differences in ease-of-use, support for different programming languages and features, and complexity of code written to implement the pipelines on each of the different DSPS technologies.

With looking at the decision of giving a clear recommendation for a particular technology out of the choices stated, we think it is import to look at both qualitative and quantitative aspects for comparison. These systems are significantly non-trivial and vastly different in design and usage. However, as they still afford the same possible functionality, it is very possible to give a properly constructed evaluation of them. Not only is a factor such as performance a critical factor in forming a recommendation, but also are the possibilities each technology affords, in terms of programming and extensibility, when recommending a technology for use in such a large, and long production-life project.

The control for each of the evaluation measures will be the host system on which the different technologies are deployed, and the test data which is used for the relevant quantitative tests.

This chapter begins with an overview of the NeCTAR testing environment, in §4.1, upon which all tests are performed. An overview of all testing data to be used in quantitative tests will be given in §4.2. Quantitative tests and results are then described in §4.3, while qualitative tests and results are described in §4.4. Discussion of the overall test results and evaluation will then be given in §4.5. Subsequent DSPS technology recommendations will then be given in §4.6, based upon those results and discussion highlighted in previous sections, before concluding in §4.7.

4.1 Testing Environment

Each identically functioning pipeline built using the different candidate DSPS technologies are to be deployed in the same NeCTAR cloud instance as touched on in detail in the previous chapter, in §3.2. This testing environment also acts as the major control constant between each of the different pipelines that have been built, along with the test data that has been used for testing input.

The system details of the given instance are detailed in Table 3.1.

The NeCTAR cloud instance used utilises a single node upon which all processing is performed. In a real deployment, it is common to use a full cluster upon which the pipeline would be deployed, where processing is distributed between nodes. Due to limited testing resources, all testing has been performed on a single node cluster, however this is still a constant feature between all tests, and the pipelines built upon each of the existing technologies are freely configurable to be deployed to run on arbitrarily sized clusters without needing changes to the underlying source code.

4.2 Overview of Testing Data

As this sub-project focuses on the realtime processing of streaming data, data streams will have to be simulated from batch datasets acquired from the Monash IRT team. An initial dataset has been given that consists of sensor data in the categories as shown in Table 4.1.

The sample data batch acquired from the IRT team includes 99,999 rows of data recorded from each of the shown sensors, organised in a CSV file with headers. This is a general example of how data is currently received and handled by the IRT team, however in the case of realtime data processing, data would be received in quite a different manner. Rather than being received in large batches of readings, such as the sample dataset acquired, streams of data may be created from each particular sensor, delivering data values to the DSPS systems along input streams at infrequent intervals. Data is streamed in an asynchronous fashion, and can be simply thought of as a given DSPS system listens in on an incoming stream, and acts upon any data value in a specified fashion whenever they may be received.

Hence, to simulate streaming data from the data samples we have acquired, the DSPS pipeline is constructed to listen on a particular network socket connection for any incoming data. Once a data value is encountered on the connection, it is fed into the pipeline and processed accordingly, while the pipeline continues to listen on the socket for further incoming data. As the pipeline is constructed in such a way, we can simulate the data being sent out

Table 4.1 The different types of sensor data received from the Monash University Institute of Railway Technology team.

Name	Data type	Description
Time	Floating point value	Time in seconds since the start of the test.
Speed	Floating point value	Speed in kilometres per hour.
LoadEmpty	-1, 0, 1	1 if load is present, 0 if empty, or -1 if unknown/in-port.
km	Floating point value	Elapsed kilometres travelled since the start of the test.
Lat	Floating point value from -90 to +90	Geographic latitude coordinate at given reading.
Lon	Floating point value from -180 to +180	Geographic longitude coordinate at given reading.
Track	Integer value from 1 to 213 841	Given track in which the train is located.
SND1	Floating point value	Spring nest deflection count in millimetres at car corner.
SND2	Floating point value	Spring nest deflection count in millimetres at car corner.
SND3	Floating point value	Spring nest deflection count in millimetres at car corner.
SND4	Floating point value	Spring nest deflection count in millimetres at car corner.
CouplerForce	Floating point value	The amount of force couple detected in the train car.
LateralAccel	Floating point value	The amount of lateral acceleration detected in the train car.
AccLeft	Floating point value	The amount of acceleration detected by the left sensor.
AccRight	Floating point value	The amount of acceleration detected by the right sensor.
BounceFront	Floating point value	The amount of bounce detected at the front of the car.
BounceRear	Floating point value	The amount of bounce detected at the rear of the car.
RockFront	Floating point value	The amount of rock detected at the front of the car.
RockRear	Floating point value	The amount of rock detected at the front of the car.

from sensors using a program, such as `netcat`[1], where we can pipe data from a file to a particular port over a connection. We can also specify time intervals between each data value, simulating time between sensor readings.

As each sensor's data reading values are of the same format of floating point numbers, the specific sensor data to be used in the quantitative tests is not of major concern. Instead, we have chosen to focus on generating data based of the values from the speed sensor. As specified in Table 4.1, each value represents a speed reading formatted in kilometres per hour as a floating point number. For example, the data value `42.002` is equivalent to 42.002 kilometres per hour.

For the test data throughput time test, it requires multiple differently-sized large batches of test data for input to the pipelines. To generate these different data batches, we have extracted the speed readings from the provided sample dataset and replicated them a number of times to generate the large batches. Finally, this leaves us with five datasets containing speed readings of sizes 100,000, 500,000, 1,000,000, 5,000,000, and 10,000,000 data values, respectively.

For a response time to streaming data test, it only requires a single data value from a sensor reading for input to the pipelines. Hence a single floating point value of `42.002` has been used.

[1] http://netcat.sourceforge.net/

4.3 Quantitative Tests

4.3.1 Tests Performed

The quantitative methods to be used to evaluate the candidate DSPS technologies include mostly performance-based metrics. Each of the metrics that will be used is as follows:

- Response time to streaming data
- Test data throughput time
- Peak system resource usage
- System start-up time

Each of these can be quantified and compared easily between the different DSPS technologies via tracking various aspects of the systems.

4.3.1.1 Response time to streaming data

Response time to streaming data refers to the amount of time it takes for each DSPS pipeline to fully process a single value of sensor reading data. This time is defined from the time that value is sent along the network socket connection, to the time the given data value is pushed onto the pipeline's output stream. This can be measured using the runtime logs provided for each of the DSPS technologies. The purpose of this test is to look at how fast a given pipeline can respond to a sensor's reading. This is a highly realistic test for the sort of processing being looked at in the IRT project, as sensor readings are expected to arrive for processing at infrequent intervals, generally with a lot of time between each reading.

4.3.1.2 Test data throughput time

Test data throughput time refers to the amount of time it takes for each DSPS pipeline to fully process large batches of sensor readings. These data batches have been defined in §4.2. This is a less realistic test of the pipelines in relation to the IRT project, however it will be a stress-test of sorts, with the purpose of seeing how each candidate DSPS pipeline scales with larger amount of data being streamed in at the same time. Five different tests will take place, each with data batches of readings of size 100,000, 500,000, 1,000,000, 5,000,000, and 10,000,000, respectively. The throughput time is defined as the time from when the batches are piped along the network socket connection, to the time the pipelines have finished processing. These times are measured using the internal DSPS technology runtime logs.

4.3.1.3 Peak system resource usage

Peak system resource usage refers to the amount of resident system memory in-use while each of the pipelines are running. These values are expected to vary considerably during runtime, so samples will be taken at the peak system resource usage of each DSPS technology during the processing of our test data throughput time tests. Measurements will be taken using the UNIX tool `top` to collect readings for the amount of resident memory being used by the DSPS technology.

4.3.1.4 System start-up time

System start-up time relates to the amount of time taken for each candidate DSPS pipeline to start-up. This time is defined from the time each DSPS technology is started via the system shell, to the time the DSPS technology is ready to accept incoming data streams. This will be measured using the UNIX `time` command, and allowing the DSPS technology to terminate upon readiness. This is considered to be a less important test, as these systems are expected to be deployed and run for long continuous periods of time. Start-up is generally not an important consideration, however we are interested to see how each DSPS technology compares, and how these results relate to the results from other tests.

4.3.2 Test Results

4.3.2.1 Response time to streaming data

The results to the response time to streaming data test are shown in Figure 4.1.

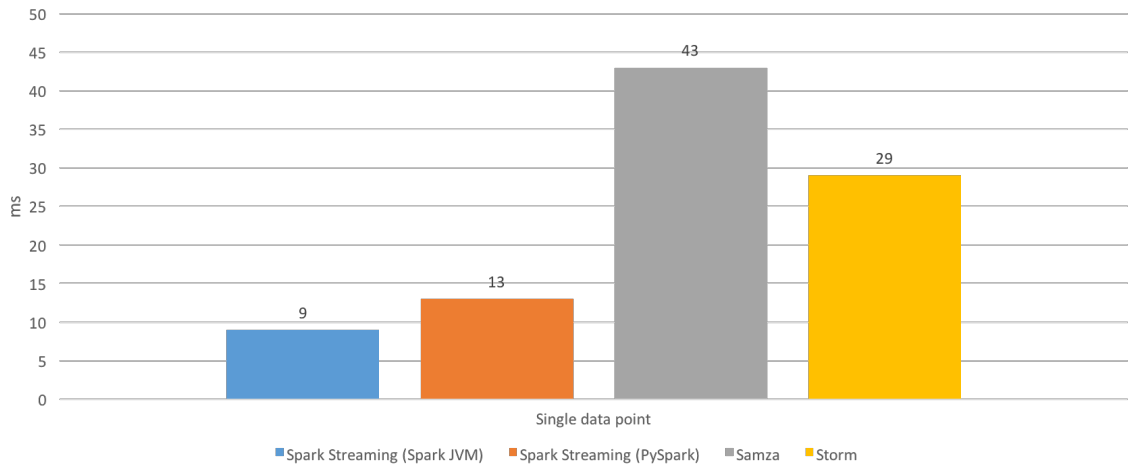


Fig. 4.1 Results of our response time to streaming data test.

Essentially, what can be can be seen from these results is that both Spark Streaming APIs exhibit relatively good performance compared to both Storm and Samza. In particular, Spark Streaming running on the Spark JVM API exhibited over 3× the performance of the next big system, Storm, and nearly 5× the performance of the slowest contender, Samza. Further of note is that at this size of data, a single sensor reading, both Spark Streaming APIs exhibit very similar performance in relation to the other DSPS technologies.

As Spark Streaming is essentially an extension built on top of a batch processing system, the fact that it outperformed both Samza and Storm, two technologies built from the ground up with the aim of processing data in realtime, is quite a surprising find.

4.3.2.2 Test data throughput time

The results to the test data throughput time tests are shown in Figure 4.2.

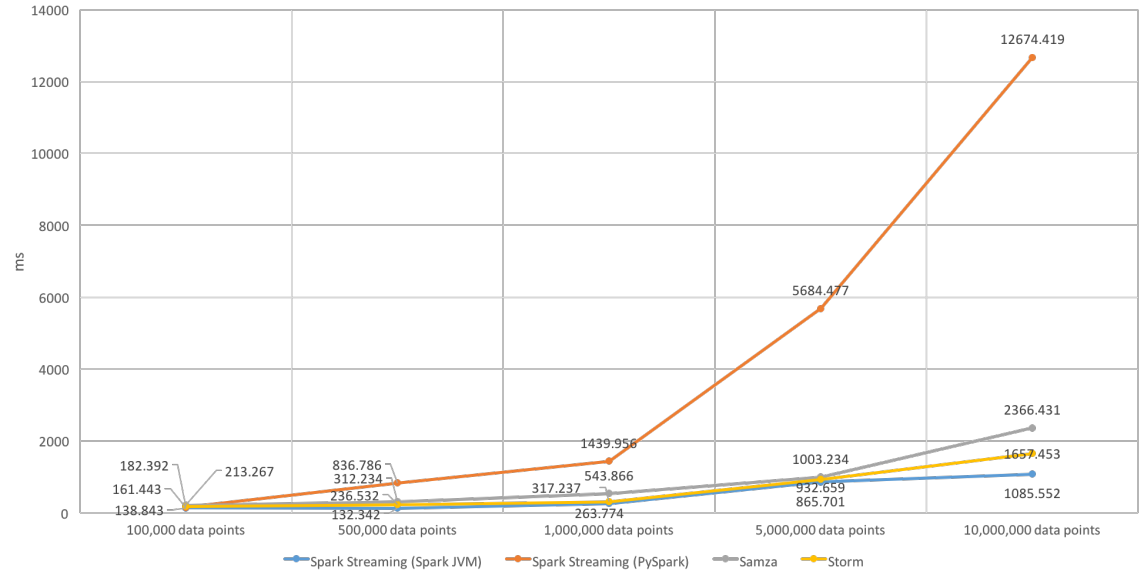


Fig. 4.2 Results of our test data throughput time tests.

The first fact to note with these results relates to the performance of Spark Streaming, running on the PySpark API, as the input data increases. Compared to the other technologies, Spark Streaming (PySpark) exhibits very poor scaling performance when dealing with much larger amounts of input data, particularly with 5,000,000 data points and greater. This drop in performance is so much of an issue that at the scale at which the results are shown, it is hard to make out differences in performance for the other systems. Hence, to better analyse the results of the other systems, a chart from the same data was created with the Spark Streaming (PySpark) results omitted. This can be viewed in Figure 4.3.

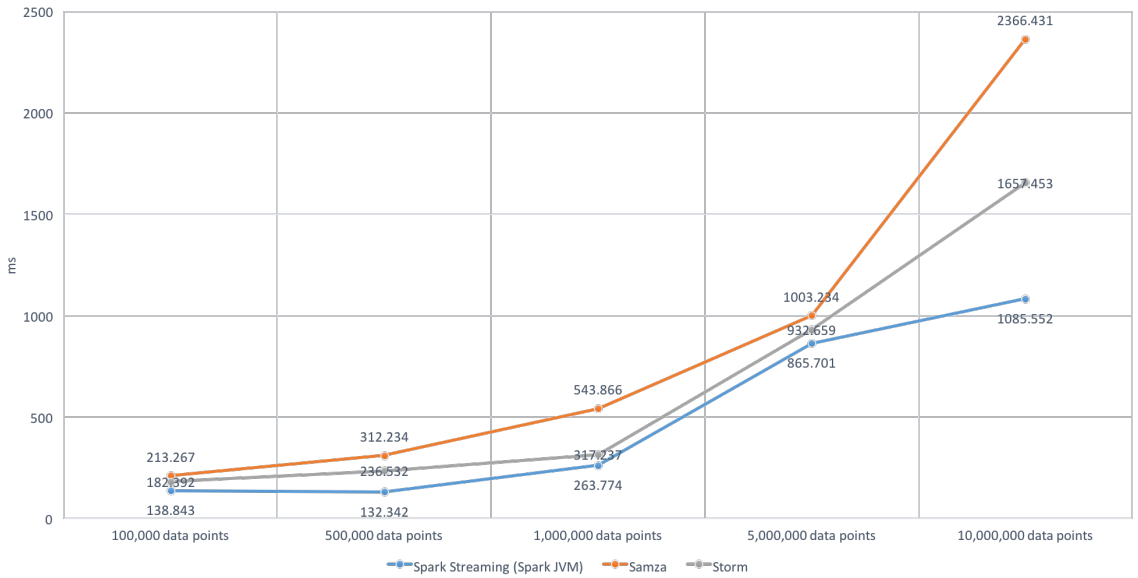


Fig. 4.3 Results of our test data throughput time tests (PySpark omitted).

Looking the results of Spark Streaming (Spark JVM), Storm, and Samza, for the most part they show a mostly similar performance at scale, relative to what was seen with Spark Streaming (PySpark). Spark Streaming shows the greatest performance, with greater performance than both Storm and Samza in all tests, while Storm generally is not too far off Spark Streaming's performance, showing a similar growth for all tests apart from the 10,000,000 data points test.

Again, Samza shows the worst overall performance out of the three candidates (not counting PySpark), with less than half the performance of Spark Streaming on the 10,000,000 data points test.

4.3.2.3 Peak system resource usage

The results for the peak system resource usage test can be seen in Figure 4.4.

Interestingly, Spark Streaming excels again here, with JVM Spark having over double the performance from the next major contender, Storm. Samza, at peak usage, takes up over 5× the amount of resident memory that Spark Streaming does, and nearly 3× the amount that Storm is shown to use. A lot of this resident memory usage may be caused by a lot of the software dependencies that Samza and Storm rely on, which Spark Streaming generally does not. Spark, the base project, not the realtime processing extension, was designed to be an entire software ecosystem in itself, hence a lot of external dependencies that Storm and Samza may have are not needed, possibly leading to this extreme reduction in resident memory usage shown here.

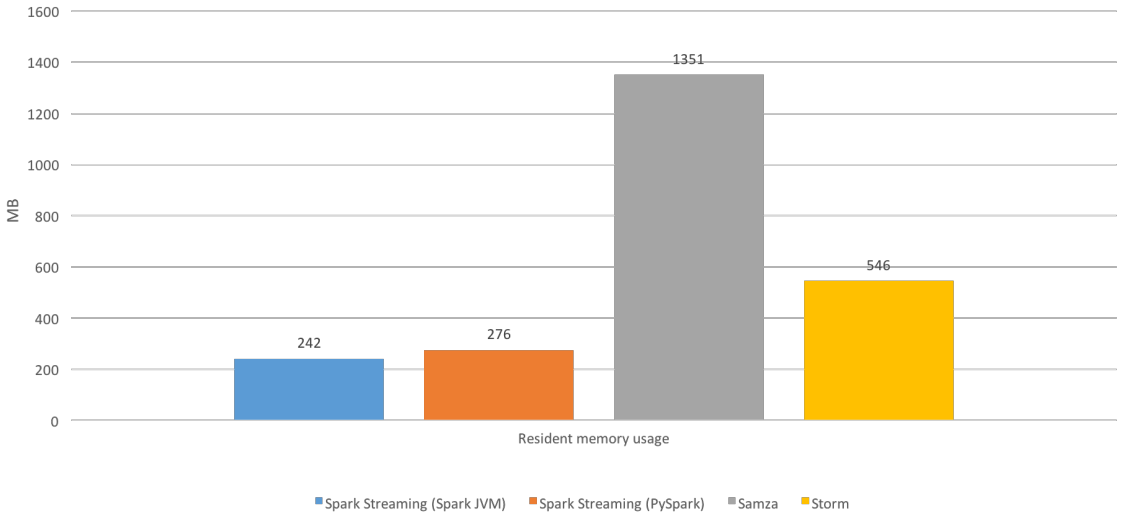


Fig. 4.4 Results of our peak system resource usage test.

A further point of interest here is that Spark Streaming, and thus Spark, is renowned for its in-memory computation model, where data is placed solely in resident memory during computation. This design decision would lead to the expectation of Spark computations using far more resident system memory than other technologies that do not make such use of resident memory. However, as seen here, Spark Streaming is shown to use the least amount of resident memory. This may be to do with the amounts of data used. For example, a more data-intensive workload would likely provide much higher resident system memory usage in Spark than that of Storm or Samza.

4.3.2.4 System start-up time

The results for the system start-up time test can be seen in Figure 4.5.

Again, we emphasise that this test is of much less importance than the previously covered quantitative tests, due to real-life usage of the DSPS technologies. However, it is interesting of note here that Spark Streaming, the overall best performing DSPS technology in all other performance-based tests, shows the worst performance in-terms of start-up time, particularly the PySpark version of Spark Streaming. Samza, the overall worst performing technology, here shows the best performance, with being able to be ready for processing from start-up in just over 4 seconds.

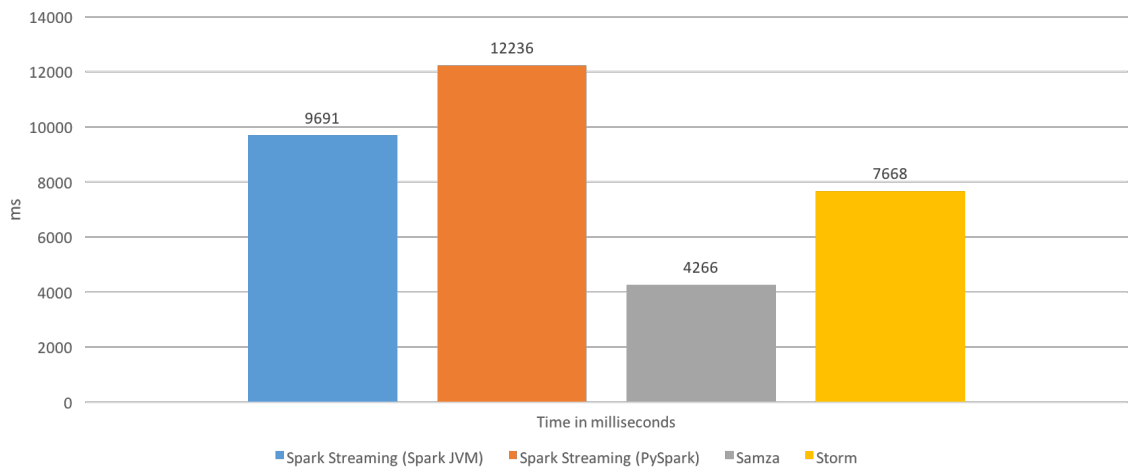


Fig. 4.5 Results of our system start-up time test.

4.4 Qualitative Tests

4.4.1 Tests Performed

The qualitative methods used to evaluate the candidate DSPS systems include mostly programmer-centric metrics, showing the programming possibilities for each technology. Each of the metrics that will be compared is as follows:

- Ease of extensibility
- Available DSPS features
- Ease of programmability
- Ease of deployment

As each of the underlying DSPS technologies are considerably different in design and usage, it is important to see how they contrast in terms of factors that they afford.

4.4.1.1 Ease of extensibility

Ease of pipeline extensibility refers to the methods used to extend an existing pipeline, or project, existing in each of the candidate DSPS technologies. For example, if an existing project is already in use in production, we will be measuring the ease that each technology allows that project to be extended in the ways of adding further processing tasks to the existing system, or restructuring an existing project processing layout (for example, a Storm topology layout). Parameters we will be looking at include whether or not an entire project

needs to be recompiled and deployed to be extended, and the ease of modifying overall system configurations.

4.4.1.2 Available DSPS features

Available DSPS features refers to the features each candidate DSPS technology supports as they relate to Stonebraker, Çẽntintemel, and Zdonik's eight requirements for realtime data processing systems [83], as looked at in the literature review chapter in §2.4. These requirements are based on research performed many years prior to the release of each of our candidate DSPS technologies, and most realtime data processing technologies in general. The supported features for each technology will be compared to show the functional possibilities natively available in each technology.

4.4.1.3 Ease of programmability

Ease of programmability refers to a number of different features including official programming language support, complexity of required interfaces to be implemented, and complexity of required system configuration. To summarise, we will look at the difficulty and learning-curve that may need to be overcome to program each candidate DSPS technology. Measuring this presents a more difficult, and somewhat subjective, task compared to the metrics used in the quantitative tests, however these will be measured using impartial comparisons of the methods which each DSPS technology affords to be programmed.

4.4.1.4 Ease of deployment

Ease of deployment refers to the methods in which each candidate DSPS technology affords its projects to be deployed and distributed. Similar to the ease of programmability test, this metric will be measured by an impartial comparison of each possible method of project deployment and distribution.

4.4.2 Test Results

4.4.2.1 Ease of extensibility

Spark Streaming is noted as the worst performing technology in terms of extensibility, overall. This is mostly due to the fact that, by design, a Spark Streaming program is generally very tightly coupled in terms of processing modules. All processing logic revolves around the `StreamingContext` object in a Spark Streaming program, hence everything is tightly integrated to the placement of this object. Of course, this object can be passed around

different modules, however it is a rather naïve solution considering the other projects exhibit loose coupling by design.

This tight coupling of logic in Spark Streaming leads to any extensions written to an existing Spark Streaming project to be written as modifications of the original source code. This means that previously written source code need to be made aware of the changes, and recompiled.

Likewise, in Spark Streaming, configuration is tightly coupled with the programming logic, through use of the `SparkConf` object, on which the `StreamingContext` object is instantiated. This leads to any future changed in configuration to be written to the original source code, thus leading to project recompilation.

Samza is noted to be far more loosely coupled than that of Spark Streaming, with any extensions of an existing project being able to be written separately of the original code. Essentially, a Samza project's processing tasks will output their results to specific Samza streams. On these streams, a new extension can be written to listen on, without the existing processing tasks needing to be aware of this new extension task. This loose coupling of processing tasks is by design in Samza, with tasks existing in a dataflow-like sequence, where tasks can be added and removed with ease.

For configuration in Samza, processing tasks and systems are configured separately to the underlying logic, using a non-compilation necessary medium. Effectively, this means that Samza projects can be reconfigured on-the-fly without recompilation of the entire project. This is an important factor which Spark Streaming lacks native support for.

Storm is very similar to Samza in the way that extensions can be written to extend an existing Storm topology without modification of the existing source code being necessary. However, extensions written to an existing Storm topology do not require knowledge of output streams from existing Storm processing tasks (bolts), which is a necessity when extending a Samza project.

Configuration in Storm is also kept separate of the logic written in the source code, hence also does not require recompilation to reconfigure. However, to reconfigure an existing Storm topology layout, this is generally written in source code using the `TopologyBuilder` class, hence does require a recompilation step.

To summarise, Spark Streaming has shown to lack in extensibility, compared to the other candidate DSPS technologies, mainly due to being tightly coupled by design, while Storm and Samza exhibit similar ease in terms of extensibility. Samza and Storm extensions can be written separate of existing logic, without the existing logic being aware of the extensions. This is, unfortunately, not the case with Spark Streaming.

4.4.2.2 Available DSPS features

As these features were looked at previously in the literature review chapter, in §2.4, this table has been reconstructed in Table 4.2, with S4 omitted due to being excluded from our candidate DSPS systems. Furthermore, certain features that were deemed as inconclusive through analysis of existing literature have been decided upon through our further understanding of the technologies through our implementation phase. These features, that have changed from the table that was given in §2.4, have been marked in Table 4.2 in bold.

Table 4.2 Available features in each candidate DSPS technology.

	Storm	**Spark Streaming**	**Samza**
1: Data mobility	Fetch	Micro-batch	Fetch (Kafka)
2: SQL on streams	Extension (Trident)	Yes (Spark SQL)	No
3: Handling stream imperfections	User responsibility	N/A	Yes (Kafka)
4: Deterministic outcomes	N/A	N/A	N/A
5: Stored and streaming data	Yes (Trident)	Yes (Spark)	No
6: High availability	Yes (rollback recovery)	Yes (checkpoint recovery)	Yes (rollback recovery)
7: Partition and scaling	User responsibility	No	Yes (Kafka topics)
8: Instant response	**Yes**	**Yes**	**Yes**

In our work, the majority of these features were not of concern as they were not required for our prototype implementation. Our implementation focused solely on the processing of independent, unordered streaming data, hence **SQL on streams**, **Stored and streaming data**, both relating to batch processing capabilities, and **Handling stream imperfections**, relating to interdependent, or sequential data, were out of scope. Furthermore, as our implementation was only evaluated using a single-node cluster test environment, **Partition and scalability** was of no concern. However, we will touch upon some of these features, as they are a concern to the IRT project's aims as a whole.

One obvious point of note, at first glance of the results, is that both Spark Streaming and Samza lack many of the realtime processing features that are present in Storm. This may be put down to the fact that both Spark Streaming, as an extension of the greater project Spark, and Samza are still relatively immature projects. In fact, Samza only became accepted as a top-level project at the Apache Software Foundation in January of 2015 [16], after spending years as an Apache Incubator project. It may also be attributed to design choices of developers of both projects.

Furthermore, of note is that many features that are supported in Samza actually depend on the integration of Kafka with Samza. If Samza is run standalone, without Kafka, certain features may not be available unless replaced with a similar Samza-compatible pluggable system. Due to the nature of Samza and Kafka's tight coupling in most installations, a lot of the time, the availability of these features, off-loaded to Kafka, can be taken for granted.

In relation to the IRT project, the most important features that these DSPS systems support are arguably **SQL on streams** and **Stored and streaming data**. These features refer to the possibility of running more traditional batch processing functions, such as aggregate functions like map reduce, on the data streams that are being processed otherwise in realtime. As the IRT project at current uses a lot of batch processing applied to their data, it would be much more ideal if the DSPS technology to be used also had the possibility of affording this type of processing in conjunction with realtime processing.

From the results, it is shown that both Storm and Spark afford these features. Storm, via the Trident extension, allows both batched data processing through aggregate functions, and an SQL-like query language [15]. Spark Streaming, being built on top of the batch processing project, Spark, natively supports batch processing by off-loading to Spark, and an SQL-like query language through the new Spark SQL extension [20]. Conversely, Samza does not afford any native type of batch processing functionality, or any SQL-like query language at the time of writing.

Regarding the features previously deemed to be inconclusive, from a review of existing literature, both **Partition and scaling** and **Instant response** features have been noted as having findings. Through our tests performed with large batches off data being streamed in all at once, we have noted that all candidate DSPS technologies exhibit instant responses to data at all scales.

4.4.2.3 Ease of programmability

Spark Streaming has been noted as being the most accessible of the candidate DSPS technologies in terms of programmability. Spark Streaming, as of version 1.2.0, has official support for Java, Scala, and Python (however some of the functionality deemed less important is yet to be implemented in the Python API). Storm has native support for Java and Clojure, and essentially any programming language through use of a Thrift definition file[2], however official documentation is only provided for Java and Clojure. Samza only has native support for Java.

Writing Spark Streaming projects, and also Storm and Samza projects, in JVM languages of course require compilation to JVM bytecode before they are to be run, and also having the

[2] https://thrift.apache.org/

requirement that the compiled bytecode be packaged as a distributable JAR archive. Python use with PySpark, being an interpreted language, allows for Spark Streaming projects to be written and run directly from the Python source files deployed to a Spark installation. This has been noted to greatly ease the aspect of getting a new Spark Streaming project up and running, without many compilation and assembly steps that are required in the JVM language DSPS technologies.

Storm and Samza have been noted as having a far higher learning curve to that of Spark Streaming, due to the number of interfaces which are required to be implemented to make even the most simple of projects, and the required configuration that is necessary for the project to run properly. Spark Streaming does away with these interfaces, and instead all processing logic surrounds the `StreamingContext` class, meaning that as long as this class is instantiated, all the possible processing functions are available to be used. Thus, a simple processing pipeline based on simple processing logic implemented in each system would require a number of different files, each implementing required interfaces, in Storm and Samza, while the same logic in Spark Streaming could be done in a number of lines in any supported programming language.

Samza, in particular, has been noted as having the most steep of the learning curves, in terms of programmability, due to a required knowledge of many of the underlying software dependencies, such as Kafka, ZooKeeper, and various serialisers and deserialisers for Samza messages. Required configuration of a Samza project requires an understanding of these systems for even the most simple of processing tasks, while most of the underlying systems in Storm and Spark Streaming are abstracted away from the programmer, who generally does not need to worry about these unless performing more advanced tasks.

In terms of required configuration, Spark Streaming allows for the most ease in configuring, due to a low level understanding of Spark configuration not being needed for most tasks. Storm also has somewhat of a easy required configuration knowledge, mostly due to having the more complicated configuration options being abstracted away from the programmer unless necessary, and the majority of configuration being defined in the topology file. Samza, however, requires separate, and particularly verbose, configuration files for every task and system within a Samza project, as well as many configuration options not having defaults. Again, this leads to a good understanding of Samza being needed to configure even a simple project, and a lot of configuration to be thought about, particularly if a Samza system expands to be greater than a small number of different processing tasks.

In summary, Spark Streaming has been noted as the most simple of the candidate DSPS technologies in terms of programmability and overall learning curve. Simple projects are able to be configured and written in a small amount of lines of code. Storm presents the next most

simple technology with a lot of the configuration and software dependencies being abstracted away from the programmer. Samza comes in last, with the largest learning curve, due to a good understanding of the overall Samza project, along with Samza software dependencies needed to implement even a simple Samza project.

4.4.2.4 Ease of deployment

Projects written for each of the DSPS technologies are expected to be deployed and distributed in similar ways. As each of the technologies are built on JVM technologies, they require compiled JVM bytecode to be packaged as a JAR file, generally using a build automation tool, such as Maven[3] or Gradle[4]. However, for deployment and distribution of Spark Streaming projects written in Python, these compilation and build assembly steps are not necessary, with Python and Scala Spark Streaming projects affording deployment simply as the Python or Scala source files themselves. This greatly eases overall deployment, and distribution of Spark Streaming projects between different Spark Streaming installations residing on different clusters.

Furthermore, Storm and Samza require the set-up of external technology dependencies, such as Kafka or ZooKeeper, on the cluster on which they will be run, if the implemented project requires such dependencies. Spark Streaming can be deployed without external technology dependencies as the Spark project provides Spark distributions with all such dependencies built-in and pre-configured [17].

4.5 Discussion

Looking over the results of both the quantitative performance-based results, and the qualitative feature-based results, a few points are of note in regards to each of the candidate DSPS technologies. First, Spark Streaming has shown the overall greatest performance while managing to have the lowest resident memory footprint. Both the Spark JVM and PySpark based Spark Streaming versions show great performance, however the Spark JVM version shows the best performance overall given that the PySpark version does not scale well when dealing with larger amounts of data being streamed in. Spark Streaming also showed the lowest learning curve and was the overall easiest technology to write realtime processing logic in.

However, while Spark Streaming has been shown to have the greatest performance through the results of the quantitative tests, it very much falls short in relation to ease of

[3] https://maven.apache.org/
[4] https://gradle.org/

extensibility due to the tightly coupled nature of Spark Streaming projects. As all realtime processing logic is focused around a single class which affords the DStream functionality, any extensions written to alter an existing project require the modification of the original project, tightly integrating the extension logic into the existing logic.

Storm, on the other hand, showed the next best performance and resident memory footprint after Spark Streaming, through the results of the quantitative tests. The performance of Storm has been shown to be closer to that of Spark Streaming, JVM version, rather than that of Samza. Furthermore, Storm affords the next best option in terms of learning curve and ease of writing realtime processing logic to that of Spark Streaming, and it has shown to be a much more mature project to Samza, with a lot more available documentation.

Samza has shown to perform badly in almost all the tests, both qualitative and quantitative, apart from ease of extensibility where it beat Spark Streaming. This is an unfortunate result from the most immature of all the candidate DSPS technologies, however it is not an overly surprising outcome.

4.6 DSPS Technology Recommendations

As discussed, while Spark Streaming has been shown to have the overall greatest performance of the three candidate DSPS technologies, due to its failing, in terms of extensibility, we cannot recommend it for use in the Monash University IRT project. This stems back to our third research question for this project, looking at the possibility of extending projects developed in each of the candidate DSPS technologies.

As Storm has shown the second best performance to Spark Streaming, the second best outcome in most qualitative tests, and shown to be the overall most extensible of the three technologies, it is a much better contender for use in the Monash University IRT project.

We cannot recommend the use of Samza, in light of its performance in relation to the other two technologies in all tests. Both of the other candidate technologies provide better outcomes overall in most areas.

Hence, we are recommending the use of Storm in the Monash University IRT project. We believe that it exhibits all the characteristics required for a DSPS technology in the IRT project, and exhibits relatively positive performance. It affords use in an advanced project, subject to many changes, such as the IRT project, and it is also the most mature of the three candidate technologies, allowing for much documentation and support available to be found on the Internet.

Spark Streaming is our runner-up recommendation, showing great performance, and a low learning curve. We recommend Spark Streaming to be used in simple realtime processing projects, where change and extension may be unnecessary or not required often.

4.7 Conclusion

In conclusion, we have nominated Storm as our recommended data stream processing system technology for use in the Monash University Institute of Railway Technology project. Overall, we have shown it to exhibit great performance and features in terms of programmability and extensibility. It has shown to have a steeper learning curve to that of Spark Streaming, however Spark Streaming has shown to be a less viable option for large, and constantly changing projects, such as the IRT project.

This nomination as our recommended DSPS technology comes after performing many suitable tests, using both quantitative performance-based metrics and qualitative use-based metrics, and evaluating it in comparison to our other candidate DSPS technologies, Samza and Spark Streaming.

Chapter 5

Conclusion

This final chapter concludes this thesis by looking at a summary of the contributions made from our research in this project, along with identifying possible areas of interest for any future work relating to this project. As this project has a large overall scope, and the research discussed in this thesis only relates to a small part, many further opportunities will present themselves for future work with the overall Monash University Institute of Railway Technology project.

This chapter begins with a summary of our research contribution, in §5.1, before identifying possible future work, in §5.2.

5.1 Research Contribution

In this research, we gave a recommendation of a data stream processing system (DSPS) for realtime data processing use in the Monash University Institute of Railway Technology (IRT) project. This project focuses around railway track condition monitoring through the use of sensors placed on train cars. At inception of this research, the project had no realtime data processing functionality integrated with it, however the engineering team working on the project expressed interest in using realtime data processing for use with numerous functions required in their use-case; mostly pre-processing of sensor readings. Hence, the aim of recommending an appropriate DSPS for use in the IRT project is to address these usage requirements, allowing for such realtime processing to be used.

In the case of the IRT project, the DSPS technologies act as a pipeline through which data flows, all the way from the sensor source to the destination. The destination is not specified, in the case of the IRT project, hence is left abstract for later extension. Suitable destinations for data processed in realtime are data stores, such as a database management system or distributed file system, or further processing, in the way of another DSPS pipeline

or even batch processing systems. This pipeline requires design, which often differs between different DSPS technologies, to adhere to the processing requirements of the use-case, and consequently implementation.

In this research project, a very basic prototype pipeline has been designed and implemented to filter speed sensor readings. This prototype pipeline has been designed in such a way that it is extensible and re-usable for different use-cases in the IRT project. This pipeline has been implemented for use of evaluating the different candidate DSPS technologies, Storm, Samza, and Spark Streaming, however it is also possible to use as a base-line for the design and implementation of the real pipeline to be used in production in the IRT project.

This prototype pipeline implemented in this research has been used for testing and consequently evaluation of the different candidate DSPS technologies, allowing us to arrive at the recommendation of the Storm DSPS for use in the IRT project. Storm shows great overall performance in processing data in realtime, and also allows for extensible projects to be created, suitable for large projects, intended to be in production for long periods of time, and subject to change. Furthermore, Storm affords a great number of possible realtime and batch processing features, while exhibiting a relatively low footprint on system resources.

Apart from the IRT project, which Storm is now intended to be used in production for, Storm has been in use for a number of years for a range of different applications in numerous companies such as Twitter, Yahoo!, Baidu, and Spotify [9].

In summary, in relation to our original research questions, this research has answered all questions and met the aims of the project. We have identified existing DSPS technologies for use in the IRT project through a review of existing literature in the area of realtime data processing. These technologies include Storm, Samza, Spark Streaming, and S4. We have identified a number of qualitative and quantitative metrics to evaluate each of the technologies based on the implementation of a prototype realtime data processing pipeline in each of the technologies. For implementation, the candidate technologies were narrowed to focus on Storm, Samza, and Spark Streaming due to the finding of S4 being effectively an abandoned project in our review of existing literature. Finally, the prototype realtime data processing pipeline that was implemented was designed to be extensible, allowing the addition of further realtime processing logic to be implemented as changes to the IRT project become apparent.

5.2 Future Work

During this research, potential areas for future work and research have been identified. These focus around looking at the batch processing of data on the candidate DSPS technologies that support it, the evaluation of different DSPS technologies on different cluster set-ups, and

the evaluation of the use of the prototype pipeline in the overall IRT project. We discus these briefly.

Given that this research was focused solely around the realtime processing features of each of the candidate DSPS technologies, it would be of further interest to look at both Spark Streaming and Storm's handling of batch data processing. As noted in the evaluation chapter, both Spark Streaming, through the Spark batch processing engine, and Storm, through its Trident extension, support batch processing performed on streaming data. As batch processing remains an important task in many data-intensive applications, it would be of interest to evaluate these technologies based upon this factor, and also in relation to existing batch processing systems, such as those that are categorised under the Hadoop ecosystem. Of course, this would be of interest to users of a different processing use-case to that which was focused on in this research, hence would not be appropriate for making a *realtime* processing DSPS technology recommendation.

In relation to realtime data processing, further work of interest, building upon our evaluation of DSPS technologies, would be to evaluate each of the DSPS processing technologies running across differently configured clusters of different node size. The research performed in this project revolved around a single cluster set-up, being a single node cluster, used as a constant factor between all tests. Looking at further cluster set-ups, this would allow an informed evaluation on how each technology scales in different environments. Of interest would be the differences in performance, and the ease of distributed and parallel programming, afforded by the different DSPS technologies across different cluster set-ups.

A notable change in the landscape of realtime data stream processing technologies, within the time of writing this thesis, has been the announcement of a new realtime DSPS, Twitter Heron, produced in-house at Twitter [37]. This technology was specifically created with the intention of replacing Storm functionality at Twitter [63]. A paper, published by Kulkarni, et al. in early June, 2015, notes Twitter Heron exhibiting far greater performance than Storm when used in the same context [63]. At current, the Twitter Heron project remains closed-source and only for use at Twitter, however given the possibility of open-sourcing and release for use outside of Twitter, Heron would provide an additional platform on which to evaluate Storm against for use in the IRT project.

Lastly, it would be of interest to evaluate the prototype pipeline designed and implemented in this research, running on our recommended DSPS technology, Storm, while in use with the entire IRT project ecosystem. Hence, it would be needed to be put in place, between the sensors located on train cars, and the data storage layer, allowing the performance of it to be evaluated. It may also be of interest to evaluate the recommended Storm pipeline

against those pipelines implemented in the other candidate DSPS technologies, that were also a result of this research, in relation to their use in place in the IRT project.

References

[1] Spark Streaming - Spark 1.3.0 Documentation. https://spark.apache.org/docs/latest/streaming-programming-guide.html. (Visited on 2015-06-02).

[2] Storm documentation. https://storm.apache.org/documentation/Home.html. (Visited on 2015-06-02).

[3] Storm, distributed and fault-tolerant realtime computation. http://storm-project.net, November 2013.

[4] About storm. https://storm.apache.org/about/multi-language.html, 2014. (Visited on 2015-05-12).

[5] Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more. http://www.amazon.com, August 2014.

[6] AMPLab - UC Berkeley | Algorithms, Machines and People Lab. https://amplab.cs.berkeley.edu, 2014.

[7] Apache Spark™– Lightning-Fast Cluster Computing. https://spark.apache.org, 2014.

[8] Australian Synchrotron. https://www.synchrotron.org.au, August 2014.

[9] Companies Using Apache Storm. https://storm.apache.org/documentation/Powered-By.html, 2014.

[10] home | NeCTAR. http://www.nectar.org.au, August 2014.

[11] Institute of Railway Technology. https://platforms.monash.edu/irt/, March 2014. (Visited on 2015-05-09).

[12] The Large Hadron Collider | CERN. http://home.web.cern.ch/topics/large-hadron-collider, August 2014.

[13] Samza. http://samza.incubator.apache.org, 2014.

[14] Spark Streaming | Apache Spark. https://spark.apache.org/streaming/, 2014.

[15] Trident API Overview. https://storm.apache.org/documentation/Trident-API-Overview.html, 2014. (Visited on 2015-09-06).

[16] The Apache Software Foundation Announces Apache™Samza™as a Top-Level Project : The Apache Software Foundation Blog. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces71, January 2015. (Visited on 2015-09-06).

[17] Downloads | Apache Spark. https://spark.apache.org/downloads.html, April 2015. (Visited on 2015-09-06).

[18] Samza - API Overview. https://samza.apache.org/learn/documentation/0.9/api/overview.html, 2015. (Visited on 2015-05-12).

[19] Samza - Documentation. https://samza.apache.org/learn/documentation/0.9/, 2015. (Visited on 2015-06-01).

[20] Spark SQL | Apache Spark. https://spark.apache.org/sql/, 2015. (Visited on 2015-09-06).

[21] Spark Streaming - Spark 1.3.1 Documentation. https://spark.apache.org/docs/1.3.1/streaming-programming-guide.html, 2015. (Visited on 2015-05-12).

[22] streaming-flow.png (1071x239). https://spark.apache.org/docs/1.3.1/img/streaming-flow.png, 2015. (Visited on 06/10/2015).

[23] AGRAWAL, D., DAS, S., AND EL ABBADI, A. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology* (2011), ACM, pp. 530–533.

[24] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., ET AL. Spark SQL: Relational Data Processing in Spark.

[25] ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., ET AL. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS) 1*, 2 (1976), 97–137.

[26] BANGE, C., GROSSER, T., AND JANOSCHEK, N. Big Data Survey Europe - Usage, technology and budgets in European best-practice companies. http://www.pmone.com/fileadmin/user_upload/doc/study/BARC_BIG_DATA_SURVEY_EN_final.pdf, 2013.

[27] BIFET, A. Mining big data in real time. *Informatica 37*, 1 (Mar. 2013), 15+.

[28] BOCKERMANN, C. A Survey of the Stream Processing Landscape.

[29] BOHLOULI, M., SCHULZ, F., ANGELIS, L., PAHOR, D., BRANDIC, I., ATLAN, D., AND TATE, R. Towards an integrated platform for big data analysis. In *Integration of Practice-Oriented Knowledge Technology: Trends and Prospectives*. Springer, 2013, pp. 47–56.

[30] BORTHAKUR, D. The Hadoop Distributed File System: Architecture and design. *Hadoop Project Website 11* (2007), 21.

[31] BORTHAKUR, D. HDFS architecture guide. *Hadoop Apache Project* (2008), 53.

[32] BORTHAKUR, D., GRAY, J., SARMA, J. S., MUTHUKKARUPPAN, K., SPIEGELBERG, N., KUANG, H., RANGANATHAN, K., MOLKOV, D., MENON, A., RASH, S., ET AL. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 1071–1080.

[33] CHAMBERLIN, D. D., AND BOYCE, R. F. Sequel: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (1974), ACM, pp. 249–264.

[34] CHAUHAN, J., CHOWDHURY, S. A., AND MAKAROFF, D. Performance evaluation of Yahoo! S4: A first look. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on* (2012), IEEE, pp. 58–65.

[35] CHUN, B.-G., CONDIE, T., CURINO, C., DOUGLAS, C., MATUSEVYCH, S., MYERS, B., NARAYANAMURTHY, S., RAMAKRISHNAN, R., RAO, S., ROSEN, J., ET AL. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment 6*, 12 (2013), 1370–1373.

[36] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (1970), 377–387.

[37] COLYER, A. Twitter heron: Stream processing at scale | the morning paper. http://blog.acolyer.org/2015/06/15/twitter-heron-stream-processing-at-scale/, June 2015. (Visited on 2015-06-16).

[38] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., GERTH, J., TALBOT, J., ELMELEEGY, K., AND SEARS, R. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 1115–1118.

[39] DARBY, M., ALVAREZ, E., MCLEOD, J., TEW, G., AND CREW, G. The development of an instrumented wagon for continuously monitoring track condition. In *AusRAIL PLUS 2003, 17-19 November 2003, Sydney, NSW, Australia* (2003).

[40] DARBY, M., ALVAREZ, E., MCLEOD, J., TEW, G., CREW, G., ET AL. Track condition monitoring: the next generation. In *Proceedings of 9th International Heavy Haul Association Conference* (2005), vol. 1, pp. 1–1.

[41] DARLIN, D. Airfares made easy (or easier). *The New York Times 1* (2006), B1.

[42] DAVIDSON, A., AND OR, A. Optimizing Shuffle Performance in Spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013).

[43] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[44] DEAN, J., AND GHEMAWAT, S. MapReduce: a flexible data processing tool. *Communications of the ACM 53*, 1 (2010), 72–77.

[45] EVANS, B., AND FENG, A. Storm-YARN Released as Open Source. https://developer.yahoo.com/blogs/ydn/storm-yarn-released-open-source-143745133.html, 2013.

[46] EVANS, B., AND GRAVES, T. Yahoo compares Storm and Spark. http://www.slideshare.net/ChicagoHUG/yahoo-compares-storm-and-spark, 2014.

[47] GATES, A. F., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAMURTHY, S. M., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1414–1425.

[48] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[49] GITHUB. yahoo/storm-yarn. https://github.com/yahoo/storm-yarn, 2014.

[50] GOETZ, P. T. Apache Storm Vs. Spark Streaming. http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming, 2014.

[51] GROSSMAN, R. L., AND GU, Y. On the Varieties of Clouds for Data Intensive Computing. *IEEE Data Eng. Bull. 32*, 1 (2009), 44–50.

[52] GRUENHEID, A., OMIECINSKI, E., AND MARK, L. Query optimization using column statistics in Hive. In *Proceedings of the 15th Symposium on International Database Engineering & Applications* (2011), ACM, pp. 97–105.

[53] HADOOP.APACHE.ORG. Who We Are. https://hadoop.apache.org/who.html#Hadoop+Committers, 2014.

[54] HARRISON, G. Hadoop's Next-Generation YARN. *Database Trends and Applications 26*, 4 (Dec. 2012), 39.

[55] HICKEY, R. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages* (2008), ACM, p. 1.

[56] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference* (2010), vol. 8, p. 9.

[57] HUNTER, T., MOLDOVAN, T., ZAHARIA, M., MERZGUI, S., MA, J., FRANKLIN, M. J., ABBEEL, P., AND BAYEN, A. M. Scaling the Mobile Millennium system in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 28.

[58] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 59–72.

[59] JONES, M. T. Process real-time big data with Twitter Storm. *IBM Technical Library* (2013).

[60] JOSEPH, D. Using Spark and Hive to process BigData at Conviva. http://www.conviva.com/using-spark-and-hive-to-process-bigdata-at-conviva/, 2011.

[61] KAMBURUGAMUVE, S., FOX, G., LEAKE, D., AND QIU, J. Survey of Distributed Stream Processing for Large stream sources.

[62] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (2011).

[63] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., AND TANEJA, S. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 239–250.

[64] KUMAR, R., GUPTA, N., CHARU, S., AND JANGIR, S. K. Architectural Paradigms of Big Data. In *National Conference on Innovation in Wireless Communication and Networking Technology–2014, Association with THE INSTITUTION OF ENGINEERS (INDIA)* (2014).

[65] LÄMMEL, R. Google's MapReduce programming model—Revisited. *Science of computer programming 70*, 1 (2008), 1–30.

[66] LIEVESLEY, D. Increasing the value of data. *BLRD REPORTS 6122* (1993), 205–205.

[67] LIU, X., IFTIKHAR, N., AND XIE, X. Survey of Real-time Processing Systems for Big Data. In *Proceedings of the 18th International Database Engineering & Applications Symposium* (New York, NY, USA, 2014), IDEAS '14, ACM, pp. 356–361.

[68] MARZ, N. *Big data : principles and best practices of scalable realtime data systems*. O'Reilly Media, [S.l.], 2013.

[69] MAYER-SCHÖNBERGER, V., AND CUKIER, K. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. An Eamon Dolan book. Houghton Mifflin Harcourt, 2013.

[70] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on Fast-forward. *ACM Queue 7*, 7 (2009), 10.

[71] MURTHY, A., VAVILAPALLI, V. K., EADLINE, D., MARKHAM, J., AND NIEMIEC, J. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[72] NEUMEYER, L., ROBBINS, B., NAIR, A., AND KESARI, A. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on* (2010), IEEE, pp. 170–177.

[73] OLSTON, C., CHIOU, G., CHITNIS, L., LIU, F., HAN, Y., LARSSON, M., NEUMANN, A., RAO, V. B., SANKARASUBRAMANIAN, V., SETH, S., ET AL. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 1081–1090.

[74] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[75] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (2009), ACM, pp. 165–178.

[76] RITTERMAN, J., OSBORNE, M., AND KLEIN, E. Using prediction markets and Twitter to predict a swine flu pandemic. In *1st international workshop on mining social media* (2009), vol. 9.

[77] SAMZA.INCUBATOR.APACHE.ORG. Samza. https://samza.incubator.apache.org/, 2014.

[78] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2010), pp. 1–10.

[79] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. *Facebook White Paper 5*, 8 (2007).

[80] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAM, J., PAN, X., GONZALEZ, J., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. MLI: An API for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on* (2013), IEEE, pp. 1187–1192.

[81] ST AMANT, K., AND ULIJN, J. M. Examining the Information Economy: Exploring the Overlap between Professional Communication Activities and Information-Management Practices. *Professional Communication, IEEE Transactions on 52*, 3 (2009), 225–228.

[82] STEWART, R. J., TRINDER, P. W., AND LOIDL, H.-W. Comparing high level mapreduce query languages. In *Advanced Parallel Processing Technologies*. Springer, 2011, pp. 58–72.

[83] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record 34*, 4 (2005), 42–47.

[84] STORM.APACHE.ORG. Storm, distributed and fault-tolerant realtime computation. https://storm.apache.org/, 2014.

[85] SU, X., SWART, G., GOETZ, B., OLIVER, B., AND SANDOZ, P. Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing. *Proceedings of the VLDB Endowment 7*, 13 (2014).

[86] TAYLOR, R. C. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC bioinformatics 11*, Suppl 12 (2010), S1.

[87] TEZ.APACHE.ORG. Apache Tez – Welcome to Apache Tez. http://tez.apache.org, 2014.

[88] THOMAS, S., HARDIE, G., AND THOMPSON, C. Taking the guesswork out of speed restriction. In *CORE 2012: Global Perspectives; Conference on railway engineering, 10-12 September 2012, Brisbane, Australia* (2012), Engineers Australia, p. 707.

[89] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1626–1629.

[90] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTONY, S., LIU, H., AND MURTHY, R. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 996–1005.

[91] THUSOO, A., SHAO, Z., ANTHONY, S., BORTHAKUR, D., JAIN, N., SEN SARMA, J., MURTHY, R., AND LIU, H. Data warehousing and analytics infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 1013–1020.

[92] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., BHAGAT, N., MITTAL, S., AND RYABOY, D. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, p. 147–156.

[93] TRAFFIC.BERKELEY.EDU. History of the Project | Mobile Millennium. http://traffic.berkeley.edu/project, 2014.

[94] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[95] WALKER, J. Streaming IN Hadoop: Yahoo! release Storm-YARN - Hortonworks. http://hortonworks.com/blog/streaming-in-hadoop-yahoo-release-storm-yarn/, 2013.

[96] WIKI.APACHE.ORG. PoweredBy - Hadoop Wiki. https://wiki.apache.org/hadoop/PoweredBy, 2014.

[97] XIN, R. S., GONZALEZ, J. E., FRANKLIN, M. J., AND STOICA, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[98] XIN, R. S., ROSEN, J., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data* (2013), ACM, pp. 13–24.

[99] YANG, F., MERLINO, G., AND LÉAUTÉ, X. The RADStack: Open Source Lambda Architecture for Interactive Analytics.

[100] YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), ACM, pp. 1029–1040.

[101] YANG, W., LIU, X., ZHANG, L., AND YANG, L. T. Big Data Real-Time Processing Based on Storm. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on* (2013), IEEE, pp. 1784–1787.

[102] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Fast and interactive analytics over Hadoop data with spark. USENIX.

[103] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

[104] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: A fault-tolerant model for scalable stream processing. Tech. rep., DTIC Document, 2012.

[105] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 423–438.

Appendix A

Samza Pipeline Prototype Implementation

Listing A.1 edu.monash.honours.system.SpeedConsumer (Java)

```
package edu.monash.honours.system;

import edu.monash.honours.util.SpeedReadingPair;
import org.apache.samza.Partition;
import org.apache.samza.system.IncomingMessageEnvelope;
import org.apache.samza.system.SystemStreamPartition;
import org.apache.samza.util.BlockingEnvelopeMap;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;

public class SpeedConsumer extends BlockingEnvelopeMap
{
  private final ServerSocket serverSocket;
  private final SystemStreamPartition systemStreamPartition;

  public SpeedConsumer(final String systemName, final String streamName, final int
      listeningPort) throws IOException
  {
    this.systemStreamPartition = new SystemStreamPartition(systemName, streamName, new
        Partition((0)));
    this.serverSocket = new ServerSocket(listeningPort);
  }
```

```
@Override
public void start()
{
  new Thread(
    () -> {
      try {
        SpeedConsumer.this.listenOnPort();
      } catch (InterruptedException e) {
        SpeedConsumer.this.stop();
      }
    }
  );
}

private void listenOnPort() throws InterruptedException
{
  InputStream inputStream;

  try {
    while ((inputStream = serverSocket.accept().getInputStream()) != null) {
      BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(
          inputStream));
      double speedReading = Double.valueOf(bufferedReader.readLine());

      put(systemStreamPartition,
          new IncomingMessageEnvelope(systemStreamPartition, null, null, new
              SpeedReadingPair(speedReading, false)));

      bufferedReader.close();
    }
  } catch (IOException e) {
    put(systemStreamPartition, new IncomingMessageEnvelope(systemStreamPartition, null,
        null,
                                                           "ERROR:_Cannot_read_from_port"
                                                                + e.getMessage()));
  }
}
```

```
  @Override
  public void stop()
  {
    try {
      this.serverSocket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @Override
  public void register(final SystemStreamPartition systemStreamPartition, final String
      startingOffset)
  {
    super.register(systemStreamPartition, startingOffset);
  }
}
```

Listing A.2 edu.monash.honours.system.SpeedSystemFactory (Java)

```
package edu.monash.honours.system;

import org.apache.samza.SamzaException;
import org.apache.samza.config.Config;
import org.apache.samza.metrics.MetricsRegistry;
import org.apache.samza.system.SystemAdmin;
import org.apache.samza.system.SystemConsumer;
import org.apache.samza.system.SystemFactory;
import org.apache.samza.system.SystemProducer;
import org.apache.samza.util.SinglePartitionWithoutOffsetsSystemAdmin;

import java.io.IOException;

public class SpeedSystemFactory implements SystemFactory
{
  private static final String OUTPUT_STREAM_NAME = "speeds";

  @Override
  public SystemConsumer getConsumer(final String systemName, final Config config, final
      MetricsRegistry metricsRegistry)
  {
    int listeningPort = Integer.getInteger(config.get("systems." + systemName + ".
        listeningPort"));

    try {
      return new SpeedConsumer(systemName, OUTPUT_STREAM_NAME, listeningPort);
    } catch (IOException e) {
      e.printStackTrace();
      return null;
    }
  }

  @Override
  public SystemProducer getProducer(final String s, final Config config, final
      MetricsRegistry metricsRegistry)
  {
    throw new SamzaException("You_cannot_produce_to_this_feed!");
  }
```

```
  @Override
  public SystemAdmin getAdmin(final String s, final Config config)
  {
    return new SinglePartitionWithoutOffsetsSystemAdmin();
  }
}
```

Listing A.3 edu.monash.honours.task.SpeedCheckStreamTask (Java)

```java
package edu.monash.honours.task;

import edu.monash.honours.util.SpeedReadingPair;
import org.apache.samza.system.IncomingMessageEnvelope;
import org.apache.samza.system.OutgoingMessageEnvelope;
import org.apache.samza.system.SystemStream;
import org.apache.samza.task.MessageCollector;
import org.apache.samza.task.StreamTask;
import org.apache.samza.task.TaskCoordinator;

import java.util.Map;

public class SpeedCheckStreamTask implements StreamTask
{
  // Constants
  private static final SystemStream FLAGGED_OUTPUT     = new SystemStream("kafka", "flagged
      -output");
  private static final SystemStream SPEED_CHECK_OUTPUT = new SystemStream("kafka", "speed-
      check-output");
  private static final double       SPEED_UPPER_LIMIT  = 90;
  private static final double       SPEED_LOWER_LIMIT  = 0;

  private static int totalMessageCount = 0;
  private int messageCount;

  public SpeedCheckStreamTask()
  {
    this.messageCount = 0;
  }
```

```
  /**
   * Listens on a stream providing speed readings, and processes those speed readings
       depending on whether or not they
   * exceed a specified threshold.
   *
   * Forwards on speed value to flagged stream if that value exceeds thresholds, else
       forwards on speed for further
   * processing.
   */
  public void process(final IncomingMessageEnvelope incomingMessageEnvelope, final
      MessageCollector messageCollector,
                      final TaskCoordinator taskCoordinator) throws Exception
  {
    SpeedReadingPair speedReadingPair = (SpeedReadingPair) incomingMessageEnvelope.
        getMessage();
    this.messageCount++;  // Keep count of number of messages that come through here
    totalMessageCount += this.messageCount; // Keep count of total number of messages
        received on this stream.


    if (speedReadingPair.getSpeed() < SPEED_LOWER_LIMIT || speedReadingPair.getSpeed() >
        SPEED_UPPER_LIMIT) {
      // Flag speeds that exceed thresholds
      speedReadingPair.flag();
      messageCollector.send(new OutgoingMessageEnvelope(FLAGGED_OUTPUT, speedReadingPair));
    }

    // Send speed on for further processing
    messageCollector.send(new OutgoingMessageEnvelope(SPEED_CHECK_OUTPUT, speedReadingPair)
        );
  }
}
```

Listing A.4 edu.monash.honours.util.SpeedReadingPair (Java)

```java
package edu.monash.honours.util;

import java.util.AbstractMap;
import java.util.HashMap;
import java.util.Map;

public class SpeedReadingPair
{
  private final double key;
  private boolean value;

  public SpeedReadingPair(final double key, final boolean value)
  {
    this.key = key;
    this.value = value;
  }

  public SpeedReadingPair(Map<String, Object> jsonObject)
  {
    this((Double) jsonObject.get("speed"), (Boolean) jsonObject.get("flag"));
  }

  @Override
  public String toString()
  {
    return "{" + this.key + ":" + this.value + "}";
  }

  public double getSpeed()
  {
    return this.key;
  }

  public boolean flagged()
  {
    return this.value;
  }

  public void flag()
  {
    this.value = true;
  }
```

```
  public static Map<String, Object> toMap(SpeedReadingPair pair)
  {
    Map<String, Object> jsonObject = new HashMap<String, Object>();

    jsonObject.put("speed", pair.getSpeed());
    jsonObject.put("flag", pair.flagged());

    return jsonObject;
  }

  public static String toJson(SpeedReadingPair pair)
  {
    return pair.toString();
  }
}
```

Listing A.5 SpeedCheckStreamTask (Samza config)

```
# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=speed-check


# YARN
yarn.package.path=file://${basedir}/target/${project.artifactId}-${pom.version}-dist.tar.gz


# Task
task.class=edu.monash.honours.task.SpeedCheckStreamTask
task.inputs=speed-input


# Serializers
serializers.registry.json.class=org.apache.samza.serializers.JsonSerdeFactory


# speed-input System
systems.speed-input.samza.factory=edu.monash.honours.system.SpeedSystemFactory
systems.speed-input.listeningPort=9999


# Kafka System
systems.kafka.samza.factory=org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.producer.bootstrap.servers=localhost:9092
```

Appendix B

Storm Pipeline Prototype Implementation

Listing B.1 edu.monash.honours.bolt.SpeedCheckBolt (Java)

```
package edu.monash.honours.bolt;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.util.Map;

public class SpeedCheckBolt extends BaseRichBolt
{
  private final static int SPEED_UPPER_BOUND = 90;
  private final static int SPEED_LOWER_LIMIT = 0;

  private static int      totalTupleCount = 0;

  private int             tupleCount;
  private OutputCollector collector;
```

```
/**
 * Runs once when this bolt is created.
 */
public void prepare(final Map map, final TopologyContext topologyContext, final
    OutputCollector outputCollector)
{
  this.collector = outputCollector;
  this.tupleCount = 0;
}

/**
 * Runs every time a tuple is received.
 *
 * @param tuple The input tuple containing a single speed field.
 */
public void execute(final Tuple tuple)
{
  updateTupleCounts();

  Object receivedValue = tuple.getValue(0);

  Values emitTuple;
  if (receivedValue instanceof Double) {
    // If received value is a double, check the speed received
    emitTuple = checkSpeed((Double) receivedValue);
  } else {
    // else, ignore processing and send the message onwards
    emitTuple = new Values(receivedValue, true);
  }

  this.collector.emit(tuple, emitTuple);
  this.collector.ack(tuple);
}

/**
 * Declare the output fields as being speed and a noise flag.
 */
public void declareOutputFields(final OutputFieldsDeclarer outputFieldsDeclarer)
{
  outputFieldsDeclarer.declare(new Fields("speed", "noiseFlag"));
}
```

```
  /**
   * Keeps count of tuples that pass through this bolt.
   */
  private void updateTupleCounts()
  {
    this.tupleCount++;
    totalTupleCount += this.tupleCount;
  }

  /**
   * Checks the given speed to see if it within limits.
   *
   * @param speed Speed to check.
   * @return A Values tuple with two fields: speed and a noise flag. Noise flag is checked
       if speed exceeds limits.
   */
  private Values checkSpeed(double speed)
  {
    if (speed > SPEED_UPPER_BOUND || speed < SPEED_LOWER_LIMIT) {
      return new Values(speed, true);
    }

    return new Values(speed, false);
  }
}
```

Listing B.2 edu.monash.honours.spout.SpeedOutputSpout (Java)

```
package edu.monash.honours.spout;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.util.Map;

public class SpeedOutputSpout extends BaseRichSpout
{
  /**
   * The key in the storm config under which the listening port is declared
   */
  private static final String LISTENING_PORT_CONFIG_KEY = "spout.listeningPort";

  private SpoutOutputCollector collector;
  private ServerSocket         serverSocket;

  /**
   * Runs once when this spout is created.
   *
   * @param conf Storm topology configuration map.
   * @param topologyContext
   * @param spoutOutputCollector
   */
  public void open(final Map conf, final TopologyContext topologyContext,
                   final SpoutOutputCollector spoutOutputCollector)
  {
    this.collector = spoutOutputCollector;

    try {
      this.serverSocket = new ServerSocket(Integer.valueOf((String) conf.get(
          LISTENING_PORT_CONFIG_KEY)));
    } catch (IOException e) {
      this.collector.emit(new Values("ERROR: Cannot open port - \n" + e.getMessage()));
```

```
    }
  }

  /**
   * Run every time a new value is sent to this spout.
   *
   * Emits a tuple containing a single speed value for further processing.
   */
  public void nextTuple()
  {
    try {
      BufferedReader bufferedReader = new BufferedReader(
        (new InputStreamReader(serverSocket.accept().getInputStream())));
      double speedReading = Double.valueOf(bufferedReader.readLine());

      this.collector.emit(new Values(speedReading));
    } catch (IOException e) {
      this.collector.emit(new Values("ERROR: Cannot read from port - \n" + e.getMessage()));
    }
  }

  /**
   * Declare the output field as being a single speed value.
   */
  public void declareOutputFields(final OutputFieldsDeclarer outputFieldsDeclarer)
  {
    outputFieldsDeclarer.declare(new Fields("speed"));
  }
}
```

Listing B.3 edu.monash.honours.SpeedCheckTopology (Java)

```java
package edu.monash.honours;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.utils.Utils;
import edu.monash.honours.bolt.SpeedCheckBolt;
import edu.monash.honours.spout.SpeedOutputSpout;

public class SpeedCheckTopology
{
  /**
   * The listening port upon which the spout will get data.
   */
  public static final String LISTENING_PORT = "8888";

  public static void main(String[] args) throws Exception
  {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("speed", new SpeedOutputSpout());
    builder.setBolt("speed-check", new SpeedCheckBolt()).shuffleGrouping("speed");

    Config conf = new Config();
    conf.put("spout.listeningPort", LISTENING_PORT);
    conf.setDebug(true);

    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("speed-checker", conf, builder.createTopology());
    Utils.sleep(100000);
    cluster.killTopology("speed-checker");
    cluster.shutdown();
  }
}
```

Appendix C

Spark Streaming Pipeline Prototype Implementations

Listing C.1 edu.monash.honours.SpeedCheck (Scala)

```
package edu.monash.honours

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{StreamingContext, Seconds}

object SpeedCheck
{
  val SPEED_UPPER_LIMIT = 85
  val SPEED_LOWER_LIMIT = 0

  def main(args: Array[String])
  {
    val sparkConf = new SparkConf().setMaster("local[2]").setAppName("SpeedCheck")
    val ssc = new StreamingContext(sparkConf, Seconds(1))
    val speedStream = ssc.socketTextStream("localhost", 9999)

    speedStream.map(checkSpeed)

    speedStream.print()

    ssc.start()
    ssc.awaitTermination()
  }

  def checkSpeed(speedValue: String): (String, Boolean) = {
```

```
    if (speedValue.asInstanceOf[Double] > SPEED_UPPER_LIMIT || speedValue.asInstanceOf[
        Double] < SPEED_LOWER_LIMIT) {
      (speedValue, false)
    } else {
      (speedValue, true)
    }
  }
}
```

Listing C.2 edu.monash.honours.SpeedCheck (Python)

```
__author__ = 'poltak'

from pyspark import SparkContext
from pyspark.streaming import StreamingContext

SPEED_UPPER_LIMIT = 85
SPEED_LOWER_LIMIT = 0
SOCKET_ADDRESS = "localhost"
SOCKET_PORT = 9999


def init_spark(name):
    sc = SparkContext("local[2]", name)
    return StreamingContext(sc, 2)


def close_spark(ssc):
    ssc.start()
    ssc.awaitTermination()


def check_speed(speed):
    try:
        if float(speed) > SPEED_UPPER_LIMIT or float(speed) < SPEED_LOWER_LIMIT:
            return speed, False
        else:
            return speed, True
    except ValueError:
        return speed, False


def main():
    ssc = init_spark("pyspark-speed-check")
    speed_stream = ssc.socketTextStream(SOCKET_ADDRESS, SOCKET_PORT)

    speed_stream = speed_stream.map(check_speed)

    speed_stream.pprint()

    close_spark(ssc)
```

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