Computational Shedding in Stream Computing

David Guerin

A dissertation submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

July 2019

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

David Guerin

Dated: July 25, 2019

Permission to Lend and/or Copy

Acknowledgements

A PhD although a journey made in solitude cannot be completed without the help of many great people who contributed directly and indirectly along the way. This thesis is no different.

I'd like to thank my supervisor Stephen, for all the helpful insight and advice you have given throughout the years. Most notably thanks for giving me the room to make this work my own and at times believing in this work, when I didn't. I'd also like to thank Stefan for all the advice and help he had given over the years when you dropped by the office for a chat. To the guys in 008 particularly Brendan and Michael, thanks for all the discussions and especially the arguments along the way. Having colleagues like this, only leads to better research. I'd also like to thank past and present DSGers in general, you all have made an impact on this work and me over time, from the brief and sometimes long conversations I've had with you.

I would also like to thank both my mother, Catherine, and father, Noel for all the help, encouragement and support you have given me throughout the years. This could not have happened without you. To my wife, Karen, the support you have given has been remarkable over the last few years. You have encouraged me to finish this thesis. For this and so much more I'll always be grateful.

And finally to Jake and Evie, thanks for letting me sleep through the night to finish this thesis off.

David Guerin
University of Dublin, Trinity College
July 2019

Abstract

Stream Computing, a generic on-line data processing paradigm has emerged as the preferred approach in the processing of continuous data streams. Data streams suffer from a bursty characteristic where the data rate of the stream can spike temporarily up to orders of magnitude greater than normal expected levels. As such, producing timely application results is difficult as queues fill. The classic response to these temporary scenarios is to shed input data. However, Load Shedding (LS) impacts negatively on application output accuracy as relevant data is discarded before it is processed. Further, LS rates tend to be proportional to the input rate of the stream, as such high data rates can lead to high data loss during overload events.

For many classes of applications, this can have a particularly negative impact on the quality of the output result, given that data is simply not processed before it is shed.

This thesis presents a new approach, Computational Shedding (CS), to the problem of maintaining application result accuracy while attempting to forgo input data loss during transient busty data events.

Rather shedding input data within the stream, we propose to adapt the application and shed tasks or *subtasks* of the executing application temporarily, to reduce message process costs in the stream. As such, this mechanism provides for an opportunity to temporarily increase processing capacity thereby forgoing the need for deliberate data loss during a bursty data event.

We have evaluated this approach against traditional LS techniques in a number of ways, such as in terms of output application accuracy and application processing duration. In experimentation, we have found in applicable applications, subtasks can be discarded for a time. Remaining subtasks can continue to produce a valid imprecise result. CS was compared to LS alternatives which simply do not process any discarded data. We show that through the results of our evaluation, CS leads to more timely and accurate results when compared to LS alternatives.

Contents

List of Tables

List of Figures

Glossary of Terms

- **Computational Shedding (CS)** - A bursty data stream management strategy to controllably discard optional subtasks of an application such the mandatory subtasks remain to produce an approximate application result during application overload.
- **Computational Shedding - Accuracy Focused (CS-A)** - is a computational shedding strategy focused on selecting subtasks in the computation with a primary focus on accuracy.
- **Computational Shedding - Binary Search (CS-B)** - is a computational shedding strategy focused on reducing the selection time in SEP lookups using a Binary Search Algorithm implementation.
- **Computational Shedding - Cost Focused (CS-C)** - is a computational shedding strategy improving upon CS-D by reducing the overload detection cost during bursty data stream events.
- **Computational Shedding - Duration Focused (CS-D)** - is a computational shedding strategy simply focused on producing results for the application as quickly as possible, with application accuracy as a second objective.
- **Directed Acylic Graph (DAG)** - a stream computing application structure, where applications are structured as a set of sequentially connected operators, User Defined Functions (UDF) or tasks.
- **Imprecise Computation (IC)** - an area in embedded devices where workloads are structured in such a fashion to complete on time by turning off optional jobs, should job scheduling indicate output results will be late.

- **Load Classifier (LC)** - A bursty data stream management component with the purpose of classifying current system load and selecting an appropriate entry in the SEP.
- **Load Classifier - Duration Focused (LC-D)** - a load classifier which was implemented based on the designs of CS-D.
- **Load Handler (LH)** - A class of components with the sole purpose of determining application overload and executing a strategy to ensure application result deadlines are honoured..
- **Load Shedding (LS)** - A bursty data stream management strategy to controllably discard input messages during application overload, to enforce application result deadlines.
- **Processing Element (PE)** - a logical association of software components. In this thesis a PE contains an input stream handler, output stream handler and any application state. This is also the location in the SSCF framework where subtasks execute..
- **Random Load Shedding (LS-R)** - A bursty data stream management strategy to randomly discard input messages during application overload.
- **Semantic Load Shedding (LS-S)** - A bursty data stream management strategy to semantically select input messages to discard during application overload based on utility value to the application..
- **Simple Stream Computing Framework (SSCF)** - a C# Dot Net Core based stream computing framework design to handling bursty data streams in it's core. Stream Computing Applications can be built on this framework using the Computational Shedding approach to handle bursty data streams.
- **Spearman's Footrule (SF)** - A aggregate measure of the displacement of items between two lists by calculating the difference in position between one list to the other.
- **Stream Processing Engine (SPE)** - a new class of software framework with aids in the development and executions of stream computing applications.
- **Subtask Execution Plan (SEP)** - is a pre-computed subtask combination plan of all optional and mandatory subtasks within the application. At runtime during overload the load classifiers select an appropriate entry in the plan fitting for the current system over utilisation. The subtasks selected then process input messages.

Chapter 1

Introduction

1.1 Overview

As more and more applications move to a stream computing model, one in which input data contained in a stream is pushed into an application, the applications must deal with infrequent but intense bursty stream events. Bursty data streams induce system overload by delivering more messages or events in a data stream than provisioned resources in the application can process within required application deadlines. Examples of such applications include, real-time web search and analytics [94], where users search for answers to queries quickly and their interactions are monitored and measured. Further, in health care where patients are monitored in real-time [12] and events are fired when something unexpected occurs in the patient. Also, in road network traffic management as describe in Smart Cities [77] to monitor and guide drivers through complex road networks include through crash events.

As such a new class of data processing system has emerged in response to the applications. The focus of Stream Computing is to provide generalised computing platforms where application logic can reside, processing messages and events as the stream is consumed, quickly [2, 10, 94, 118]. However, without adaptation in these systems to match bursty input workloads to available resources, application results may be delivered late. Subsequently, if no action is taken, input data can be lost as applications buffers fill.

We propose an alternative approach to existing works, which we call CS. Rather than discarding input messages during immediate bursts as current works suggest, we reduce the cost of processing messages in the data stream. This is achieved by temporarily disabling tasks, or subtasks of applicable application, such that each message in the stream now has a lower processing cost. By disabling more and more subtasks, when available, the processing cost of each message in the stream can be further reduced allowing the application to increase ingestion rates, attempting to match input data rates with available computational resources. This has the consequence of reduced output accuracy as subtasks are temporarily disabled but helps ensure messages are not discarded intentionally or uncontrollably as input rates rise.

In this chapter, we first describe the motivation for this work including example applications, and outline our approach to the bursty data stream management problem. We outline the objectives of this research, the scope of the work as well as the contributions made.

Finally, we conclude the chapter with a brief outline of the thesis.

1.2 The Overload Challenge

Stream computing, a generalised stream processing model to consume on-line data, has matured into a stable platform for the processing of continuous data streams over the last ten years. The use of stream computing platforms as both a solution to time bounded data and processing large quantities of data, has led to many entities adapting it in lieu of batch processing models used in the past [5, 7, 22, 54, 79, 94]. Unlike batch computing where data is static and resides in a central repository, stream computing must process data on the fly to meet application latency requirements. Streams of data are generated from sources and pushed into stream computing applications as soon as possible. This push based nature sets stream computing apart from the pull model in batch computing, where an application acquires data at a rate it can control and tolerate [111].

However, the push based nature of the input data in the stream can lead to bursty data characteristics, where unexpectedly, the input data rate grows past the processing capability of the stream application and it's resources. A stream application must process pushed input data as fast as it is delivered. If not, this can lead to application result latency violations, or worse lead to unintentional data loss as input buffers overflow with data. Under normal operating conditions, an appropriate amount of processing resources can be provisioned, capable of filling this requirement, but during bursty loads, the volume of input data cannot be anticipated and appropriate resources cannot be provisioned in advance [15].

This problem can be expressed as a Resource Utilisation Problem, where the available CPU resources are not enough to process the expected incoming workload, given it has increased. There are two solutions to this, one is to increase CPU resources such that it can match the required workload in the stream. The second is to reduce the cost of processing the workload such that the available CPU resources are enough to process the workload with it's increased message volume. These two approaches can be characterised by two main approaches to this problem in the literature.

Current approaches to solving bursty data streams reside in two categories. First, data based approaches where input data is dropped or intentionality shed to ensure output result deadlines are met, this is called Load Shedding (LS). However this is at the expense of application accuracy[2, 28]. The other category, consists of scaling the application at runtime with additional local or remote resources attempting to match the input workload processing requirements. However, this assumes resources can be acquired quickly enough and in sufficient quantity [85, 104] which is difficult given bursty data streams tend to evolve, increasing or decreasing in velocity and volume unpredictably.

As such this problem requires a new focus. This thesis investigates a new approach to this problem using a partial computation processing model. By reducing the computational cost of processing each input message at runtime, a system with a finite set of resources can increase its processing rate or ingestion rate to match higher input data rates during bursty data periods, matching the data stream as it evolves during short-lived bursts.

1.2.1 Example Applications

To help motivate the design of this proposed approach, in this section we suggest applications to which our proposed CS approach is applicable during bursty data events. It is assumed that an application for which CS is applicable is constructed or can be constructed in such a fashion as to have individual tasks or subtasks within it which implement the core business logic of the application. This is an important requirement and as such an application is required to have sufficient complexity for CS to be applicable. We discuss this further in chapter 3. Additionally it is also required that the removal of a subtask or multiple subtasks will not prevent the application from generating a result. As such the removal of one or more subtasks should reduce the cost of processing a message in the data stream while also ensuring that message ingestion rates increase (the rate at which a message is processed including queueing time).

1.2.1.1 Real-time Search and Indexing

Web search applications provide search results to an input search query when entered into a search engine. Behind this request, a search engine includes a number of services which index, store and retrieve web page information based on the entered search query. The index used is a large body of indexed information held in memory, such that when a new search query arrives in the search engine, the index information can be accessed quickly and results can be returned in a ranked order, an order determined by numerous metrics used to index the information. This solution requires the large body of index web pages to change infrequently, given it takes considerable time to index [95]. However, should the web pages change frequently, the search index would not have the latest changes and thus may provide stale or inaccurate search results when queried, until it is updated. In applications where their sources of information change frequently, a new search model is needed.

Twitter, (http://www.twitter.com) is an American on-line news and social networking service where users post and interact with messages known as "Tweets" [57]. Tweets are causal ordered messages which are sent between users of the platform. Tweets contain a large variety of meta-data relating to the Tweet and the user that posted it, such as username, location, time of day, retweet count, liked count and 32 other properties [65]. As of Q1 2018, there are estimated to be approximately 335 million active monthly global users generating approximately 500 million Tweets each day, averaging 6000 Tweets per second[1]. In a classical search application it becomes incredibly difficult to search and distribute this data to users in the platform. As such Twitter has created a two stream computing platform to manage this large ever changing data set. Both are designed with stream computing principles in mind [110], which they call Storm[119] and Heron [80]. We discuss both in chapter 2.

However, the real-time distribution nature of Twitter has been very advantageous for distribution of live news events and emergency management. In 2009, Twitter was used as an early warning system for the *Red River Valley flood threat* along the border of North Dokata and Minnesota in the United States (U.S) [96]. In 2010, Twitter was first used an early warning system for Earthquakes in Japan [102]. In the U.K floods of 2013/2014, emergency responders used the

[1]https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/ (accessed, August 2018)

service to help guide search and rescue efforts[97].

There is little doubt that during times of need, the reliance on such a service being robust and able to handle the volume of data generated from users is important. During a data burst the amount of data generated can be vastly larger than normal levels. This is key to ensuring this data is processed in the system successfully as well as meeting the expectations of it's users. Further ensuring that even under the effects of a burst in the data streams from these sources, that all input messages are received and processed to some degree without being late or without being discarded.

CS has the opportunity in this case to ensure that during a bursty event in a Tweet data stream that the indexing services behind search functionality in the platform, continues to operate without causing delayed processing of input messages. This is especially the case at increased data input rates. Some metrics in the index can be temporarily discarded during overload. Such that indexing activities which comprise of indexing the 37 properties in each Tweet can be discarded gracefully (removing functionality while still producing an application result) and continue to produce results for users on time. Thus this provides for increased message throughput rates (Tweets in this case) as a result of reducing message processing costs. Thereby allowing messages to get indexed more quickly and allowing results to get where they need to as quickly as possible.

The alternative is to allow messages to start filling input queues which begin to slow down the distribution of much needed information to users of the service. If load shedding was applied to ensure message delivery times, this may lead to message shedding and data loss. The shed messages which may contain useful information to a news event or emergency management notification. Or shed messages from users, whom are simply communicating between one another during such an event. Individual user messages of this nature are difficult to classify with a priority over other messages in the platform, given messages of a heterogeneous nature.

As such CS has the potential here to provide for the best outcome in that all messages can be processed while also ensure the application can increase it's message ingestion rate, while maintaining a level of accuracy in the index and without the need to discard input messages.

1.2.1.2 Social Media Content Classification

The Sigma Service at Facebook [2] provides content classification services for various types of content on the Facebook platform [88]. Facebook (http://www.facebook.com) is the worlds largest social network, with 1.94 billion monthly active users, including close to 1.74 billion monthly active mobile users who interact with one another though messages, posts, and the uploading of digital content such as images and videos [3]. Sigma is designed to prevent the publishing of spam on the platform soon after it has been uploaded. Sigma clients ask high-level questions, communicating with central service much like a data stream, asking questions such as "Is this post spam?" or "Is this URL associated with malware?" using classification rules. The Sigma central service responds with answers such as "Yes, block this account" or "No, carry on".

Sigma is a rule based system built in a domain-specific language called FXL which connects these questions to a dozen or so distinct sources of facts necessary to answer spam related questions, as outlined above. Facts are pieces of information which can help answer these questions from information on hand in these sources. For example, recently tagged users or accounts which have been reported as posting spam to the platform. These classification rules can execute in hierarchical order to gain further clarity on individual pieces of content when preliminary questions have been answered but primarily each rule executes in isolation against it's respective data source.

One of the key problems with this service is balancing expressiveness and performance. Facts come from multiple diverse sources, so in order to run efficiently rules must be able to fetch facts quickly. As such they have chosen a parallel request solution to get the answers to these rules when applicable, such that multiple requests are sent to each queried service in parallel. However, FXL implementations suffer from IO scheduling issues and high CPU utilisation as a result.

This becomes particularly troublesome when there is a large influx of uploaded content centred around real world events. Given the event occurs without prior knowledge, the already burdened system struggles to keep up with classification requests. As was the case during the 2010 Haiti Earthquakes in the Caribbean [78, 123] or during Hurricane Sandy in 2012 [56]. Helpful users posted content aiding in the U.S. relieve efforts in both cases, aiding by uploading content to the platform of the effects of each disaster in different localities. The relief efforts then used this

[2]https://code.fb.com/core-data/open-sourcing-haxl-a-library-for-haskell/ (accessed, August 2018)
[3]https://www.statista.com/topics/751/facebook/ (accessed, August 2018)

information, determined the best locations to supply aid quickly. However, it become difficult to classify content quickly as valid content or spam, given the volume of which was uploaded in such a short space of time and as such delays arise.

CS can help here by ensuring that data is processed quickly during such bursty events. In this case, modifying the heavy classification workload by providing for a graceful degradation (removing parts of the application and still being able to produce a valid result) of the Sigma Service. Such that during the bursty event where users are uploading content or interacting with one another, classifications rules - each contained within a subtask, can be gradually removed to increase ingestion rates of the content into the platform while continuously to supply a partial classification result. As load increases further, more and more rules can be relaxed until just a set of base rules are left, a mandatory set. It is expected then after a period of time the burst has passed, that full application functionality can revert back to it original level with the full list of classifications rules. Alternatively, CS could provide time to provision additional computational resources such that the Sigma Service can scale up the new input data rate during the bursty event.

1.3 Approach

Current work in stream computing shows that overloaded data streams can have a drastic effect on the accuracy of a stream computation and cause intolerable delays in output results [28, 73]. As stream data rates increase, latency of the output result increases and input data is lost uncontrollably. In the case of load shedding data is lost intentionally sacrificing output result accuracy for output result latency. Scaling of the computation improves this but requires additional resources, in either the local or remote environment, when resources are available and can come on-line fast enough. The move to cloud computing has helped but many have assumed that the resources will be available when requested [54]. Others wish to preallocate additional resources that are not in use until an overload occurs however this becomes wasteful. In addition, the resources provisioned may not be enough to handle the new input data rates of the bursty data stream given that data streams can be ever evolving in duration and velocity.

In this thesis we propose an alternative perspective, where we shed computation rather than data. Given a streaming application consisting of a set of subtasks both mandatory and optional. At runtime during application overload, optional subtasks are discarded from the application temporarily reducing the computational cost of processing new input messages. As load increases further, additional optional subtasks can be disabled until just mandatory subtasks remain. We aim to implement this approach dynamically at runtime such that sub-selections of subtasks can be selected at a proportional rate to the measured overload amount. Even though selection is dynamic any additional activities in a stream computing application should be minimal and optimised such that resources are not taken from the functional activities of the application and used for non-functionality purposes.

As such, selected subtasks combinations are determined statically and off-line, creating a Subtask Execution Plan (SEP) such that each entry in the table provides for a different subset of optional subtasks as well as always selecting any mandatory subtasks present in the application. To optimise the selection for application requirements in one direction or another, the SEP entries in the table can be prioritised. We show this by presenting two ordering types in the application, first in order to priorities for message processing duration and another by selecting application accuracy first. As higher and higher data rates are received by an application, entries in the table are selected which gracefully degrade the number of subtasks and combination of subtasks in the application. Consequentially degrading the quality of the output result of the application. We show this in our evaluation. We further improve upon our original implementation by providing improvements in the selection process to further reduce the cost of SEP entry selection.

In addition to determining how this adaptation should take place, we must also determine when this adaptation should take place. Some literature works attempt to statically determine workload limits which would otherwise cause end-to-end message latency violations. This estimation process is computed off-line. This is to aid in the the reduction of non-functional processing costs on-line [116]. In CS a similar approach has been taken by determining the expected processing cost of each subtask combination in each row in the SEP. With these pre-computed estimate values new workloads can be matched with appropriate SEP entries by estimating the required processing cost. We further improve upon this selection process to reduce the cost of utilising CPU resources for non-functional purposes with multiple improved versions. Each version is outlined in chapter 3. Evaluation results are presented in chapter 5.

Finally, we measure the benefits and drawbacks of this approach against appropriate load shedding alternatives. This is evaluated through a proof of concept real-time search application which has been developed. This was done to study and analyse the effects each approach has on the application during bursty stream periods. We provide the results to a large scale quantitative analysis, which measures numerous factors, such as message processing costs, application durations and applications output accuracy. This evaluation is preformed as such to allow for comparisons between overload handling approaches within the complexities of a simulated real world application using a real world data set. The results of which are presented in this thesis.

1.4 Objectives

Certain stream applications as outlined in section 1.1 and section 1.2.1 cannot tolerate late application results and do not tolerate data loss well. In such applications, prior works as outlined in chapter 2 are not suitable in managing bursty data stream overload. Some of these works suggest providing additional resources. However the burst may end well before resources come on-line. Additionally bursts have the potential to evolve and increase in volume, such that the additional resources provisioned may not be enough to process this higher data rate without application result delay. Further, deliberate acts in inducing controlled data loss may not be appropriate, given that input messages in the stream cannot be aggregated when the data is of a complex nature or if the data is heterogeneous. For instance, a subsequent message received later cannot replace the previous discarded message when considering the context of the message, such as the text of multiple Twitter tweets from different users. Additionally applications as outlined in section 1.3, cannot handle induced data loss well, given that to do so may have devastating consequences in the applications outcome.

In this thesis, the central hypothesis will address the question:

1. Can an application adaptation approach during a bursty data event produce application results which are delivered on time ?

2. Further, when timely results are generated, can the results also be as accurate or more accurate than load shedding alternatives ?

3. Additionally, can this adaptation also occur without the need to induce data loss during intense but short-lived bursty events ?

We will compare existing appropriate load handling approaches, namely Random Load Shedding (LS-R) and Semantic Load Shedding (LS-S), to our CS designs and implementations, in order to demonstrate that an application adaptation at runtime is a viable alternative in the event of a bursty data stream. This can improve upon application result latencies and application result accuracy.

1.5 Scope

This thesis primarily describes a partial computation adaptation model which modifies an applications processing complexity with the aim of reducing message processing costs in a stream computing application temporarily. Additionally, this thesis is scoped to just CPU resources, that is not to say that memory or network bandwidth are not limiting factors in bursty data stream scenarios.

CS as it will be later described, requires a certain level of application complexity for the approach to be applicable. Further, the application must be constructed in such a manner to allow parts or subtasks of the applications business logic to be temporarily switched off. Such that if a component of the business logic was discarded a partial result is still possible.

Within this work we also scope the concept of bursty data streams of which there are many in the literature. We are primarily concerned with short-lived but intense bursty data streams, such that it would be a waste of resources to dynamically scale up an application. An example would be through the use of cloud computing resources. We assume that the burst has already dissipated by the time resources are available for use.

The performance of CS, through it's implemented versions and load shedding alternatives are measured through the analysis of a proof of concept Twitter search application. The data for this application has been extracted from Twitter and is replayed numerous times at increasing data rates to simulate data stream overload with a large variety data set. The analysis of the implemented approach is preformed quantitatively, observing the effects of a high data rate stream on the results of the application with each implemented approach applied. Further, we use a single data stream for analysis and clarity purposes, that is not to say the underlying framework cannot support more than one stream at a time. Further this work does not present a formal analysis of CS.

We assume with the use of CS which reduces processing complexity costs, that reclaimed processor cycles are used to further increase the message throughput rate of the application processing the bursty data stream. We assume that the reclaimed processor cycles are not used for non-functional work such as other activities outside the application. We also assume that the operating system executing the application will allocate the reclaimed time to the stream application to process messages at the higher rate, when the application makes system calls to the operating system for more CPU time.

Finally, it should be noted, that issues of node or container failure, trust and security are not considered in this thesis.

1.6 Contribution

This thesis makes a number of key contributions, as follows: we introduce a novel approach to bursty data stream management in stream computing, called Computational Shedding (CS) as an alternative to existing approaches. We demonstrate that by reducing the cost of processing messages in the application by discarding a portion of application functionality temporarily, message processing rates increase during the busty data events. During such times the application results which are produced are not late and are more accurate than other alternative approaches within the bounds of the enclosed experimentation. Further we aim to show graceful degradation, such that as subtasks within the application are disabled, an output result is produced albeit it with a lower level of quality.

This approach is shaped by three important principles:

First, when an overload in the application occurs, the detection of this overload or future overload, based on assumed future workloads, must be detected quickly to ensure applications adaptations happen quickly. Helping to ensure output results are delivered both on time and are accurate. In such a case the use of queues and a message-per-message load assessment approach provides for a means to detect and measure both current and future workloads. Additionally this approach provides for a granular means to detect load and overload.

Second, any practical load handling algorithm must be very light-weight, given the system and application are already under duress whenever overload intervention is applied. To minimise further overload, our approach relies on pre-computing and materialising a Subtask Execution Plans (SEP) in advance, determining which selections of subtasks to execute for a measured input load. SEPs are generated in two conflicting orders in this work; to optimise for reduced message processing latency and to optimise for application result accuracy. CS essentially trades application accuracy for lower latency result processing and thus higher message throughput.

As such it is important to attempt to control result accuracy loss and more appropriately that accuracy loss is kept to a minimum while also producing results on time. Thus SEP selection and ordering has been optimised to direct the application results in this way.

Third, we further minimise runtime-overload by providing two additional approaches to select SEP entries in a more optimal manner and thus reduce both computational resources for non-functional tasks. As well as reducing any further delays introduced by load handling activities.

Finally, we provide a novel distributed stream computing framework to the community called Simple Stream Computing Framework (SSCF) in which the CS technique was developed and evaluated. Load shedding implementations are also provided for evaluation purposes. This framework and the proof of concept application were evaluated in Microsoft's Azure Cloud Computing Platform which was used to provide a basis where the evaluation could take place at scale. In this evaluation a large proof of concept real time search application was developed to help capture the complexities and data variabilities of a stream computing application. In this context, load shedding and computational shedding strategies are evaluated within this application's remit.

As we will show, the performance of the work in this thesis has been extensively studied through analysis of the results produced by this application. Within this we should that when the SEP is prioritise, it can optimise for both latency and application result accuracy. In our experimentation is produced results which were more accurate and timely than load shedding alternatives.

1.7 Thesis Roadmap

The remainder of this thesis is organised as follows:

Chapter 2 Related Work presents further background information regarding bursty data streams and provides a review of existing approaches and existing works to manage bursty data streams in stream computing domains. The perspective taken in this thesis focuses on data discarding techniques and approaches which scale the message ingestion rate of a stream application at runtime.

Chapter 3, CS: Computational Shedding, describes our proposed CS approach, including the rational for the approach, expectations, assumptions, constraints and our design decisions, as well as outlining how this approach was applied to a stream computing framework which was developed as part of this work. CS was developed with a number of different versions each of which has a different improvement, the positive and negatives of each are outlined in this chapter.

Chapter 4 Computational Shedding (CS) and The Simple Stream Computing Framework (SSCF) outlines the implementations of the approach as well as the SSCF a novel distributed stream computing framework developed within this thesis. CS has been implemented in SSCF and the framework priorities load handling strategies within it's architecture. The chapter elaborates on the motivations for it's creation and further outlines the major components within the framework. The chapter also outlines the proof of concept application which was developed on top of SSCF to simulate the complexities and variances in a real world application. The application ranks and searches a simulate live Twitter data stream based on a users input search query, presenting results soon after. The application has been developed with CS to facilitate overload data stream management.

Chapter 5 Evaluation outlines the evaluation methodology and the findings from the experimentation carried out to compare each version of CS implemented as well as evaluating two load shedding strategies against CS. The comparison is presented within the scope of the proof of concept Twitter search application outlined in chapter 4. The aim of the evaluation is to determine if both message throughput rates and application output accuracy could be improved during a bursty data stream using a CS approach when comparing against LS alternatives. The chapter ends with conclusions drawn from the results of this experimentation.

Chapter 6, Conclusion and Future Work summaries the thesis and its achievements. As well as outlining the major findings of the thesis and drawing conclusions from the work. Finally the chapter ends with recommendations for future work building on top of the results and open questions that remain in CS.

1.8 Chapter Summary

In this chapter, we outline the main problem this thesis addresses and the motivation for doing so in section 1.2. Section 1.3 outlines the approach taken by attempting to reduce the demands on CPUs during overload events. The limits of this work and example applications as to where this approach can be applied, are presented also.

Section 1.4 outlines the objectives of this thesis and the research questions we wish to find answers to. Section 1.5 scopes this work to CPU bounded overload situations as well as to high intensity but short-lived bursty data streams. Section 1.6 outlines the contribution of this thesis, with the development of a novel Computational Shedding approach which reduces the computational complexity of each processed message during bursty data stream periods and thus allows for increased message throughput within a stream application. Additionally we outlined the stream computing framework developed as part of this work.

Finally section 1.7 outlines the chapters in this thesis as well as giving a brief description of their contents.

Chapter 2

Related Work

2.1 Overview

In this chapter, we present the background to overload handling strategies in stream computing and the current state of the art in the field.

We begin by describing this basic problem in stream computing applications and why this problem needs to be solved in section 2.2.

We then move on to describing the related work in this area to solve the bursty data stream problem. This is divided into three main categories within three sections. Each category includes individual techniques described within each section. First section 2.4 describes data based approaches. Which during system overload attempts to modify the input data stream to reduce input data volumes, thus reducing input workloads. Next, section 2.5 addresses task based approaches, where research has been directed into task migration and scaling up application resources at runtime. This approach uses additional threads, additional local and remote resources whilst also using cloud computing as well as containerisation resources in an elastic manner. Finally, in section 2.7 we discuss emergent task modification techniques. Which aim to provide an adaptation within the application at runtime to reduce the cost in processing individual messages and thus allow the application to increase message processing rates with the same fixed computational resources.

Each section outlines its contained techniques as well as describing the individual systems which implement such techniques, accompanied with an analysis of the important parts of each system. The focus of this review is to investigate related works which handle bursty data streams in a timely manner as well as attempting to keep application output result accuracy high during such an event.

We begin this chapter with background information.

2.2 Background

This section outlines the background information to contextualise the subject of stream computing, providing definitions of a data stream and outlining the bursty data stream problem.

2.2.1 Stream Computing

Stream computing has emerged in the last number of years as a new processing paradigm to processing large and continuous streams of data. The definition of a data stream varies across domains, but in general, it is commonly regarded as input data that arrives at a high rate, often being considered as a big data problem [8, 24]. Hence stressing communications and computing infrastructure in an endeavour to process it's contents. Typically data is pushed into a stream application such that it must process this data stream on-line quickly. Stream computing applications tend to be time bounded such that if results are late, this has a detrimental effect on their value. The type of data in a stream can vary depending on application scenarios, but examples include data generated from sensor networks, stock ticks from the stock market, network traffic, web traffic click though rate, real-time health monitoring and social media data streams to name but a few [10, 22, 89].

2.2.2 Stream Data Model

Multiple works have attempted to classify stream types [34, 92]. These models include, *time series*, *cash register* and *turnstile*, each model describing the style of event in both order and frequency [10]. However, many of the application domains in which these models were envisioned were concerned with operational monitoring such as system log events and financial markets. More recently streams of data generated by social networks can be semi-structured or unstructured, this can carry information about multiple events within the same data stream flow. Within these works data stream have been defined as an on-line and unbounded sequent of events. Many of the elements of the streams can be homogeneous and are structured or in other cases heterogeneous, and as such are semi-structured or unstructured [42]. Formally, a data stream and the contained elements can be defined as a causal sequence of data elements $e_1, e_2, e_3, ...$ that arrive one at a time, where each element e_i can be viewed as $e_i = (t_i, P_i)$ where t_i is the time stamp associated with the element, and $P_i = (p_1, p_2, p_3, ...)$ is the payload properties of the element, or represented in this thesis as a message.

New messages are constantly generated from these data sources in an ever ending flow. To effectively process these messages, rather than using a batch processing model as outlined in past processing system, a new class of software framework has been devised which processes messages contained in a data stream on-line [80, 94, 101, 118]. A Stream Processing Engine (SPE), a class of software framework was developed to aid in both construction of applications to process data streams as well as providing a means to process single and multi-stream sources concurrently. Many SPEs tend to use a data flow abstraction where a stream application is structured as a graph of operations, generally refereed to as a Directed Acylic Graph (DAG). In such a graph, the application would contain a set of stream operators or User Defined Functions (UDFS), which were later called tasks when designed in imperative language constructs rather than as simple queries which execute within each operator. In general terms we will refer to these as tasks unless otherwise stated for precision and clarity in the text. Tasks tend to be structured in sequential order and also represented logically (many tasks could reside within a single or multi-host system) where each provided a specific operation on the data stream as messages were passed from one task to the next. This is generally refereed to as the *logical plan* to process the data stream. Many logical plans could be deployed to physical hardware to provide for parallel data flow processing [10].

2.2.3 Stream Operators

Stream operators themselves can be defined primarily by two major characteristics. First, *selectivity* determines an operators message selectivity on messages in the stream which tend to fall into one of three sub-categories. *Selective* tends to select specific messages fitting a particular filter function [27]. *One-to-one* tend to represent operators which select one message at a time in the stream. *Prolific* tend to describe operators which consume a number of messages. Secondly, operators tend to be categorised by their state, this also falls into three sub-categories.

First *stateless*, where each operator maintains no state from one message to the next, such as a filter function, filtering single messages in the stream. *Partitioned-stateful* describes state that is maintained based on some partition key, for example monitoring room temperature values above a particular temperature threshold. Finally, *stateful* where operators which maintain state of a more general nature, such as counting the number of messages receive in the stream thus far [42].

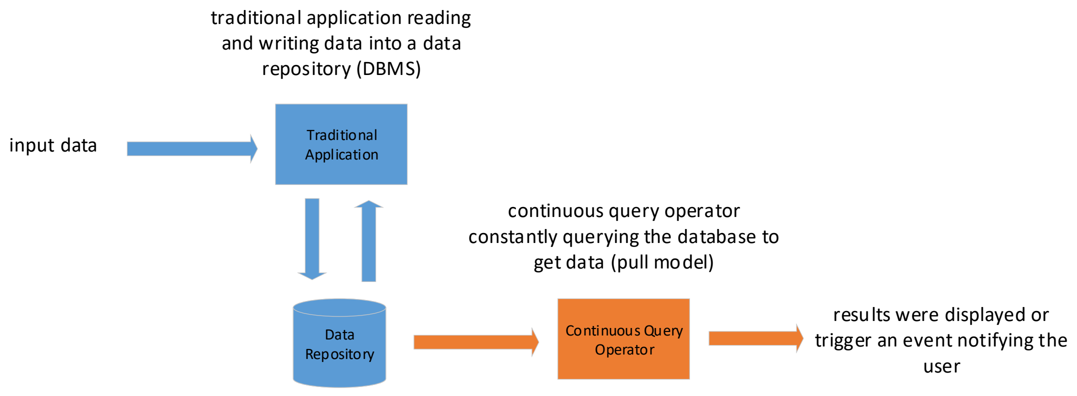
traditional application reading and writing data into a data repository (DBMS)
input data
Traditional Application
continuous query operator constantly querying the database to get data (pull model)
Data Repository
Continuous Query Operator
results were displayed or trigger an event notifying the user

Fig. 2.1: Design of a continues query stream computing system. Orange components represent explicit stream computing components. In this case, generation 1 continuous queries making queries to a data repository and pulling relevant data back in a pull data consumption model.

To understand the types of functions operators are used for, we must look at the four different generations of SPEs briefly and their evolution to date. Each generation can be outlined as follows:

- First, extensions to traditional Database Management Systems (DBMS) (execute continuous queries in operators)
- Second, distributed stream processing systems (execute continuous queries in operators)
- Third, user-defined functions (execute tasks),
- Fourth, highly distributed, edge computing and cloud (execute tasks)

The first generation as outlined in figure 2.1 was based on extensions made to DBMS which enabled long-running queries over dynamic data sets as data was updated in the database. This interface into the data allowed for interactions with the data to follow an SQL style syntax, where queries allowed a user to specify algebraic like operations on the data. Given the location of the data, most operations were restricted to a single machine and were not distributed amongst a set of machines. Much like the Aurora system from MIT and Brown which pioneered this paradigm [2]. Figure 2.1 represents this model, where a stream application consumes and populates data with a DBMS. Aurora was a single host system which consumed streams of data but it's performance limitations were found quickly, given the workload that is required of an SPE can be very intensive. Vertically scaling, adding more resources to a single machine has physical limits. Bursty data streams and load shedding were first mentioned in this within this SPEs published works, we revisit this topic in section 2.4.1.

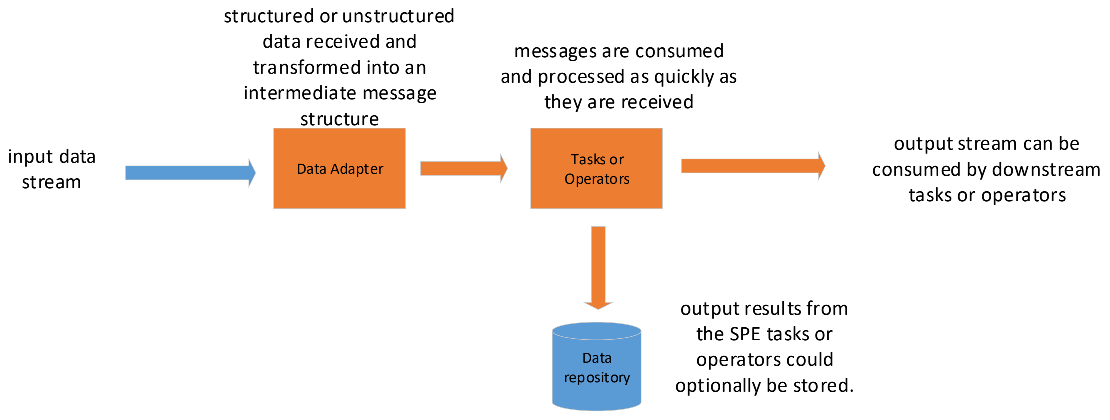
structured or unstructured data received and transformed into an intermediate message structure
messages are consumed and processed as quickly as they are received
input data stream
Data Adapter
Tasks or Operators
output stream can be consumed by downstream tasks or operators
output results from the SPE tasks or operators could optionally be stored.
Data repository

Fig. 2.2: Design of a stream computing processing model. Orange components represents explicit stream computing components which tend to fall into the description of a Stream Processing Engine (SPE) using a push data consumption model.

In general the limits of a single host system were found quickly, where computational resources were found to be limiting. The second generation of SPEs describes solving this resource problem by designing SPEs to process data streams across a distributed DAG involving many machines. This allow SPEs to scale by locating stream operators on different physical machines in single cluster, maintaining the same logical DAG structure. Operators in this new setting needed to communicate across network interfaces, which tended to follow a message passing paradigm. Systems such as Borealis[3] and TelegraphCQ[29] represent this generation. Figure 2.2 illustrates this at a basic level with a single operator or task. Figure 2.3 illustrates what a pipeline of Processing Elements (PEs)could look like with stream splits and stream mergers. A PE general refers to a single or set of application specific operations applied to the data stream. They can be consider a set of application tasks encapsulated into a single logical unit. This processing model introduced many new complexities in load-balancing, message routing and resource-management in a distributed system setting. Given the great improvements made in generation 1 and 2 SPEs, operators continued to followed a DAG model and operations consisted of algebraic like operations. By algebraic we mean operations similar to SQL like functions such as Filter, Union, Aggregate, Join and Map which tend to be run as continuous queries against messages within the data stream. Figure 2.3 represents this generation.

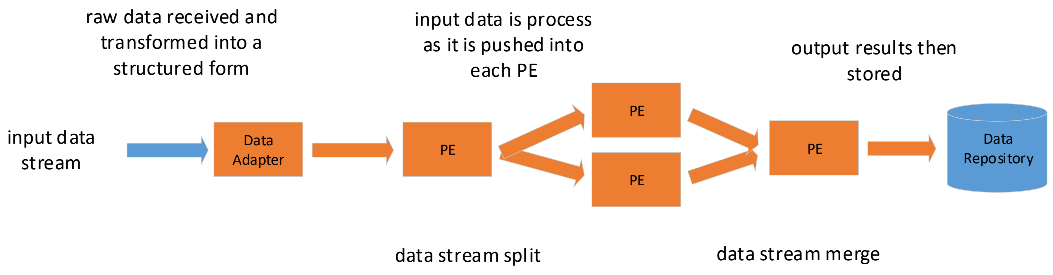
raw data received and transformed into a structured form
input data is process as it is pushed into each PE
output results then stored
input data stream
Data Adapter
PE
PE
PE
PE
Data Repository
data stream split
data stream merge

Fig. 2.3: A typical stream computing pipeline with multiple downstream PEs feeding from the output of upstream PEs.

System S from IBM, later called IBM Infostream [12, 23, 49], classified in the third generation allows for static declarative operators as well as User-Defined Functions (UDF). Rather than being a set of operations which can be performed on the stream, UDFs allow for a more natural programatic style of interaction with messages in the stream. UDFs encapsulate stream application logic.Allowing application developers to write specific functions which process messages in the stream in Processing Elements (PE). Apache S4, formally Yahoo! S4 [94] and Storm from Twitter[119] fall into this category. These systems removed the declarative SQL like syntax opting for a Object Oriented programming paradigm instead, where messages are received by PEs, processed in some way, exposing some result or creating a new stream which is then sent and delivered to downstream PEs performing some other function. However, with more complex functions and the existing processing constraints with stream computing applications, a new scaling model was needed.

The fourth generation regards the use of elastic resources to help scale applications and provide better fault tolerance semantics around stream based applications. In cloud computing, resources are readily available, which can scale tasks, abstracting away the underlying physical resources. However, with the benefit of cloud elasticity comes new challenges in maintaining application state within tasks as tasks are scaled and duplicated for performance purposes. Other existing challenges regarding effectively managing stream splits and stream merge events while maintaining application state also remain. SPEs such as StreamCloud [54] and Stormy[85] illustrate examples in this area. Twitter Heron, the successor to Twitter Storm also falls into this category. Storm was not able to scale well with the ever increasing data generation rates year on year within Twitter. Further, it was difficult to debug and deploy [80] and thus has been replaced with this newer engine.

Scaling stream computing applications effectively and quickly, particularly through horizontal scale techniques has been difficult and remains as current challenge. Stream splits and mergers as well as the volume of messages in a stream coupled together with intensive workloads which are time bounded, have made stream computing problems difficult to solve. Additionally, in general SPEs have a push based data ingestion model which allows for quick consumption of messages in the stream, rather than using a pull or retrieval model. Using a push based model has led to a new problem where at times a stream application and it's resources become overloaded as too many messages are pushed within a short period of time. We focus our attention on this in the next section.

2.2.4 The Bursty Data Stream Problem

At times stream computing applications can suffer from overload where available application resources are insufficient to process messages in the stream within application time bounds. Under normal message loads, application results can be published on time but during high input message events application resources cannot keep up with input message volumes. This has become known as the bursty data stream problem [6, 62, 73]. A stream computing application consumes messages in a stream using a pushed based model (e.g: data sources push data into the application for processing). If enough resources are not available to process the data stream within time bounds, messages begin to build up in input queues in the SPE. This has an effect of inducing longer than desirable queuing times on messages. If the ingestion rate, the rate of message consumption and processing of the stream, remains the same and the rate of data flow remains at this higher level or worst yet, increases, queue volumes increase to the point queues overflow with messages. As a result application results are not only late but input message becomes lost. Workloads and applications as outlined in section 2.2.1 tend to encourage highly uncertain input data flows, of unknown duration and velocity [114, 117].

A stream computing application which becomes overloaded must change and adapt to the new input message rate in one form or another such that it can process the data stream at the higher temporary input rate while continuing to produce timely and accurate application results. This can be rationalised as a resource optimisation problem for which the current CPU capacity of the application or a specific PE is not enough to process the current data stream workload. As mentioned in section 1.5 we scope this adaptation to computational and CPU based resources. Deduced from this assertion, there are two approaches which ensue [116]. First, we either increase the available CPU capacity such that it matches the new workload processing requirements and thus increase the ingestion rate of the application to match that of the bursty data stream. Or second, we reduce the cost of processing the workload such that the existing available processing resources can increase their ingestion rate to match the new input rate of the stream [16]. Additionally, the suggested application adaptation should also occur quickly such that in the transition from one application state to another, minimal or no input or intermediate data should be lost. Finally, the adaptation activities should have low over-heads such that in an already overloaded system, further resource consumption should be low for non-functional purposes as to prevent further CPU saturation.

In the literature, these two application adaptation approaches outline the state of the art literature review which we present next.

2.3 Literature Review

The literature pertaining to handling bursty data streams can be divided up into two main categories namely, data based approaches in section 2.4 which attempt to modify the input workload by reducing it, using load shedding techniques or synthesis structures.

Secondly, task based approaches as outlined in section 2.5 attempt to scale up the CPU resources of the application thus increasing ingestion rates. This is achieved through a variety of techniques such as, task and pipeline duplication to process streams concurrently or using round-robin techniques, sending different messages through different paths. Both streams end up merging later. Others include operator migration from one heavily loaded PE to another PE. Finally, elastic resources are used in the cloud. Tasks, operators and User Defined Functions

(UDFs) are used interchangeably throughout the literature depending on the focus of the SPE in which they are used, for simplicity going forward we refer to them as tasks for clarity purposes, unless otherwise stated.

We further present a third category call task modification in section 2.2.4, which changes tasks at runtime to reduce application processing costs thus increasing message ingestion rates but at a reduce accuracy cost to the applications result.

Figure 2.4 outlines the individual techniques contained within each category:

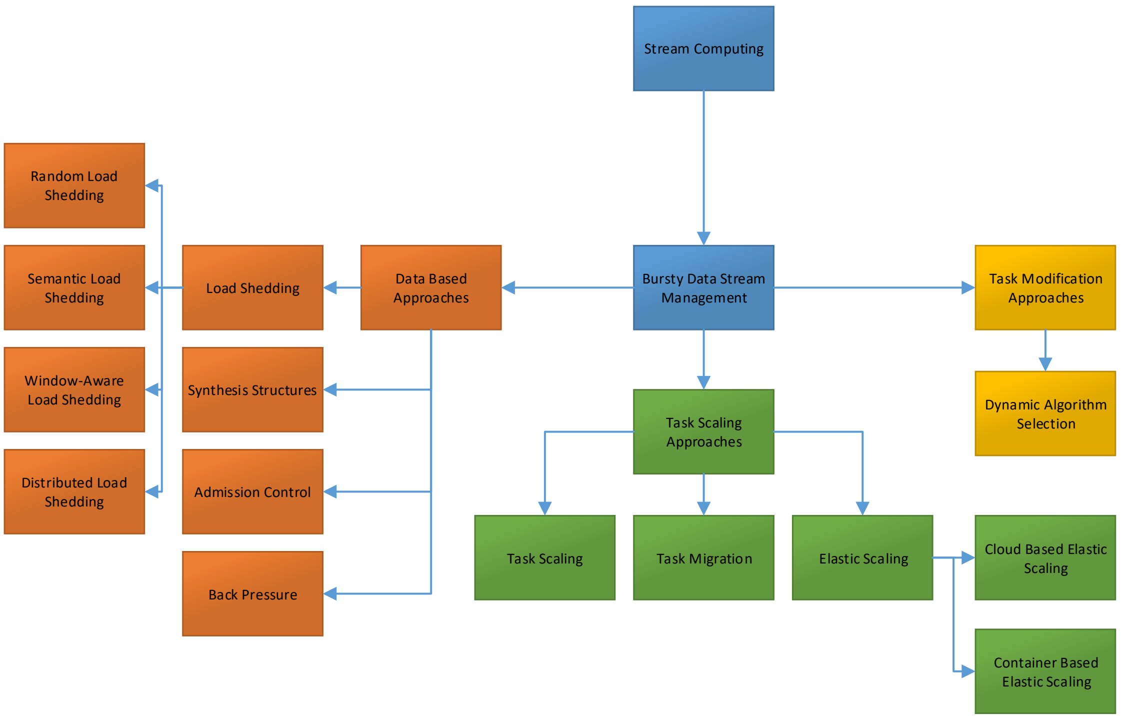

Fig. 2.4: State of the art overview containing adaptation techniques analysed to manage bursty data streams. Orange components indicates data based adaptations which modify input stream data. Green components represent task scaling approaches. Yellow components represent task modification approaches specifically tailored to handler bursty data streams.

We begin this review by addressing data base approaches first.

2.4 Data Based Approaches

The review of the literature begins by addressing data based adaptation techniques. Data based adaptation approaches attempt to reduce the potential workload of data streams by reducing the amount of messages to process during bursty overload periods [16, 28, 114]. Using this approach message processing rates can be increased. This tends to be achieved by reducing the quantity of messages in the data stream in a controlled manner. As a result growing input message rates do not cause messages to reside in queues for longer than desired periods of time. Additionally, the selected messages which are retained are selected in a controlled manner, unlike the efforts of buffer overflow which occur when a bursty data stream fills application buffers, causing data to be discarded uncontrollably.

However, as a result of this intentional data loss there is an application accuracy reduction within the results of the application as the application processes a subset of messages within the data stream. As such many techniques in this category perform message selectivity operations depending on application context. For example, messages are selected at random to keep application results statistical validity, or messages which are deemed more important to the application are retained while messages of little value are discarded first [19]. Further when an estimated result or imprecise result is generated with appropriate statistical techniques, error bounds of that result can be estimated.

One form of data based adaptation is load shedding where data is dropped or discarded selectively as input data rates surpass ingestion rates. A number of strategies for load shedding exist in both local and distributed architectures which we will discuss next.

2.4.1 Load Shedding

One approach in tackling overload in stream computing is to discard input message at the input to overloaded PEs. By reducing the amount of input messages entering stream operators within PEs, system overload and its consequences can be avoided. Discarding data held in input buffers, helps ensure output result latency will not violate application timeliness requirements. In the simplest of cases, load shedding can be described as a simple filter operation if the system is overloaded then data is dropped until the system is not overloaded. However, this approach is detrimental to the output result and the accuracy of that result as input messages which have not been processed do not produce a result. Data stream overload is not a new problem and bares close conceptual resemblance to congestion in computer networks [111].

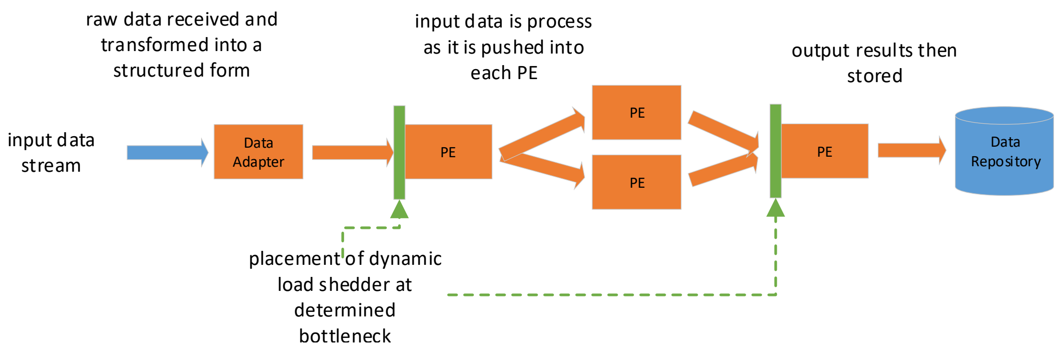
raw data received and transformed into a structured form
input data is process as it is pushed into each PE
output results then stored
input data stream
Data Adapter
PE
PE
PE
PE
Data Repository
placement of dynamic load shedder at determined bottleneck

Fig. 2.5: A logical stream computing pipeline with load shedders present at the input to each processing element. This diagram also illustrates stream split and stream merge activities which are used for parallel workloads.

In computer networks, routers and other network devices discard data during transient network overload. At the simplest level this is done randomly giving all network packets a fair chance of arrival in the simplest case. Other semantic mechanisms have been implemented to ensure overall data flow accuracy, either by higher level application constructs in UDP or via TCP congestion and flow control [87]. In the case of Quality of Service in computer networks, data with strict latency requirements such as streaming video, VOIP or networked computer games are given priority over data that does not have such latency tolerances such as simple web traffic like web page retrieval. When routers become overloaded, web surfing traffic can be discarded to reduce network load at the router level to ensure latency intolerant traffic is given priority, continuing it's service uninterrupted. As a consequence of this activity, web site loading times for the delayed traffic increase, but given the nature of this data, delays can be tolerated [111].

In stream computing however, this approach becomes more complex. Load shedding is a delicate balance between the amount of data discarded versus incurred application result error [3, 52]. Load shedding strategies must ensure the majority of processor cycles are reclaimed from the discarded messages while ensuring that message loss in a data stream is minimised where possible.

Aurora is a generation 1 SPE [2, 15, 17]. It was designed as a single hosted log monitoring application out of work from MIT and Brown. It also contains the first mention of load shedding as it applies to stream computing in the literature. The researchers focused on both overload detection, placement of load shedders in the DAG and how to determine what data should be retained whilst discarding other messages during the bursty event. In its processing pipeline of connected PEs in a DAG, the question was asked as to where to place load shedders to maximise the reclaim of resources i.e.: CPU time or memory, whilst also ensuring that the applications output accuracy was effected as little as possible. The recommendations of this work is to ensure that you load shed at the right time in the right place in the pipeline with the correct data, so as to not incur a loss of application accuracy and ensure end-to-end latency requirements are met. Figure 2.5 outlines their recommendation by having load shedders as far up stream in the DAG as to reclaim the most amount of resources. Discarding messages up stream have a larger accumulative effect on resources when not sent downstream. However, shedding far upstream has the largest detrimental effect on output result accuracy given that discarded messages upstream are not processed in any downstream PEs.

Further in their work, to measure overload in Aurora, the entire processor cycle requirements of all messages in the single hosted application were measured to determine overload. Load coefficients, or the cost of processing, is predetermined for each stream operator in the DAG and total counts for processing each message in the stream were derived from this. Giving a total cost, in processor cycles. If this value was greater than the system available cycles, the application was assumed to be overloaded and load shedding was initiated proportional to the overload amount [73, 116]. However, in this case the messages in the stream were of trivial data types such as integer based temperature measures accompanying algebraic like operators as mentioned in section 2.2.3. Thus workload processing costs were easily computable to a high degree. Consequentially, more complex applications were not possible given UDFs and tasks were not available in this works.

Load shedding operators tend to be inserted dynamically into the query plan before PEs when overload is detected and deactivated when the input rate of the stream is of a sufficient level it can be processed with available resources. In MavStream [73], load shedding is integrated into the PE such that the load shedder and PE share the same processing cycles rather than being a dynamic activity running within it's own thread. This has the benefit of reducing interprocess communication overload as well as reducing any application plumbing that might be needed at development time to connect load shedders to PEs. But it looses any pipeline benefits given the Aurora system was multiple threaded, such that if the dedicated load shedder was busy discarding messages, the PE was also occupied processing messages within application operators and thus could process more messages for the same time window than MavStream.

We next discuss the types of load shedding available in the literature.

2.4.1.1 Random Load Shedding

Random load shedding is the simplest form of load shedding as implemented in the Aurora SPE [28, 116]. When system overload is detected, load shedders are activated discarding input messages using a random predicate. All messages in the input stream are treated equal hence data discarding is a function of resource availability (as overload increases beyond the capacity of the system, so too does the percentage of message discarded). Aurora [2] and later Borealis [3], a second generation SPE which distributes operators across multiple physical hosts, implemented random load shedding as a weighted probability as a function of system load, and is expressed in the form of $p = 1 - (x - 100 * STEP\_SIZE)/x$, where x is the drop amount. x here must be greater than the combined drop to load ratio to maximise loss utility. Shedding is triggered when output result latency violate application thresholds. As overload increases, the weight attributed to this probability increases at a proportional rate to the overload determined in the system. As mentioned all input messages are treated equal with this mechanism, utility of a message to the applications outcome is not considered.

Borealis implements a Load Shedding Road Map (LSRM) to indicated how many and where messages should be shed throughout the DAG. Additionally, input messages are dropped in groups using a Step Size, where the cost of dropping a message must be greater than the cost to drop the message. The Step Size defines the increments in which more messages are dropped there after, ie: 3, 6, 9 messages. However as this may be it's advantage top drop in greater numbers, it is also it's disadvantage to output accuracy such that larger amounts of messages maybe dropped than desired when the Step Size is large. Further, the message model assumes messages do not have a large level of complexity and as such when generally applied, does not consider message utility and the effect has on the output result. One larger issue with random load shedding assumes that messages are all of equal importance, even in log event monitoring, which Aurora and Borealis were developed for, that is not always the case. [3].

STREAM [14] takes a slightly different approach rather than using random predicate to select a message drop, STREAM uses random sampling to select which messages are retrained and delivered to the PEs for ingestion. Data which is not sampled is not consumed. Thus, sampling at a rate that does not match bursty data stream input rates leads to unintentional data loss. However, local resources do not become overloaded as such, given that as messages are selected as fast as the sampler can select them. However, this approach is more susceptible to unintentional data loss as a result of it's capture model.[18].

Improving upon single message decisions, random load shedding has also been applied in sliding window scenarios, were a window of input messages is considered for dropping rather than each message individually. This has been shown to increase message throughput and decrease message drops by a small amount given that discarding decisions are consolidated together. For example, 10% of a window of messages containing 100 input messages are dropped, while 90 messages continue to the PE [116].

The use of random load shedding allows for a fair and statistically valid output result in the computation, where error bounds on the output result can be estimated, making this ideal for data mining and statistical applications [40, 41] but limiting for applications of a more complex nature which involve UDFs or tasks.

2.4.1.2 Semantic Load Shedding

Semantic load shedding, another contribution from works within Aurora and Borealis [28] describes utility based load shedding. Semantic load shedding makes specific drop selections of input messages in the data stream based on their value or utility to the application. This semantic understanding where applicable, can increase the accuracy of the output result when load shedding is required. By assigning a utility value to each message in the stream, based on some intrinsic property or value, lower utility value messages can be discarded in favour of retaining more valuable messages [3]. Messages within the SPE are filtered based upon an application utility of some sort, for example distance, or temperature thresholds. An internal frequency table or histogram is built up as messages are received in the load shedder. When overload is detected by again measuring the total cost of CPU required to process all messages weighted against the current system CPU capacity, a pointer into the histogram indicates what bins of the histogram to drop. The overload amount determines which bin or set of bins to select based on the volume of data represented by each bin. Bins are organised from lowest to highest utility, such that if bin

1 is selected, containing 10% all messages, messages signified by bin 1 are dropped as messages are received by the load shedder. If more messages need to be shed to match drop volume to overload amount, bin 2 and subsequent bins are selected proportional to the overload. Thus depending on data distribution in the histogram, more messages may be discarded than desired with this selection approach. Further, for each application type specific application filters must be devised to attribute some semantic value to the data contained within messages in the stream. This still assumes that messages contain trivial properties and as such that cost of a semantic drop in more complex messages models is not considered.

Work on the MavStream recommends numerical values for utility rather than boolean output on semantic filters so as to allow for a gradual gradient increase in drop messages [73]. This is preformed through select operators using a syntax akin to SQL. For example, the highest and lowest values for a specified filter threshold are retained as messages are processed at runtime. During overload message drops are preformed on a range of values as input load increases, centring around the medium, up to or down to the limits of selected threshold values. This is dependant on the filter type requested, as not all semantic filters provide such rich selection functionality and is very much application dependant in the ranges which are selected [73].

Semantic load shedding can be an expensive activity to run, particularly in an application which is already overload. Given the cost of semantic load shedding some work had been done in attempting to move as much of the computation off-line to reduce the cost of semantic drop determination and the required processor time [116]. The Borealis project has devised a unique means to measure the loss tolerance for discarded messages by using a value-base QoS graph. By comparing messages against this predetermined graph, their value to the output result of the computation can be measured and a utility derived at each stage in the DAG. Additionally, the use of this graph can define an estimate error bounds on the curtailed output result [1, 14, 116].

Loadstar is a load shedding scheme for classifying data streams in data stream mining. This work addresses the issue of how to classify data that is to be discarded from a load shedder. Using a Markov model they predict the distribution of values which then allow them to generate classification predictions on what data to drop [31]. However, the architecture defines that a single load shedder is used for all streams entering a selection of stream operators locally and thus semantic model generation is applied to all streams equally rather than being a specific model derived for each stream and the different messages types and properties contained within.

As such the model becomes inaccurate when data stream message properties differ.

Mozafari et al., propose a load shedding algorithm for server overload scenarios which samples data streams to provide an output approximate error result. They state that many applications can accept this provided results are produced with accuracy guarantees and error boundaries [91]. They achieve this using sampling distribution using past performance of a stream and SPE performance as future indicators for what data to sample and the entailed error bounds on approximate error results very similar to the work in Stream but equally as similar to the work in Borealis. Borealis focused on reducing the eventual error within the applications results through the QoS graph mentioned earlier. However, the underlying assumption is that stream applications consist of a set of algebraic operations executing continuous queries on the data stream. Rather than operators being more complex UDFs or tasks with more complex application logic. Additionally, they assume that past messages in the streams can be used to predict what the content of messages in future will contain which may not be the case, as streams evolve.

In general, semantic load shedding is an active selective filter function requiring additional computation and memory resources to analyse and process the data stream in an already overload application. The additional filtering computation adds additional latency to the end-to-end processing time for each message analysed. Unlike random load shedding, semantic load shedders tend to require more resources and tend to be application specific such that they need to be developed and configured for each developed application unlike random load shedding. However, they do allow classification of data which is highly beneficial for many application domains. Further, most designs assume operators are algebraic rather than being UDFs or tasks with more complex logic and as such messages models in these domains tend to be simplistic in nature.

2.4.1.3 Window Aware Load Shedding

Further improvements upon semantic load shedding leads to window load shedding [111, 28]. Message drop considerations are not applied to individual messages in the stream but aggregate sets of messages within a sliding window. Sliding Windows tend to be bounded in time (e.g. a 5 minute sliding window) but can also be bounded by element count (e.g new window created after 10 messages have been received). Through the use of sliding windows this has the capacity to improve output result accuracy by aggregating values within the window and thus reducing the volume of messages within the stream during overload periods. The transformed contents of the window are then sent to the associated PE for processing, which in most cases is now processing a sub set of the original amount of messages. This work uses aggregate queries consisting of one or more stream aggregate operators within a sliding window. The operator is aware of the window properties such as window slide and window size. Input messages are logically divided in the input stream into groups of windows as they arrive. During overload, messages within the window are discarded, producing a small accuracy penalty on the output window aggregate result.

For example, if five messages each containing temperature values arrive at the load shedder pertaining to temperature of a floor in a building, the messages are grouped together by the floor number. Assume three of the temperature values are the same while the other two differ. During overload data must be discarded, as such the two values which differ are discarded. The floor temperature to the applications perspective is determined to be one of the three remaining messages. In essence, five messages become one, thereby reducing load for only a small deviation in the output result. This may be a worthwhile approach which has the potential to reduce loads significantly during overload periods.

Conversely, this is dependent on a grouping mechanism being available and whether the application can tolerate an aggregated input value. The additional complexity of the computation in the load shedder further takes from the available resources to process input data in the PE when sharing the same set of raw resources. Further, messages that are disallowed from the window are disallowed from starting new windows later.

Window harvesting presents an additional improvement to window-aware load shedding for CPU overload [48]. This is an in-operator load shedding technique for multi-way windowed stream joins. As outlined in figure 2.5 depending on application requirements, streams may be split and subsequently joined back together later for load balancing or concurrent processing reasons. The basic approach is to use only a certain fragment of the join window for processing. By sub dividing a join window into smaller window sizes, CPU demands are reduced for the operator per window as dictated by a throttle function which dictates how much data to drop in the event of overload. The reduction in window size then reduces the level of computation per window to aggregate results, there by reducing the delay to the PE [115]. Moreover, the use of timestamps within each input streams decides which segments of the join window are more valuable to the output result and thus lower value can be ignored. Further increasing output result accuracy and latency. Thus this approach follows a sampling strategy akin to Stream but within the confinement of a sliding window during a join operation of two streams.

Window-aware load shedding provides for an aggregation function over the data contained within the window, with a similar objective to Synthesis Structures as we outline in section 2.4.2 later. Fundamentally, any aggregation strategy intrinsically looses output result accuracy when different data types are combined together. Work in this area has attempted to focus on reducing accuracies in output results when results are aggregated with a primary focus on output result latency [4]. As with other load shedding strategies this assumes messages in the stream to be of a primitive nature to enable aggregation, such as integer based data, like temperature or frequency counting.

2.4.1.4 Distributed Load Shedding

With the move to second generation SPEs, distributed load shedding was considered as part of the framework. In this case, load shedding operations needed to be coordinated across multiple distributed physical node. Unlike prior work, most load shedding strategies were assigned locally to a single physical node. With the use of distributed resources in stream computing, stream computing applications can scale horizontally and thus consume richer data while also increasing application complexity. However, bursty data streams continued to be a problem which now needed to be solved in a distributed setting. Distributed load shedding here refers to load shedder coordination including load shedding duration and quantity.

Borealis was the first in the literature to suggest to use distributed resources and to also implement distributed load shedding [28, 111, 115]. In this work, distributed load shedding is modelled as a linear optimisation problem, for which they have proposed both a centralised solver-based approach and a distributed approach based on meta-data aggregation and propagation. Both of the implementations are based on generating a series of load shedding off-line plans in advance with estimated operator processing costs. The focus is on tree-based server topologies, while the query network can have both stream splits and stream mergers where data is processed in different PEs in the distributed network. An on-line plan is used to determine how many messages to shed in different locations in the DAG based on the measured overload amount.

In the central server approach, a server is selected as the coordinator node, which contacts all nodes in each DAG to periodically collect their current processing rates and CPU utilisations. During a bursty data event with the retrieved data, specific distributed PEs are asked to drop specific amounts of messages in the stream based on the pre-computed off-line plan and newly generated on-line plan. Local load shedding is preformed randomly, semantically or in a window determined in the plan off-line. The coordinator then monitors the load on each node. If an overload is detected the coordinator selects the best plan to apply and sends the required information to appropriate nodes to start shedding at a desired rate.

Given the centralised coordinator and the periodic utilisation updates it asks of all PEs, it can take some time to generate a shedding plan. Scaling this approach is limited as more nodes are added. On-line load shedding plan generation times increase given the coordinator must query each node within the selected DAG. Furthermore, between the time of an overload and the time it takes to get a report from a node, subsequent processing delays can occur and in the most severe cases data loss in overloaded PEs can occur while the coordinator retrieves plans for other PEs and the overloaded PE.

To this end, Borealis also implements a decentralised approach improving upon the original centralised approach. This is achieved by removing the direct dependency on a central coordinator which maintains global system state and thus is slow to update. Instead, each node maintains its own Feasibility Input Table (FIT), which expresses what a feasible input load for this nodes is, and using a tree structure can trace the path upstream from where its input data originates from. With FIT an overloaded node can decide to shed messages locally to alleviate loads if it has sufficient scope to do so. If not, it can contact upstream nodes to initiate load shedding activities there, shedding at a selected amount, using random or semantic load shedding. The load shedding active is determined in an off-line pre-computation step, as well as the placement of load shedders in the DAG. If upstream nodes contain a stream split the preference to shed there rather than in the overloaded PE is made. This decision is made to maximise the regained CPU cycles given that message drops upstream have greater benefits to alleviate load in both streams after the split [27].

Distributed load shedding yields many benefits not present in the centralised approach. For example, one major advantage is that the coordinator itself cannot become overloaded given that emergent load shedding behaviour occurs locally. Additionally, load shedding plans can emerge at a faster pace, given that nodes monitor their own load and capacity rather than a centralised orchestrator. Each node can then itself request its upstream PE for direct action. In both approaches, discarding messages at an upstream PE in the pipeline subsequently increases the inaccuracy of the output result at any downstream produced result, compared to discarding data further down in the DAG. This is even more the case when a stream is split at the upstream PE which is the original source of messages into the SPE, incurring an accuracy penalty at all downstream PEs as a result of discarded messages [117].

2.4.2 Synthesis Structures

Synthesis structures are structures which approximate the contents of the data stream. When using these structures the volume of messages in a bursty data stream can be reduced while accuracy can be retained using approximation techniques.

For example Rivetti et al. propose a model called *Load-Aware Shedding* which unlike prior semantic load shedding works provides a runtime dynamic and on-line based cost model to compute individual messages in the stream using *sketch* data structures. Sketch data structures attempt to approximate the messages contained within the stream, using techniques like aggregation (e.g: using a histogram with bins and frequencies for the types of messages which arrive). Rivetti et al. assume that if the total execution duration of a message is known before hand, they can then predict the total processing time of each task in the stream and thus can drop messages which will violate message ingestion times before hand using prior knowledge of all previous messages received [101]. This is quite interesting work in that they have provided both a formal proof of the algorithms but have also implemented it and evaluated it on Twitter Storm, a generation 3 SPE.

This builds on work outlined in the StayingFit design for load shedding, which we describe in section 2.4.1.4. However the use of sketches tends to approximate the contents of the data stream and over time older message approximations can skew newer values such that the best guess for a message processing cost can in fact drift away from the reality it took to process it.

BlinkDB [4] trades query accuracy for response time, enabling interactive queries over massive data by running queries over sample sets of data rather than the whole set. Results are then presented with approximate error bars given that a sample of the data set was analysed. They show that they have improved upon an implementation using Hive, they produce results which are 100 times faster when querying over a 17 TBs data set. Approximate results were produced with error range of between 2-10% from the ideal value.

2.4.3 Back Pressure

Work in Twitter Heron [80] a generation 4 SPE, uses back pressure to alleviate load on local Heron Instances which become slow at processing input messages due to either slow processing or increased queueing durations in the local instance. This is achieved using two approaches, *Spout Back Pressure* and *Stage-by-Stage Back Pressure*.

In *Spout Back Pressure*, each Heron container, encapsulates a number of Heron Instances which process the data stream within their respective application logic. When the local Stream Manager in the container detects that an instance is overloaded it makes a request to the responsible input spout (input adapter to the topology or DAG) to reduce the volume of messages being sent to the instance. When the overload has passed, a similar request is made to the spout to increase message rates returning them back to normal levels.

State-by-Stage Back pressure works an a grander scale, when a Heron Instance receives traffic from an upstream Heron Instance and that in turn that instance receives traffic from another Heron Instance and so on an so fourth. Each stage makes a request upstream to it's parent instance requesting a reduction in data stream traffic all the way up the topology (DAG) until the request makes it's way to the input spout for the topology. The input spout then reduces traffic volumes. After the overload period has passed the Stream Manager detects this in the previously overloaded instance and a similar propagated request is made to increase input rates, returning them back to normal levels.

Heron is primarily focused on ensuring that queues do not fill which can result in subsequent data loss. Result latency as a consequence of back pressure is less of a concern.

2.4.4 Analysis

Load shedding is an important strategy in overload data stream handling. Originating in network data management, it makes sense to reduce the quantity of input data into overload PEs to ensure a level of control on the output result in both latency and in result accuracy. Random Load Shedding, improves upon output result accuracy more so than simple access control approaches. With random probability, a selection is made on which messages to drop or alternatively which messages to accept. Thus ensures a fair chance of accepting any message.

Additionally, random load shedding tends to use less resources than other load shedding approaches which is of high benefit given that the processing system is already under duress [111]. Though, its fairness in message selection is at odds with output result accuracy given that not all input data may be required for the computation at hand. In this regard semantic load shedding is a better fit when specific messages should be retained and discarding other messages with less value to the results of the application.

Semantic load shedding provides an alternative to random load shedding and in some cases, an improved approach in terms of output result accuracy when appropriate. As mentioned, data that is less relevant to the application can be discarded with the use of semantic filters. In doing so, it increases output application accuracy during overload when compared to simple access control and random load shedding. However semantic filters must be created for each specific application increasing development investment.

At runtime semantic filters require more computational resources than random load shedding which subsequently takes resources from an already overload application or more specifically more resources from the local overloaded PE. Splitting a load shedder and PE into two separate threads is an obvious means to remove this constraint but in a multi-threaded PE processor time is still taken from the PE and given to the load shedder during overload. Additionally, given semantic filters perform more work than random load shedders, in message analysis, they can delay message arrival times to a larger degree than random load shedding.

The use of semantic filters is very much dependent on the requirements and scope of the application and whether filters can fit this model, i.e.: there are distinguishing features of a message which we can use to discard some messages from others. In a complex pipeline with multiple PEs in the DAG, an upstream semantic drop can effect the downstream PEs getting message, making them complex design considerations in placement and shedding quantity [1].

Window based load shedding creates synthesis structures on messages within the incoming data stream using window framing techniques, not directly discarding results but creating aggregate data values for simplistic aggregate operations. In such applications, this type of structure is well suited to frequency counting or generating descriptive statistics for example, such as means and medians. In more complex cases however, window base load shedding is not applicable given that the data cannot be aggregated [115].

Distributed load shedding provides an approach to handling overload and data shedding in distributed environments. A centralised approach to query plan load shedding decisions reduces complexity in system structure. However, due to processing complexity in plan generation in the centralised case, as well as requesting individual node state, delays in its generation may ultimately lead to the latency violations one tries to avoid, perhaps even inducing data loss during extreme bursty data events. This becomes more prevalent as node count increases, hence a centralised approach does not scale well. Comparing a centralised co-ordinator approach to a distributed approach, where each node monitors its own state and induces load shedding locally or requesting it upstream when required. Information from nodes can decide when, where, and how much load to shed in a more scalable manner.

In the distributed case convergence time (time it takes from overload detection to load shedding action in the correct place in the DAG) of the load shedding on-line plan, may become a factor in the overall stream pipeline and subsequent data may be lost until a plan is decided upon. This may become particularly apparent in highly volatile data stream cases. Though this approach decreases the requirement of global knowledge before action can take place, node polling penalties are removed when considering the centralised co-ordinator approach [3, 27].

Fundamentally, load shedding is an unsafe approach [62] to reduce the effects of stream burstiness. Messages are lost by either dropping input messages controllably or by creating synthesis structures which summarise arriving messages using descriptive statistics. As data rates during a bursty data event rise and data adaptations techniques activate to compensate. However, by discarding input data, CPU time can be reclaimed but this results in application result inaccuracies and tends to be at a proportional rate to the overload amount. If a PE is overload by 50%, 50% of the input messages must be discarded but the question arises as to which 50%, and this comes down to shedder type and context specific information [85].

Another interesting limitation of load shedding relates to its effects on stateful applications. In stateless applications the duration of load shedding has little or no impact on the application operations outside its activation periods. That is to say that the effects of lost data in a stateless application does not effect periods of time when the system is under load. In stateful operators or tasks, where state is maintained overtime in some form or another, load shedding can have a drastic effect on the results of the application. Messages which are discarded will not be seen again in the stream. During high data rate periods messages that are relevant to the application are still lost at a proportional rate to the applications overload. This has been noted in the literature.

MillWheel, a stream processing application from Google [7] outlines that generation 1 and 2 SPEs which have implemented in the past with load shedding techniques can be described as streaming SQL systems, which solve simple and succinct streaming problems. Continuing they argue that intuitive state and complex application logic is more naturally expressed using the operational flow of an imperatives language rather than declarative languages like SQL with limit algebraic like operations. S4 by Yahoo! now Apache S4 makes a similar argument in their work such that application logic is expressed more freely and more easily in higher level application constructs than declarative languages [94]. Both make the argument that stream applications which maintain state may need a different approach to handle application overload.

Other approaches to solve the burst data stream problem are required because of the lack of stateful application considerations, the level of application complexity outlined in load shedding examples, the limited semantics of an SQL like language to query data streams for certain application types. More importantly certain classes of applications with strict Service Level Objectives (SLO) which cannot tolerate data loss require other approaches to solve this problem [80].

In the next section we look at task scaling approaches which aim to try and scale stream applications which are generally written in imperative languages at runtime. The aim of the next section is to discuss solutions which process higher input message rates by acquiring additional processor capacity and CPU resources at runtime.

2.5 Task Scaling Approaches

As mentioned in section 2.2 there are two approaches to handling bursty data streams. This section outlines operator scaling, which addresses the second possible approach by increasing the available processor capacity of the application to match the new input data rates of the stream.

2.5.1 Task and Pipeline Parallelism

In general scaling of stream tasks relates to scaling up the applications computational resource capacity during system overload. This work is implemented as distributed and/or parallel computing techniques on a variety of platforms [21] to add more computational resources at runtime.

Note this section pertains to techniques which scale tasks within the confinement of the distributed infrastructure where the distributed system is of a fixed size.

With the current capabilities of hardware in both fixed infrastructures and mobile computing devices, parallel execution models on multi-core processors and distributed nodes are possible for a vast array of stream computing applications. A divide and conquer approach has been well adopted to counter bursty data streams by duplicating tasks. Thus this increases ingestion rates of the stream during overload using parallel and round-robin DAG techniques [54, 68, 71, 107, 124]. The approach in each of theses works, rely on either partitioning of the data stream at runtime where input data is split such that a DAG is now processed in two parallel flows. Or using round-robin techniques with load balancing such that every second message is sent down a different parallel path rather than streams being duplicated.

In the parallel processing sense in Aurora, Borealis and Flux [3, 5, 106, 118] all have presented a share-nothing architecture to allow for scaling parallel operators in parallel workflows within duplicated streams. In a DAG it becomes difficult to scale tasks given that stream splits cause eventual stream merging operations to be sensitive to load given two or more input data streams merge in a single task. Thus parallelism is preferred to process streams concurrently. In such a case, a single stream is split and is not later merged. There two or more DAGs process the same stream where the operators, UDFs or tasks process the stream independently. Thus the streams are processed across multiple concurrent threads, removing I/O blocking and possibility of deadlocks within a share nothing architectural approach [30]. Additionally it is difficult to scale stateful tasks while retaining any intermediate results which have already been generated and stored within this state, given that state must be shared while also processing the stream.

These approaches attempt to increase throughput during bursty periods but DAG reconfiguration and traffic reshaping can take time which ultimately allows the application to scale. Thus matching a bursty data stream message delivery rate. During this adaptation processed messages can be delayed or lost until the application reaches a new steady state. Further adaptations may be necessary if the stream input rate continues to grow, perhaps resulting in further late messages during the transition periods.

System S, later re-branded to IBM InfoStream [22, 61], uses a similar approach with task and pipeline parallelism, where tasks and pipelines are scaled in a similar manner to Borealis and Flux. Tasks are scaled by migration actions within the same distributed architecture. Pipeline parallelism entails scaling multiple operators at the same time, including data flow redirection and duplications. In System S they also leverage data parallelism, such that duplicate data can flow down parallel pipeline routes. Here the duplicate operators in the parallel pipelines process different parts of the stream. Using a compiler and run-time system, duplicate data is ensured to exit parallel regions in the same causal order data arrived into the parallel region. More importantly this work also applied to stateful operators where the compiler analyses application code to determine which sub-graphs in the DAG can be parallelised and the system enforces this using round-robin or hashed sequence numbers at runtime to coordinate message ordering. Consequentially, this can delay the output from a stream merge while waiting for ordered results from one stream which moves at a faster pace than the other parallel stream being merged with [105, 125].

In large scale infrastructures, the application can be distributed locally amongst a set of local threads [105] allowing for parallel execution of PEs, or it can be distributed amongst a set of network connected nodes. Some recent work has deployed stream workloads dynamically to cloud infrastructures which we cover in section 2.6.

Balkesen et al. [21] suggest that data streams should be partitioned but parallel executing pipelines can be unrealistic as it introduces duplicate data volume flows which later have to be merged into a single operator or task to produce a single set of results. This phase can introduce its own set of bottlenecks. They propose an admission control and stream management mechanism for input data streams which splits workloads amongst individual nodes in their system using a bin packing approach. Other works suggest this also [20, 124]. By minimising the cost of stream redirects with the aid of of meta-data and forecasting, they can estimate the peak input data rates for known data streams which have occurred before and thus anticipate the required infrastructure to process the streams at higher data rates. Streams are then associated with the appropriately resourced PEs for processing.

2.5.2 Task Migration

Another approach to handle overload bursty data streams is to migrate tasks from nodes which are overburdened to nodes which have more resources or are not as burdened, processing easier workloads. SPEs generally are multi-node, multi-application systems in which a DAG or processing pipeline, shares it's resources with other DAGs. In such a case, a logical DAG in it's entirety, deployed on a single physical node can overburden the node. The noise neighbour problem can also arise in such a situation [58]. When a single task within a local node becomes overburdened processing messages from a bursty stream and as such context switching within the application is not scheduling enough processor time for other tasks processing other streams in other applications.

There are two approaches to solve this in the literature, first move the overloaded operators to a new node which has free processor cycles or move all other operators which are not overburdened. Stormy, a distributed multi-tenant streaming service designed to run on cloud infrastructure [84] (we cover cloud techniques in the next section), implements load balancing such that small stream tasks can be moved between physical nodes to uniformly distribute load in the system. Using a Distributed Hash Tables (DHT), allows for wide spread data distribution by using a key based hashing mechanism using consistent hashing which produces hashes in a ring topology. The DHT is used to distribute streaming queries across all nodes uniformly. When a node becomes overload, it's area of responsibility in the ring network reduces and it hands this portion of the ring hash to an under loaded neighbour in the DHT. The DHT is then updated using a Gossip Protocol, broadcasting the change to all other neighbours.

As such Stormy is decentralised with no central point orchestrating task migrations. However, it can take some time to update the DHT and migrate any existing tasks. In the meantime however, the application struggles to produce results on time and may even loose data as buffers overflow during the migration and DHT update process.

StreamCloud [54] on the other hand works in a private cloud setting using resources that are available in telecommunications operators to allow for provisioning of new instances very quickly, stated in a matter of milliseconds. This is facilitated by the use of hot stand by nodes which are already deployed and configured to accept data streams but are in a standby state for the current application. As system load increases past predetermined thresholds, the input stream or streams are directed to the standby nodes to consume input data. These new nodes, if not needed can then be reclaimed by other applications to share their resources quickly to scale these applications [54, 55]. In StreamCloud stream queries operating on the data stream are split into sub-queries and are then distributed to different physical nodes within the same DAG. This removes the need to reroute network traffic given the data steam is already routed to this physical node, thus saving valuable time in the migration process. In some cases however, data stream rerouteing may occur when the operator is migrated to a physical node not in the existing DAG. These new sub-queries are moved to underutilised nodes. Moving stateless operators has minimal overhead. However, queries are SQL like and "light" but when traffic redirection is required, routing the existing data stream to new StreamCloud Instances can take some time. Statefull operators on the other hand require splitting sliding windows and as a result may lead to the old instances and new instances processing the same set of messages for a time, until the stream moves forward to a new point in time when the duplicate message has passed.

UniMiCo [98] builds on the work in StreamCloud but takes a different approach which moves tasks in a cloud setting but without the need to migrate task state in stateful tasks. Additionally, they can support multi-task migrations rather than just a single task migration in StreamCloud. This is achieved using a Window Recreation Protocol (WRP), where UniMiCo migrates the current stream tasks at the window boundary. The *originating* task continues processing until it completes the last in progress window, while the *target* task starts processing from the first message in the next window. As such, two consecutive sliding windows overlap and messages are processed in both tasks until the end of the window. State is reconstructed in the target task without the need to migrate state from one to the another. After the migration, when all window state has been reconstructed, the *originator* tasks stop processing the stream. Given the concurrent processing of the operator, no noticeable delay in operator response time is mentioned in their evaluation.

Heinze et al. [60] argues that even with such optimisations, operator migrations still take time and are not able to provide latency guarantees defined by a service level agreement (SLA). As such they suggest a model which estimates the latency spike created by a set of task movements in the application within an elastic SPE. With this estimation they create a latency-aware elastic operator placement algorithm, which minimises the number of latency violations during task migration. By preemptively moving the task before it missed application result deadlines.

Gkatzikis et al. [51] argues a similar point. Rather than waiting for a task to become overloaded for a time, before migrating a task migration costs should be estimated and used to predicted future migration costs. Thus task migration can be performed preemptively. In their work, tasks on mobile devices are migrated to the backend server infrastructure when overload in the local devices occurs. This obviously incurs a large migration cost and traffic redirection but that cost is pre-emptively considered before the migration and migration occurs before the effects of overload are felt in the output results of the application.

Although task migration has provided means to move tasks from one burdened physical node to another, less burdened node, it assumes resources are available within the existing SPE. Another scaling technique assumes this is not the case such that more processing resources need to be added to the SPE to allow to process higher rate data streams. Thus elastic scaling techniques scale the resouces of the application usually in a cloud computing setting where resources are usually available when requested [67, 84]. We discuss this technique next.

2.6 Elastic Scaling

Fixed size distributed systems cannot scale past their processing environment capabilities. However adding more nodes at runtime and migrating operators to newly available resources can extend its processing capability and thus scale with higher input data streams. When done manually, deploying physical hardware and configuration can take considerable time. As such automation approaches to scale stream operators elastically has been suggested.

2.6.1 Elastic Tasks

Elastic tasks [105] address deployment in multi-processor systems such as the IBM InfoStream a generation 3 SPE which scales up and down at runtime as a consequence of higher input data rates. These tasks are a manifestation of an embarrassing parallel computation ideally suited to algorithms that are stateless and based in pure function. Data is partitioned to ensure state is not shared between tasks.

Stela [122], a stream computing framework built on top of Twitter Storm a generation 3 SPE, which supports scale-out and scale-in operations in an on-demand manner. The SPE aims to optimise the post-scaling throughput and minimises interruption to the ongoing computation while the scaling operation is being carried out. Stela asses the impact of scaling an overloaded task towards the application overall throughput by using an Effective Throughput Percentage (ETP) metric. With ETP the effect of scaling a single task can be measured as a ratio of it's throughput against the applications total throughput. Tasks, with the highest ETP which are congested are selected for scaling. When one is selected, this is usually the most congested task in the DAG, it is then parallelised. The stream is split and data is processed within two round-robined streams. This work supplements the default Twitter Storm Scheduler. However convergence time of the new scaled networks is shown as being up to 20 seconds after the overload occurs using a Yahoo! benchmark, a long period of time during a bursty data event in which data can be late or lost. Additionally, the scale out is based on a central scheduler which can take some time to compute new schedules. Finally, this work assumes all tasks are stateless and do not propose a solution for stateful tasks.

Much of the elastic scaling works assume a certain class of stateless task which in the case of aggregation, stream mergers and stateful tasks, are not applicable. Stream mergers must ensure that message causality in the stream is ensured and thus may have to retain messages for a period of time before they can be released, while waiting for other messages to arrive. Some solutions implement ordering identifiers with timestamps which were applied before the stream was split. Thus in two streams being merged into one, ordering constraints can delay an output stream when one input stream is faster than the other [26].

The Stream Processing Language (SPL) specification [50] was used to address the distribution of elastic stateful operators in a distributed and parallel environment. At compile time using a custom compiler, regions in the applications can be found which can be replicated later at run-time and thus apply data partitioning techniques to scale burdened operators. Operators are then distributed and controlled by the runtime environment which will only update part of the state associated with a particular operator and its particular associate state on a message by message basis as the operator consumes new messages. This is given the highest level of granularity when sharing state though the process can be slow given such level of granularity. Results are then merged at a merge operation later in the pipeline.

However work in Balkesen et al. [21] suggests that this might not scale well given the restricted physical environment and the subsequent merge operation needed later after a stream split. Additionally Balkesen et al. suggest this work has been tailored to stream operators with algebraic like operations and ignores more complex UDFs and tasks.

In stateful tasks other problem arise, as state between now duplicated tasks must be shared. Stateful operators in this work are limited by the granularity of synchronisation that can be applied to a thread in the IBM System S and Spade Declarative Language (SDL), a custom SPL. Blocking at this level may not yield the benefits of multi-processor architecture assuming waiting times around locked resources. Conversely, data divided into duplicate isolated streams may disregard any ordering that is required later in the DAG. Subsequently, this may effect downstream operator assumptions on data ordering, hence this work assuming this is not important. The application DAG maybe deployed to a distributed environment but elasticity scaling stateful operators is restricted to a single node to allow for locally thread locking structures to share state.

2.6.2 Cloud Scaling

Rather than elastically scaling operators which has a finite limit within the resources of the current processing environment, other works have pushed for the use of cloud resources, i.e., renting Virtual Machines (VM) on a “pay-per-use” model [76]. Virtual Machines are a sandboxed virtualised computer environment used to better utilise physical hardware such that a single physical machine could host many VMs, where each VM appears like a physical computer machine. Examples include "Elastic Stream" which improves System S by deploying the SPE on top of Amazon EC2. Existing parallel pipeline approaches are used but when resources are limited new Virtual Machines (VMs) are requested from the cloud provider. After their quick deployment operators and UDFs are deployed or migrated to the new nodes and the application can scale processing rates further [67].

To address highly dynamic stream computing environments, other approaches suggest elastic execution models which are required to scale into the 10,000 queries per second processing 100,000 messages per second [82]. This is achieved through high levels of parallelisation of operators executing continuous queries on Linked Stream Data (LSD). LSD is a stream based processing model for Resource Document Framework (RDF) data, which in this work describes semantic sensor webs [81].

How additional nodes are added to the existing SPE resource pool is answered in the literature [84].Stormy as mentioned previously uses a DHT to locate tasks and their areas of responsibility in the ring network. However, Stormy also has the capacity to add extra nodes to the SPE based on the decisions of the SPE leader node [25]. A single leader is responsible for this decision to ensure that just enough nodes are requested. This is to control how many nodes are requested from the cloud provider and to prevent more than the needed amount of nodes from being requested. Tasks are migrated after node deployment. This can present some small delays as DHTs are updated on multiple nodes after task migration. Additionally, deployment of nodes can take some time after they are required from the cloud provider. Conversely, work in LSD uses a central coordinator service which maintains query execution locality and uniform distribution of queries across nodes which join and leave the network at runtime. Leader election and state replication of this are considered such that the leader does not become the bottleneck in the large network [82].

In SteamCloud a generation 3 SPE, a centralised traditional model is used to monitor local instance utilisation and to request new VMs from the cloud provider, in this case Amazon EC2 [55]. As well as deploying new cloud instances controlled by an Elastic Manager (EM) interacting with a Resource Manager (RM), traffic load balancing is also pragmatically controlled by the EM. The EM receives periodic updates from each local manager within each StreamCloud Instance and when overloaded requests new instances. After this duplicate tasks are deployed to the new instances and traffic is rerouted to provide the additional processing capacity needed to process the stream.

Drizzle which was evaluated atop Amazon EC2 attempts to optimise the cloud system by decoupling message processing events from SPE reconfiguration events. They argue that message processing events must take place in relatively small millisecond periods of time but reconfiguration events can take seconds or even minutes without effecting the data processing rate [120]. This work was evaluated on 128 r3.xlarge instances in Amazon EC2. However they have noted that during reconfiguration the data rate of the system drops at first as tasks are migrated from one instance to the next. In their evaluation they add 64 machines to an existing 64 machine cluster and note that reconfiguration took between 10 to 100 seconds. However, detail is lacking as to whether the machines were on hot-standby, requested from a cold boot and how the framework was deployed and configured after being requested.

As abundant as cloud resources appear to be, after requests are made it can take considerable time to receive ready nodes, deploy operators or tasks and redirect traffic during overload periods. Additional unlike fixed size distributed systems where costs are known, scaling cloud resources has additional costs to consider.

2.6.3 Elastic Scaling Costs

Moving into the cloud has additional costs and scaling up nodes which are charged at a per minute or hour rate of use, can have significant monetary costs. As such this has been treated as an optimisation problem optimising for cost when weighted against application output latency during overload situations [67]. Work in T-Storm [121] built on top of Twitter Storm, which implemented a custom scheduler which moves executing bolts (tasks), to other executors (nodes). Thus freeing up the executors and allowing other free executors (with no executing tasks) to be turned off when not utilised to save money. The standard Storm scheduler distributes bolts across all available executors to uniformly distribute load in the system.

Subsequently, redistribution of an existing topology (DAG) is required when new nodes are added, however this is complicated by having to stop the topology, redistribute it and re-start the topology again. This is detrimental, to state in stateful operators as well as temporarily stopping stream processing [42, 80].

StreamCloud as mentioned earlier also considers costs in it's scaling strategy with the centralised Elastic Manager (EM) getting updates from local managers within each StreamCloud instance. When an instance is measured and determined to be under utilised, determined by a predetermined CPU utilisation threshold, and resources are available within existing instances somewhere else in the cluster, the underutilised instance is turned off. This happens after tasks have been migrated and any state has been reconstructed in these new tasks. State is reconstructed using a Window Recreation Protocol (WRP) as mentioned earlier. Additionally, EM has access to the application load balancers and through the Resource Manager (RM) redirects traffic to the appropriate instance during task migration [55]. After migration, and state is reconstructed the task begins processing the redirected stream.

Given the additional costs that have been noted and that requesting resources from a cloud provider can take some time, others have attempted to scale applications during overload using containerisation technologies, which we review next.

2.6.4 Containerisation

Containerisation, a lighter executing environment than VMs have been chosen as an alternative to requested VMs from a centralised cloud provider. Containers can be deployed more quickly than VMs. Containers are sandboxed from one another through higher application abstractions and share the same kernel as the host operating system to help reduce virtualisation overhead. Additionally containers tend to be single threaded (though they are not limited to a single thread) and the application executes inside the container under this single thread. Horizontal scaling approaches are used to scale application further by deploying more containers.

Generally the underlying computer resources are deployed on a set of large physical servers in a cluster and orchestrated using a cluster management system such as Apache Aurora, or Docker Swarm [80]. This allows for an application to request new containers from the cluster management system relatively quickly and redistribute traffic very quickly, rather than having to wait for VMs to be provisioned from the cloud provider.

Twitter Heron [80] a generation 4 SEP and improved architecture upon Twitter Storm, executes within a container environment on top of Apache Aurora. Confusingly, a Heron Container executes within a container environment consisting of a number of Heron Instances. The ratio of Heron Instances to Heron Container is based on available resources in the host container as well as application requirements and is determined at deployment time. Each instance consists of two threads, one for a gateway which orchestrates message flow into an out of the instance and another thread with executes application code (task). The two threads communicate between one another with three unidirectional queues. Within the *Data In* queue messages flow from gateway to task. With the *Data Out* queue and *Metrics* queue messages flow from task to gateway.

Each container also contains a local Stream Manager to monitor stream distribution and monitor instance overload. Heron improves on the management of the full life cycle of container deployment and traffic management which Storm did not provide. As such, during application overload or during a state of anticipated overload, Heron can request new containers in which overload bolts (tasks) are deployed and subsequently process the bursty stream. State is maintained by a state manager in a large in memory cluster for faster read and write performance (such as Zookeeper, a popular high performance coordination service and key value store. Zookeeper is used in this thesis which we discuss in our implementation in section 4.3.11). Each bolt can checkpoint state through predefined APIs available to each bolt and identified by a unique id. State can be retrieved by any bolt using that checkpoint id.

Dhalion [44], an SPE built on top of Twitter Heron [80] which provides for self-regulation such that application configuration, management and deployment are all self regulated within the SPE. Through a dedicated health manager, application Service Level Objectives (SLO) are monitored within the application which attempts to avoid incursion of the SLOs such as application latency violations or message loss. This is achieved by dynamically deploying more bolts and redirecting the stream to these bolts to scale processing rates.

In this work SLOs, such as higher than allowed message ingestion times or data loss are not tolerable for certain classes of applications. Thus they avoid discarding data in the data stream (cannot not use load shedding). This work was implemented given that the default scheduler in Heron did not monitor the SPE in fast enough cycles to prevent latency violations and thus to implement adaptive behaviour quickly enough when bolts become overloaded.

2.7 Task Modification Approaches

In more recent works a task selection technique has been proposed which selects different implementation of tasks at runtime during bursty events to ensure output result accuracy [100]. This work selects different algorithms or different implementations of algorithms at runtime. Such that the functionality of the algorithms are the same but alternatives trade output result accuracy for result timeliness and device resource usage during bursty events. This work is scoped primarily in IOT devices. This work reduces the algorithm selection problem to a Multi-choice Multi-dimensional Knapsack Problem. In this multiple algorithms (each implementation produces the same output result but has a different implementation) contained within a task t_i exhibits different resource characteristics r_{cpu}, r_{mem} (algorithm cpu and algorithm memory per processed message) and also a base processing duration r_t (processing duration or latency per message). At runtime the choice of selecting an algorithm implementation comes down to resource availability but optimising for base processing duration.

It should be noted this approach bears close resemblance to the philosophy of the approach suggested in this thesis. However, their work relates to entire algorithm replacement rather than gradeful degrading the current algorithm at runtime (removing parts of the application at a proportional rate to the perceived overload) as messages are received. For example, their suggestions include replacing security hash functions such as AES-128 which processed a hash in 62 ms, exchanged for Blowfish which took 44 ms to hash the same input value. Further suggestions include exchanging image compression libraries such as JPEG which take 240 to compress and convert an image. The implementation is exchanged for JPEG2000 which took 163 ms to convert the same image during benchmark tests. Additionally they suggest that this work is the first approach in the literature which applies dynamic algorithm selection to stream computing problems during bursty data stream events.

2.8 Literature Analysis

Distributed and parallel computing have been designed and implemented to overcome the problem of bursty data streams on deployed infrastructures by scaling up the applications processing rate at runtime. Some approaches attempt to restructure existing computations using load balancing, task migration and data partitioning techniques. Others migrate to the cloud to acquire more resources elastically, taking advantage of the relatively low cost and more importantly readily available resources. As we have seen from other works, the assumption that scaling can happen quickly is not always guaranteed. As such, application latency violations and perhaps data loss can ensue while waiting for new resources [54]. Containerisation attempts to facilitate faster adaptation of the overload operators and the subsequent redirection of the stream, although the performance degradation of such adaptation is observed it is reduced when comparing to cloud VM requests.

Furthermore, scaling a stream application is not as simple as adding new nodes to consume new input messages at higher data rates. The data stream may need to be partitioned at runtime to distribute it to any new nodes which have been deployed or duplicated thus requiring traffic shaping techniques. However later in the stream pipeline, the result of the stream splits will require a stream merge to reduce the stream back into a single stream once again. That particular point in the processing pipeline can become the bottleneck next. A stream merge adds additional complexity in this regard as messages tend to be casually ordered, as such the merged stream may need to keep this ordering requirement, a complex task to achieve in a distributed environment which has been shown to delay the stream.

Many of the these techniques assume applications do not maintain any application state, at least not beyond a stream window. Application state is transferred in it's entirety between one point in a system to another, in much the same way REST provides for [43]. However this tends to be communicate via a message passing approach [94] Improvements have been made in sliding window migration however, this work assumes windows are short-lived and reconstruct windows at their boundary.

Gulisano et al.[55] argue that much of the work presented in evaluating stream computing scaling approaches are quite limited. In most cases, evaluations are using limited application scope i.e.: evaluating stream splits while ignoring the subsequent stream merger later in the pipeline. Gulisano et al. continue to argue that these evaluations inadequately describe real world applications or the complexities they inherent. However, the theoretical contributions of this work are appropriate and valid. Further, they argue that the complexity of the evaluated applications should be more akin to real world applications which they assert are not. Further, they suggest real world applications would not only enhance understanding around the complexities of stream computing applications in the world at large but also help devise appropriate solutions. This has started to emerge with work on Heron and Storm where streaming applications used in industry however, more work is required [80].

When comparing scaling approaches to load shedding approaches, load shedding does not require any additional resources, does not incur any additional monetary costs and has quick execution. However it's scale is limited and load shedding work in single host systems has been scoped to short-lived bursts. In distributed stream computing systems, the scope has been increased given the greater range of places data shedding can take place. As such the shedding of data as a result of overload can occur shortly after detection. However, a balance between application QoS requirements and reduced latency is needed and depending on application requirements, existing works have optimised for one or the other.

Task and operator scaling on the other hand provide for a longer term processing position given that the application can scale with the increased data rates contained within the bursty data stream. Most solutions do mention that it can take time for the migration, or application to acquire additional resources and that has been minimised with the advent of containerisation, partly because resources are already provisioned but not utilised. Though therein lies the cost of cloud and container offerings with large amounts of resources being provisioned on demand to it by directly interacting with the cloud provider in an automated way during overload.

Further, scaling tasks is not without it's complexities. Described solutions address this with stream splits and stream mergers within stateless tasks. Stateful tasks require more sophisticated techniques to ensure during the split and later merge phase, results are ordered causally and not duplicated. Most solutions assume stateful tasks maintain short lived windows in either time or element count and use this contract to migrate or reconstruct state at the window boundary. An approach to scaling longer term sliding windows is left as open questions in the literature.

Newer works, have also suggested task replacement which follows the same physiology as suggested in this thesis. In applicable cases, algorithms can be exchanges at runtime to optimise for a particular criteria in this case, cpu time and result timeliness. However, approach assumes an entire algorithm replacement rather than a graceful degradation (removing parts of the application at a proportional rate to the perceived overload while still producing an application result).

Finally, Rivetti et al. [101] leaves an open suggestion in their paper that load shedding and scaling approaches could live side by side rather than being alternatives and thus provide a complete comprehensive approach to handling bursty data streams.

2.9 Chapter Summary

In this chapter, we provide background into stream computing and discuss streaming data models as well as stream operators, User Defined Functions and tasks. We also outline the bursty data stream problem in more depth.

Next we move to the literature review, describing how the community has attempted to solve the overload problem during bursty data stream event. The literature divides this into two main approaches, data based approaches and task based approaches.

Data based approaches attempt to reduce overload by reducing the workload of the application. This is achieved through load shedding techniques or synthesis structures. Much work has been done in ensuring that the output result of the application are accurate in the face of lost data, given the inherit trade-off in discarding data.

Task based approaches address the application, scaling the application and it's internal operators, UDFs and tasks in the face of overload. Load distribution techniques are used to distribute the load then by splitting the stream allow concurrent operators to consume it. Other works address how to scale the application more quickly, using operator migration techniques as well as scaling into the cloud and more recently into containers. As much as convergence times of the scaled system have reduced delays, performance degradation are still noted. However, many approaches neglect long-lived statefull operators and during migration or scaling activities, performance tends to be degraded until the application enters a steady state once more at the higher resource level.

The approaches described provide a means for adaptation at runtime to the bursty data stream problem in stream computing domains. Each of the approaches has its own merits and drawbacks. Although, no single solution provides for the combined benefits of quick adaptation without data loss or the effects of stream bursts on output result latency. Scaling techniques can provide a longer term solution when the application and infrastructure have reached the new steady state with additional computational. With this in mind the use of imprecise computation in stream computing is the focus of our investigation.

Chapter 3

CS: Computational Shedding

3.1 Overview

Stream computing has been adopted in domains as a result of two main requirements, the output of low latency results and reduced data storage requirements. Streaming applications have been used to process data from devices such as mobile and tablet devices, in addition to data generated from embedded sensors and devices classified within the Internet Of Things Phenomenon (IOT) [90, 112]. In stream applications, the push based nature of the data ingestion model in combination with data generate from these sources, can lead to system overload as input data rates grow beyond the processing capacity of the application's provisioned infrastructure and processing resources. The very nature of these devices and domains can lead to sporadic data generation rates during uncommon events.

Our approach, which we call Computational Shedding (CS), is designed to tackle this overload problem in a different way than existing approaches in stream computing applications. CS facilitates the graceful degradation of the runtime applications to produce approximate output results by reducing the cost of process messages in the data stream. This is achieved by trading application output accuracy for higher message throughput rates, facilitated by the reduced cost to process each message in the stream at the time of overload. This is achieved without the need to discard input data unlike other existing approaches.

This design chapter contains a detailed justification for our computational shedding approach. Firstly, a discussion of the rational for CS is presented in section 3.2. Section 3.3 describes our approach to runtime adaptation during overload. Section 3.4 describes the design elements of CS, in addition to describing how this adaptation model fits into a typical stream computing application.

Four version of CS have been designed, each with a different enhancement which are also contained in this chapter. An example of how each operates and description of any pros or cons with each version is also supplied. Computational Shedding - Duration Focused (CS-D) was designed to evaluate the approach of CS and determine it's viability with a priority focus on message delivery times, it can be found in section 3.5. Computational Shedding - Accuracy Focused (CS-A) takes a similar approach but different focus, setting application output accuracy as the priority while also attempting to honour application messages deadlines, it can be found in section 3.6. Computational Shedding - Cost Focused (CS-C) attempts to reduce the cost of executing CS upon each message in a data stream by reducing the cost of determining current application load and future work, it is outlined in section 3.7. Computational Shedding - Binary Search (CS-B) further attempts to improve the cost of CS executing on the stream by attempting to reduce the cost of selecting an appropriate set of subtasks to process messages with using a Binary Search Algorithm. It's discussion can be found in section 3.8.

This chapter concludes with a summary in section 3.9. In the next chapter 4 the implementation details of CS and the framework, Simple Stream Computing Framework (SSCF) developed to evaluate it are discussed.

3.2 Rationale

Application domains for stream computing, such as real-time web search and analytics, health care monitoring, road network traffic management, live financial stock tickers and combat battlefields, have experienced the benefits of low latency data processing after the change to a push based input data model[10, 22, 94].

However, given the sources of their data, the accompanying data streams have been shown to produce randomly occurring bursty data streams as a result of real world phenomenon. For example, consider a car crash during peak congestion time in a managed traffic road network, a system which indicates to road users real time route information. Suppose the system becomes overloaded due to the crash event, the result of which manifests in delayed instruction to drivers. Another example live web based searches surge after a popular news event occurs where people wish to find more information. Or perhaps more importantly real-time patient monitoring in health care where an event occurs in a patients body and sensors light up with new data points. These events as with any bursty data stream, cannot be planned for, in occurrence, in volume and/or in duration.

As such, the appropriate amount of resources to provision prior to any bursty data stream and for an application to continue to process this stream in a timely fashion, is difficult. There are two high-level solutions to the CPU overload problem in this case: (i) increase the CPU capacity to a level that is higher than the demand, or (ii) decrease the demand below the available CPU capacity. The former option may not be a feasible solution due to several reasons: as described in chapter 2, numerous works in the area of bursty data stream handling strategies have emerged in stream computing which attempt to scale up the application during bursty data loads. Some works in this area have shown that this can take considerable time in the case of cloud computing, when in fact the problem requires an immediate solution. Others apply hot-standby solutions at considerable cost but given the nature of the bursty event, the streams velocity may increase further thus requiring more resources than were provisioned for. Finally, bursty data streams as we outline in this work are usually short lived in duration but are of high intensity, such that the CPU demand during bursts may be orders of magnitude larger than the CPU demand during regular workload. In such a case it may not be worth provisioning resources dynamically in temporary overload scenarios, such that by the time the resources come on-line the bursty has already ended.

To decrease the demand on the CPU to below available levels requires a decrease in the workload of the data stream. One area of research into Load Shedding (LS) provides an adaptation model of the data stream itself, thus avoiding any possible long delays as more processing resources come on-line. As was discussed with some of these approaches in section 2.5. LS discards input data in the stream to ensure that data that remains continues to meet it's application deadlines. Random Load Shedding (LS-R) discards data stochastically, making the assumption that all data is of equal importance. This may not be the case in many applications. Take for example the live web based searches mentioned earlier searches based on an important recent event where an indexing application is responsible for taking web page update notifications from the greater web. Suppose new indexing updates are sent based on that particular news event, indexing updates need to be received and processed as quickly as possible to provide the best results for querying users in relation to that event. These specific ‘time-bounded‘ events become intermixed with other general update events from other web sites in the input stream to the indexing application. Causing all update events to queue up waiting for indexing activities. However, if LS-R was activated on the input stream to prevent update events from becoming late without prejudice, some events would be lost as a result of shedding. In this case, the new update events may be discarded indiscriminately leading to inaccuracy and perhaps stale and/or poorer search results.

Further improvements in load shedding strategies have led to Semantic Load Shedding (LS-S) which attempts to attribute a utility value to messages in the input data stream and thus sheds messages of lower utility value first. However, in such a strategy the semantic value of each input message must be assessed and in the case where the data model is complex, classification can take time delaying the input data stream further.

In both LS-R and LS-S the amount of data that is shed is proportional to the overflow amount of the data stream such that in large bursty events larger amounts of input messages must be shed to ensure application deadlines are met. In cases of patient care, shedding messages in an input data stream may not be the ideal approach to get the most accurate picture of the patients circumstances.

Our contention with these classes of approaches are two fold:

1. First, additional application resources may not be available in sufficient time or in sufficient volume to process any evolving data stream.

2. Second, in applications which do not or cannot tolerate data loss well, load shedding approaches are not applicable or in non-trivial domains cannot classify input data sufficiently and quickly enough without consequence.

Another approach to handling bursty data streams is necessary which can provide a means to continue to balance between the input data stream velocity and the available application resources to process the data in the stream. Modifying the application at runtime such that at higher input data rates, the messages contained within the stream continue to be processed, even to a partial degree without the need to be discarded. Further this adaptation must also ensure that application deadlines continue to be honoured as this is the most important premise in streaming computing, with accuracy as the next most important premise.

As such bursty data stream management in stream computing can be viewed as an optimisation problem, mapping requested input workloads to available resources during bursty data stream events or application overload. As the application becomes overloaded or is anticipated to enter an overloaded state in the near future, our view is that the runtime application should adapt to reduce it's processing cost per message. With such an adaptation, more messages in the input data stream can be processed with the same set of application resources, continuing to meet application deadlines, but with an accuracy cost to the applications output result. Here we scope this work to computational resources, though this is not to say that memory and bandwidth are not also limiting factors in application overload scenarios.

Our approach, titled Computational Shedding (CS) provides an adaptive mechanism to handle overload situations in applicable stream computing applications by temporarily removing local tasks (subtasks) of the executing application during a bursty data event. In such a case, an approximate application result is produced as a trade-off to ensure all input messages are processed within allowed time bounds. The processing time which is reclaimed by effectively turning off selected subtasks temporarily is used to increase message processing rates, also know as ingestion rates, attempting to match higher data rates contained within the input data stream. Thus, message latency violations are avoided while ensuring each input message is processed to a certain extent. Moreover, should input data rates continue to grow further subtask removal can occur until just a single subtask remains executing in the application. When the overload period has passed, the discarded subtasks can be reallocated and a full set of subtasks can continue to produce accuracy results further.

In the following sections we will elaborate further on the rationale for our approach and describe how CS continues to produce application results during overload situations without the need for additional application resources or discarding any input data while producing timely, application results.

3.3 Approach

Our approach, Computational Shedding (CS) attempts to manage input stream overload situations by temporarily reducing computational costs in processing messages contained within input data streams. By reducing the cost to process a message, higher throughput rates should ensue. This is achieved by discarding local subtasks contained in the running application. This overload handling strategy was achieved without the need to discard input messages. However during application overload with this strategy at work, input messages are processed to an approximate level with the remaining subtasks.

Our aim during the design of CS was two-fold. First, to investigate the viability of the CS's application adaptation approach in a stream computing domain. By this, an understanding is needed in how to construct an application in such a fashion, as well as understanding as to when to activate CS and to what degree subtasks should be shed. For instance, in an application containing many subtasks, a selection process is necessary to determine what subtasks to keep and which to discard, proportional to the current application load. Secondly, for application results that are generated by a subset of subtasks, the aim was to ensure the subtasks which were selected to remain in the application, produced accurate results within acceptable application time bounds.

We assume distributed stream computing applications to be the normal processing platform for stream computing applications rather than the exception and in such a case overload effects tend to be localised to a single node or container, not necessarily all containers in the application.

In this work, a container as outlined in section 4.3 with further details of the Simple Stream Computing Framework (SSCF) in which CS executes, is an association of local components which process the data stream. This include a Load Handler (LH), to handle local data stream load, as well as two message queues for temporarily holding input messages, an adapter to interface with the world outside of the container, and finally a Processing Element (PE) (a component where subtasks are instantiated, executing business logic for the application). Load Classifier (LC) in this regard are a type of LH which implement a version of CS to make current container utilisation decisions as well as determining the selection of subtasks to compute messages in the data stream. This is outlined in more detail in section 3.4.

Further, in such an application architecture, containers may feel the effects of data stream overload at different rates given different subtasks with different processing characteristics are distributed amongst a set of containers. As such overload handling strategies must be locally executed to overcome the effects of increased data rates on each container to prevent application results becoming late.

3.3.1 Design Stages

In order to investigate our proposed CS approach and understand what design choices or optimisations where effective, CS was developed in four stages:

First, Computational Shedding - Duration Focused (CS-D) was developed to evaluate the approach and to determine the applicability of the approach to a application in the stream computing domain. CS-D contains an overload detection mechanism, as well as a pre-determined set of subtask combinations which are selected at runtime, a selection based upon the degree of overload in the local container. CS-D was optimised for reduced message processing duration. The second version, Computational Shedding - Accuracy Focused (CS-A) was developed to focus on application accuracy such that, of the subtasks selected rather than focusing primarily on message processing duration within time bounds, the resultant application output would be also accurate. The third version, Computational Shedding - Cost Focused (CS-C) attempts to reduce the cost of subtasks selection at runtime, attempting to prevent further delay in appropriate subtask selections. The final version, Computational Shedding - Binary Search (CS-B) further improves upon Computational Shedding - Cost Focused (CS-C) by reducing the computational complexity in searching for an appropriate subtask combination by using a Binary Search implementation [33].

Our expectation with the four versions of CS and resultant implementations is outlined in figure 3.1 and figure 3.2. Our evaluation comprises of the four versions of CS against one another, in addition to both LS-R and LS-S within the remit of a simulated search application, as outlined in section 4.2. In figure 3.1 the expected application accuracy graph is shown. This figure represents the expected output of each overload handling strategy in terms of application inaccuracy, such that lower values are better. As the input data rates increase, each strategy attempts to adapt the input data stream going into the application or adapts the application itself. All strategies here sacrifice accuracy in the applications output result to ensure message processing deadlines are met, a trade-off given that output application timeliness is deemed more important. As such, we expect as input data rates increase in the input data streams, the output result of any application will become more and more inaccurate. It is expected that CS versions will produce results which are more accurate than either LS approaches in most cases. CS approaches process all messages in the stream. Where as with LS, as application overload increases message are discarded in the stream thus they are simply not there to process.

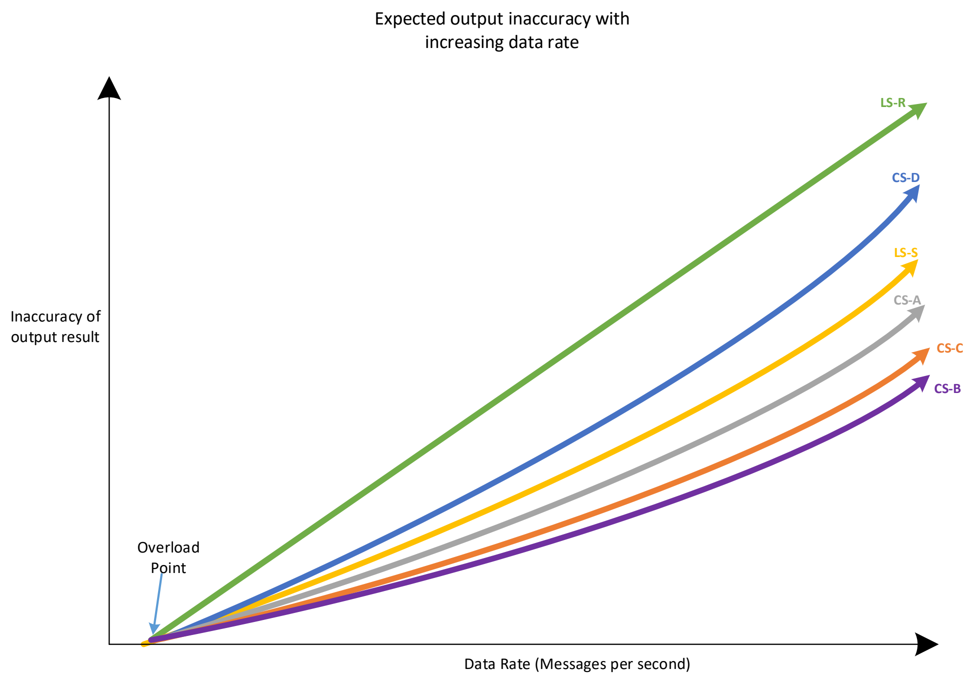

Fig. 3.1: The expected output inaccuracy result trends of each version of CS designed, plotted with load shedding alternatives. It is expected that with each strategy as input data rates increase, higher and higher levels of inaccuracy in the application results will ensue. Given that CS approaches process all data in the stream and thus do not discard any messages, this will lead to a more accuracy result. Whereas LS approaches may have simply discarded the messages during overload.

Further we expect CS versions to process more messages within the same time-frame than LS counterparts as a result of CS focusing on reducing the processing cost of each message in the stream. Note, we expect LS-S to preform well in this regard given it's grouping strategy and as such can discard groups of messages at a time rather than just a single message. Figure 3.2 illustrates our expectation of message throughput within each overload handling strategy. Here, to ensure application timeliness, each approach effectively increases message throughput thus ensuing deadlines are met. The overload point illustrated within each graph represents the point which a stream application becomes overloaded with data and must trigger some form of adaptation to continue processing in a controller manner.

To summarise, CS was developed in four stages. First, CS-D was developed purely focused

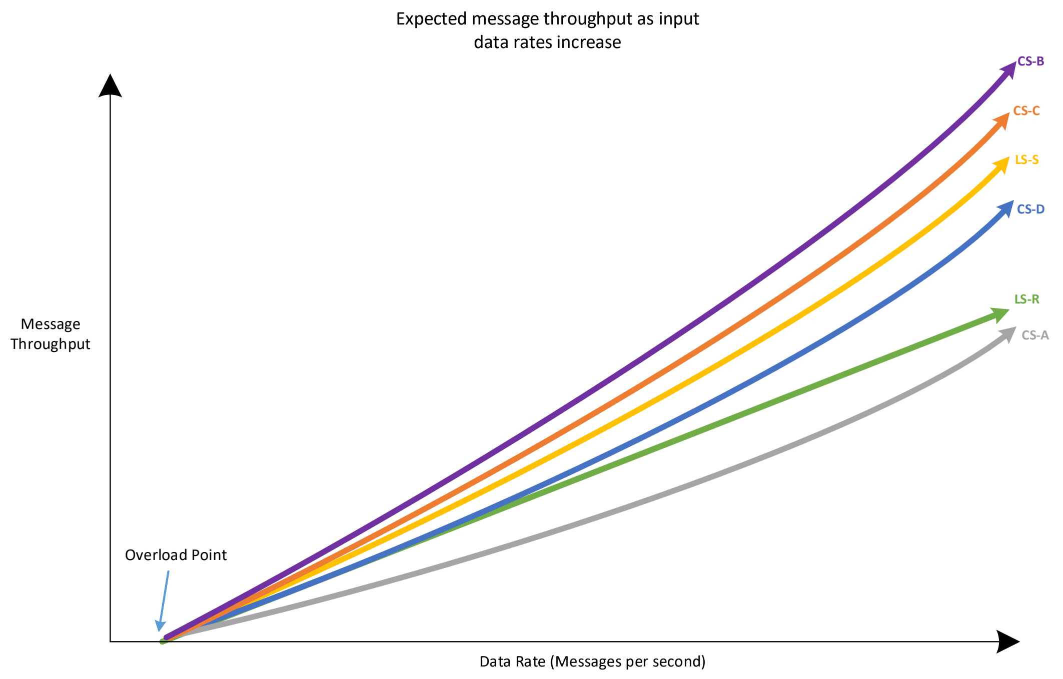

Fig. 3.2: The expected output throughput trends of each version of CS designed, plotted with load shedding alternatives. It is expected that with each CS version as input data rates increase consequentially causing adaptations in the application by shedding subtasks, higher data throughput rates will ensue.

on reducing the computational cost of a message and validating the CS approach. Second CS-A was designed to improve upon the output accuracy of the result which was not the primary focus of CS-D. CS-C was developed to improve upon prior designs by attempting to reduce the cost of making a subset subtask selection during bursty data events. Finally, CS-B was designed to improve upon CS-C to further reduce the selection time of subtask combinations using a Binary Search Algorithm.

The following sections will discuss these designs in greater detail.

3.4 CS - Core Design

The four versions of CS share a number of similar design choices; this section will describe these common elements in more detail. Section 3.4.1 will discuss the partial computation approach used by CS where individual jobs or subtasks of an application are removed to produce a partial result on time. A subtask can be defined as a discrete part of an application such that it can produce a result without direct involvement of another subtask. That is not to say that a subtask output cannot be an input for a subsequent subtask later but in that sense both subtasks must be disabled. Section 3.4.2 describes the Subtask Execution Plan (SEP) a pre-computed permutation of subtask combinations created prior to application start where entries are selected at runtime. Section 3.4.5 describes the need for subtask profiling to obtain a measure and cost of subtask execution within a particular application and data set. Finally, section 3.4.6 outlines how overload detection was achieved in this work and how entries within the SEP were selected.

3.4.1 Load Handling Approach

The core function of a load handling strategy in stream computing is to facilitate controlled handling of input messages in a data stream when the input data stream becomes bursty. We define a data stream and the messages contained within as follows:

A data stream and the contained elements can be defined as a causal sequence of data elements $e_1, e_2, e_3, ...$ that arrive one at a time, where each element e_i can be viewed as $e_i = (t_i, P_i)$ where t_i is the time stamp associated with the element, and $P_i = (p_1, p_2, p_3, ...)$ is the payload properties of the element.

When a stream becomes bursty, we are referring to the received volume of messages within the stream exceeds what is expected. In a stream application this has the effect of increasing message end-to-end times, given that message processing rates tend to remain the same, thus causing message queuing times to increase. As such the end-to-end processing time or ingestion time increases and can exceed the applications ingestion time requirements. The purpose of a load handling strategy in stream computing is to ensure that messages are processed within expected time bounds such that application results are not late.

CS takes inspiration from the Imprecise Computation (IC) domain where, if an output result from an embedded device cannot meet it's scheduling deadline, optional jobs or tasks within the device are temporarily turned off to ensure the result meets it's deadline[83, 108, 109]. Within the Imprecise Computation (IC) domain it is better to produce a partial application result delivered on time than a complete result delivered late. Stream Computing takes a similar philosophy.

Applications using CS must be constructed in a similar fashion as required of embedded jobs within IC. An application must consist of a set of individual jobs or subtasks for which there are both mandatory and optional subtasks. Subtasks execute in sequential order such that one subtask's output intermediate might be the input to the next subtask in the processing pipeline. As such there may be an ordering requirement between subtasks. Unlike IC, mandatory subtasks are not required to execute first in the pipeline but are required to complete message processing before the associated results become late. In the case of a multi-container application, subtasks may execute across multiple containers, provided the processing pipeline order is honoured.

The CS approach was chosen for a number of advantageous reasons and applied within CS. First, although an input message may be only partially processed during an overload event, messages are processed to some degree rather than being discarded. The reclaimed processor cycles from subtasks which have been effectively disabled temporarily, aid in scaling the message throughput rate in the application. Second, in the case of a stateful application (one which maintains an aggregate set of messages or results after it has been computed) can benefit from the CS approach as when the overload period has passed the stateful results have an opportunity to be processed again with the full set of subtasks. Third, given that the adaptation of the application happens without the need for external resources and the the detection of overload occurs locally, the impending adaptation can occur quickly, thus ensuring the application adapts as fast as possible to ensure timely result delivery.

However, a number of drawbacks to this approach must be mentioned. An application must be constructed such that it consists of a number of individual subtasks with the removal of one or more subtasks, allowing the application to continue to produce a result. This curtails the general applicability to certain classes of applications which fit into this model. Further, if the velocity of data during the bursty is so great as to cause CS to remove all optional subtasks such that this adaptation is not enough to stave off the new input rate, no further adaptation is available in the application as only mandatory subtasks remain. If this should occur, message throughput cannot be increased further within this approach alone, messages may begin to queue up within the internal queues which pass a certain threshold can lead to message output results becoming late. However, as mentioned previously, this work is scoped to short-lived intense bursts such that between the adaptations available in the application and the size of the message queues, there is enough capability to process messages on time, even at the cost of the result produced being to the lowest degree of accuracy available.

In the next section, the Subtask Execution Plan (SEP) used in CS is explained further.

3.4.2 SEP : Subtask Execution Plan

To optimise the selection of subtasks at runtime, subtask combinations are precomputed in CS. The SEP is a table of pre-computed optional and mandatory subtask combinations where each entry contains unique permutations of subtasks taken from all available to the application. This approach was taken to help reduce the cost of determining optional subtask selections at runtime. In an application and system which are already overloaded, any activities which can be computed off-line should be considered to best reclaim any processor cycles which can be used to increase message throughput. Remember in most cases of overload the CPU of containers running the application are at 100% utilisation.

To generate the SEP an understanding of the cost of processing messages within the application must be determined in both time and accuracy contribution to the applications result. This is the purpose of Subtask Profiling which we outline in section 3.4.5 later. When both values have been generated, the complete unique combination list of all optional subtasks in the application can be generated. Mandatory subtasks are added to each entry in the table given they are required to execute upon each message or set of messages in the input data stream.

During subtask profiling, the average message processing cost and average accuracy percentage contribution of each subtask is measured when computing a sample set of data used in the application. From both these metrics, a subtask execution plan is generated constructed from all permutations of all optional subtasks and mandatory subtasks available to the application. This plan is generated statically and off-line. Within each row of the plan, the total average accuracy cost of a all contained subtasks and the total estimate of the accuracy contribution of all subtasks are recorded within the plan.

The SEP table entry order can define a priority order in subtask combination selections. For example, suppose row 1 in the table has a higher priority over row 2 in the table because row 1 produces results which are more accurate. This is as a result of row 1 having just a single subtask more than entry 2. This is important to consider as all optional subtasks may not provide for an equal measure of message processing duration or the effect that subtask has on the accuracy of the output result, accuracy being the measure of how close an output application result is to its ideal result. Thus removing one subtask between entries could lead to more messages being processed for the same time frame with a relatively small impact on output result accuracy. Take for example entries in table 3.1 between the first entry in the table, *Subtask Plan Id 512* and *Subtask Plan Id 167*, subtask 8 has been excluded from the plan which incurred a 0.036 accuracy loss and saved on average 0.008 ms per message received. Compared to removing subtask 7 in *Subtask Plan Id 189* which leads to an accuracy loss of 26.8609% for a processing duration gain of less then the removal of subtask 6.

For any output application result which has been processed by a set of subtasks during overload, an approximate error bounds can be determined from an understanding of how much of an accuracy contribution the removed subtask brings to the application. As we will see during subtask profiling in section 3.4.5, each subtask must be profiled to understand it's message processing cost and accuracy contribution in the application. With this subtask meta-data, by removing a subtask or multiple subtasks from the application, a measure of how inaccurate the result may be can be determined. This is a further benefit of a deterministic selection of subtasks.

The SEP was generated by supplying a bit counting function and running it upon a list of optional subtasks to create a unique set of combinations, the limit of which was defined by the Power Set. The Power Set of any set is all subsets, including the empty set and the full set itself [63]. Algorithm 1 represents this algorithm in pseudo-code format. Algorithm 1 works by creating a binary string representation for an iterator up to p_{count}. Each iterator value creates a unique key to create unique combinations for all values in the Power Set, limited by the amount of optional subtasks. By representing an iterator i from 0 to the count of the Power Set combination in binary format, p_{count}, the bit counting algorithm could iterate through a list of binary representations and thus construct the unique subtask combination list.

The following equation was used to generate the total power set combination count:

$$p_{count} = 2^n \tag{3.1}$$

where p_{count} is the count of unique combinations and n is the number of optional subtasks in the application.

After p_{count} was calculated, the next step is to use the bit counting algorithm to generate the unique combinations of subtasks or SEP entries efficiently. This is best described with a worked example which is provided in section 3.4.4. Note this algorithm may not scale well given that as n becomes large, the nested loops may lead to a worst case complexity of $O(n^2)$ and as such

```
**Data:** input $p_{count}$
**Data:** let m = array of mandatory subtasks
**Result:** list of unique lists, consisting of subtask combinations
let p = a new array of arrays, for the whole plan;
let a = the list of optional subtasks;
for $i < p_{count}$ do
    let e = a new array for each subtask combination set;
    let str = convert i to a binary representation in string format;
    update str by padding it to the left with zeros, to the count of $p_{count}$;
    for $j < str.length$ do
        if $a[j] == '1'$ then
            add a[j] to e;
        end
    end
    add m to combination;
    add e to p;
end
```

Algorithm 1: SEP generation algorithm using bit counting to generate a set of unique subtask permutations.

SEP table generation may take some time. In experimentation we show that for nine optional subtasks and four mandatory subtasks, the table was generated in 5 ms on average, not including subtask profiling activities.

Table 3.1 shows the output result of Algorithm 1 when subtask profiling results have been supplied and the SEP has been given a priority order of message processing duration first. That is to say that rows in the table are ordered by decreasing message processing order. In this example SEP it is deemed that producing the results for the application quickly is more important than accuracy. The table can be ordered in any manner to facilitate application criteria and priority. In our experimental work two SEP table orders are provided, one in which accuracy output is deemed most important within CS-A. The second is what we see here were message processing duration is most important as presented in CS-B, CS-C and CS-D. The results of both are provided in chapter 5.

3.4.3 SEPs in SSCF

The view outlined in section 3.4.2 of an SEP is not optional, to facilitate subtask activation and deactivation in a SSCF containers. As subtasks are unique within an application and deployed to containers uniquely, the SEP needs to be updated to be optional in this environment. By this we mean that the local subtasks which are allocated to a container need to be considered in the SEP. If the application consists of just a single container the SEP format outlined in section 3.4.2 is enough to implement CS efficiently, given that all subtasks have been allocated to a single container. However we assume that most applications will consist of multiple containers in which subtasks are distributed amongst.

Although a local container in SSCF should concern itself with the load or overload in the entire application, to consider a global view and implement such changes as appropriate to meet application deadlines. However given it's local setting, it cannot enforce subtask changes in another container. Each container is isolated from one another in this regard and only communicate between one another via the data in the stream and meta-data attached in the SSCF messages, as outlined in section 4.3.2. Rather local containers are concerned with local executing subtasks and the decision to have a subtask execute or not for the next message received is a local concern. To further improve subtask selection speed and prevent the LC from searching a large SEP for possible subtasks to deactivate, the SEP is updated with two more columns to account for subtask locality in a distributed SSCF application. The SEP is updated by the local subtask profiler which is available to all containers as illustrated in figure 4.1, after the configurations are received by the local container. The manner in which configurations are distributed to containers is outlined in section 4.3.8.

To illustrate the updated SEP, table 3.1 outlines an example of an SEP given all subtasks in the application in a local setting this table is updated with the locally allocated subtasks including their message processing durations. Table 3.2 illustrates this update. When the local container starts up and receives the full SEP, the subtask profiler with knowledge of what subtasks has been allocated to the local container at design time, updates the SEP with the *Local Average Processing Duration* column as well as the *Local Subtasks* column. *Local Average Processing Duration* is generated by taking each of the subtasks in the SEP row which are assigned locally and adding their average messages processing times taken from the subtask profiler results.

To illustrate the SEP an example is provided in the next section.

3.4.4 SEP Example

Table 3.1 outlines an sample SEP where an example application consisting of nine unique optional subtasks as well as four mandatory subtasks, labelled S1-S13, were used in it's construction. This is an output result from our experimentation, which we outline in chapter 5. Suppose subtasks S2, S3, S12 and S13 are mandatory subtasks and are added to a list. The remaining subtasks are add to another list in numeric order.

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Accuracy (%)	Average Processing Duration Per Message (ms)
512	1,2,3,4,5,6,7,8,9,10,11,12,13	NA	100	0.0599002
191	1,2,3,4,6,7,8,9,10,11,12,13	5	86.1223	0.0596259
254	1,2,3,5,6,7,8,9,10,11,12,13	4	99.9612	0.0594282
62	1,2,3,6,7,8,9,10,11,12,13	4,5	86.0835	0.0591538
422	2,3,4,5,6,9,12,13	1,7,8,11	86.8458	0.0523959
167	1,2,3,4,5,6,7,9,10,11,12,13	8	99.9694	0.0523403
189	1,2,3,4,5,6,8,9,10,11,12,13	7	73.1391	0.0523291
163	1,2,3,4,5,6,7,8,10,11,12,13	9	99.9993	0.0522192
192	1,2,3,4,6,7,8,9,10,12,13	5,11	72.9681	0.0521215
231	1,2,3,4,6,7,9,10,11,12,13	5,8	86.0917	0.0520659
239	1,2,3,4,6,8,9,10,11,12,13	5,7	59.2613	0.0520548
227	1,2,3,4,6,7,8,10,11,12,13	5,9	86.1223	0.0519448
126	1,2,3,5,6,7,8,9,10,12,13	4,11	86.807	0.0519238
38	1,2,3,5,6,7,9,10,11,12,13	4,8	99.9306	0.0518682
47	1,2,3,5,6,8,9,10,11,12,13	4,7	73.1003	0.0518571

Table 3.1: Shows an example SEP given a set of nine optional and four mandatory subtasks within the application. Subtasks 2, 3, 12 and 13 are mandatory and thus are allocated to every row.

Suppose we want to calculate the included subtasks in the plan for *Subtask Plan Id 254*. An iterator in the algorithm, i is set to 509, the loop iterations starts at zero. 509 is represented as 111111101 in binary format, let this be str. str padded to $n = 9$ with zeros equals to 111111101, padding is not needed here. Iterating through each value in str and searching for any value equal to *1* gives a lists of all positions except position 2. Using these positions as an indexes into our optional subtasks list gives all optional subtasks except subtask S4, remember that mandatory subtasks are removed from this list such that S2, S3, S12 and S13 have been removed. Next we add any mandatory subtasks, and add this list as an entry in the SEP. We preform this function for all values between 0 and p_{count} to create the entire SEP.

Given that the binary representation of every value between 0 and p_{count} are unique, by using the *1* positions as indices into the optional subtask list, every permutation of optional subtasks created is unique. Adding each unique permutation to a single list of lists, creates the basis of the SEP within this thesis. Subtask profiling is mentioned in the next section 3.4.5.

Next, when a local container retrieves the subtask profiling information and SEP which was generated off-line, it updates the table with the local subtasks in each SEP row. As well, a new column is added which contains the cost of processing messages in each row with the local subtasks. This small optimisation is assumed to improve local subtask selection times. Table 3.2 illustrates this for a container which has been allocated subtasks 1-5 in the *Local Subtasks* column. Subtask processing times were taken from table 3.3.

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Local Subtasks	Local Average Processing Duration Per Message (ms)	Accuracy (%)	Average Processing Duration Per Message (ms)
512	1,2,3,4,5,6,7,8,9,10,11,12,13	NA	1,2,3,4,5	0.009255	100	0.0599002
191	1,2,3,4,6,7,8,9,10,11,12,13	5	1,2,3,4	0.008981	86.1223	0.0596259
254	1,2,3,5,6,7,8,9,10,11,12,13	4	1,2,3,5	0.008783	99.9612	0.0594282
62	1,2,3,6,7,8,9,10,11,12,13	4,5	1,2,3	0.008509	86.0835	0.0591538
422	2,3,4,5,6,9,12,13	1,7,8,11	2,3,4,5	0.001056	86.8458	0.0523959
167	1,2,3,4,5,6,7,9,10,11,12,13	8	1,2,3,4,5	0.009255	99.9694	0.0523403
189	1,2,3,4,5,6,8,9,10,11,12,13	7	1,2,3,4,5	0.009255	73.1391	0.0523291
163	1,2,3,4,5,6,7,8,10,11,12,13	9	1,2,3,4,5	0.009255	99.9993	0.0522192
192	1,2,3,4,6,7,8,9,10,12,13	5,11	1,2,3,4	0.008981	72.9681	0.0521215
231	1,2,3,4,6,7,9,10,11,12,13	5,8	1,2,3,4	0.008981	86.0917	0.0520659
239	1,2,3,4,6,8,9,10,11,12,13	5,7	1,2,3,4	0.008981	59.2613	0.0520548
227	1,2,3,4,6,7,8,10,11,12,13	5,9	1,2,3,4	0.008981	86.1223	0.0519448
126	1,2,3,5,6,7,8,9,10,12,13	4,11	1,2,3,5	0.008783	86.807	0.0519238
38	1,2,3,5,6,7,9,10,11,12,13	4,8	1,2,3,5	0.008783	99.9306	0.0518682
47	1,2,3,5,6,8,9,10,11,12,13	4,7	1,2,3,5	0.008783	73.1003	0.0518571

Table 3.2: Shows an example SEP with local container updates. A set of nine optional subtasks (Subtasks 1,4,5,6,7,8,9,10,11) as well as four mandatory subtasks (Subtasks 2,3,12,13) within the application are illustrated.

Next, we discuss how subtask profiling works. Profiling generates the message processing and accuracy contribution of each subtask within a streaming application.

3.4.5 Subtask Profiling

Subtask profiling is an important activity in CS given that it attributes both accuracy and message processing duration metrics to each subtask within a given application. From the profiling results, the unique subtask combinations generated using the process outlined in section 3.4.4 construct the entries within the SEP. Within the SSCF, subtask profiling is preformed by the subtask profiler as outlined in section 4.3.10 in chapter 4.

Subtask profiling is preformed on a sample set of input data or training data where all subtasks execute locally prior to the application executing in normal operations. Within SSCF subtasks can be distributed amongst a set of containers but for profiling purposes they must all be located locally, within a single container. The data used should represent a sample set of data which the application will process under normal operations. If not, the output from profiling will be skewed leading to sub-optional entries in the SEP which may not be selected correctly during application overload. Thus producing a sub-optional result on the processing duration and accuracy of output results. Subtask profiling executes all subtasks available to create a baseline result output for the application. For each optional subtask thereafter the application is run again with the same data set to create another set of results but with that optional subtask disabled. Then by comparing the baseline results against the result produced when the subtask was not active, an understanding of the subtasks contribution to the application begins to form. By repeating this process for all optional subtasks the cost of message processing in each optional subtask can be understood for the current data set.

For each optional subtask, total execution time is measured and divided by the number of messages it has processed to give an average message processing time for the subtask. Accuracy on the other hand is a more complex value to measure and is very much intrinsic to a stream application. Accuracy can be defined as the deviation of a value when compared against the ideal value. Through profiling by comparing the ideal result against the result produced when the subtask was not part of profiling, the deviation of that result to the ideal can be determined.

After the subtask profiling information has been collected and the SEP generated by the profiler, it is placed into Zookeeper for distribution and retrieval by each container in the application on startup. Zookeeper provides a number of services in SSCF, more information on Zookeeper can be found in the next chapter in section 4.3.11.

The advantage of profiling allows each subtask to be profiled before application start, thus both gaining an understanding of it's cost and contribution in the application. Additionally, it also has the benefit of again reducing non-functional actives in the application at runtime when the application is already overload.

However, the main disadvantage of this approach is concerning how well the sample data set supplied is representative of the data the application will be processing. Should this data change over time, the effect a subtask has on the application can change such that it's message duration processing times may increase or decrease. Thus SEP entries and selection may become sub-optional. A similar consequence can also be concluded with subtask accuracy.

Subtask Id	Accuracy Loss (%)	Accuracy Contribution (%)	Average Processing Time (ms)
1	99.9808	0.0191698	0.00819911
2	NA	NA	0.0002978
3	NA	NA	0.000012
4	99.9612	0.0388034	0.000472032
5	86.1223	13.8777	0.000274322
6	79.391	20.609	0.0103822
7	73.1391	26.8609	0.00757104
8	99.9694	0.0306099	0.00755992
9	99.9993	0.0007	0.00768102
10	74.5905	25.4095	0.00833841
11	86.8458	13.1542	0.00750432
12	NA	NA	0.000694456
13	NA	NA	0.000925529

Table 3.3: An example output of subtask profiling as taken from our experimentation, showing the accuracy contribution and loss as well as the average message processing cost within the application.

Table 3.3 illustrates the output from subtask profiling in an example application consisting of thirteen subtasks, nine of which are optional and the remaining four subtasks are mandatory. From it we can see each subtask which has been profiled. Subtasks with values equating to NA cannot be given an accuracy contribution given they are mandatory subtasks and their effect on the application cannot be measured as they cannot be disabled per application requirements. Further, the average message processing cost for each subtask is shown. The methodology used to generate accuracy values can be found in section 5.2.6.1. Given accuracy is application specific it is included with implementation and evaluation methodology.

Finally, the accuracy loss and accuracy contribution of each optional subtasks is also shown. As mentioned in section 3.4.2 given that subtask selections are deterministic for each application result created with a set of subtasks as indicated by the entry in the SEP, the resultant accuracy contribution of the listed subtasks can be determined for that particular entry. Error bounds for each output result can be consequently determined given that any output result is a partial result computed without a subtask or set of subtasks. Thus error bounds (Accuracy Loss) as presented in table 3.3 is an aggregation of each subtasks inaccuracy contribution.

Now that we have outlined how subtask profiling is preformed to create the entries in the SEP, overload detection and when how SEP entries are selected are covered next.

3.4.6 Overload Detection

Overload detection is an important part of any stream computing application handling input stream overload. As such we have developed an approach in detecting application overload which fits into each LH component within the SSCF.

An overload is detected by measuring the current volume of messages within the local container and determining if they can be processed on time. On time here refers to the local processing duration limit, l_{limits} applied to each local container. This value is derived from the total length of time the application has to process a message before a message becomes late. In an application consisting of multiple containers, each container is assigned a portion of time to process the message. This portion of time is determined by the processing duration of the assigned subtasks. For example, if a message must be processed in 1 second, and there are ten subtasks in total each taking 100ms to process the message, if 4 subtasks were allocated to one container, that container would have a l_{limits} of 400ms, with all subtasks being equal in message processing duration.

As shown in figure 4.1 and mentioned previously, each SSCF container contains two queues and a PE in which messages may reside. Aggregate message counts come from the count of messages in both queues within the container as well as any other messages which may reside within the PE. For example, in the case of a stateful PE messages are retained within the PE over time and could be processed again as new messages are received.

Equation 3.2 and equation 3.3 outlines the general equations used in this thesis to detect application overload in the local container in all but two versions of CS. CS-C and CS-B used an improved approach which we will discus later in this chapter.

$$p_{total} = m \times m_{average} \tag{3.2}$$

$$l_{total} = \frac{p_{total}}{l_{limits}} \tag{3.3}$$

Where m is the count of messages in the local container and p_{total} defines the total processing duration of all messages within the local container. l_{total} defines the total estimated average cost of processing all current messages in the local container, where $m_{average}$ is the average cost to process a message with all local subtasks determined by profiling and l_{limits} is the processing duration limit attributed to the local container. If l_{total} was greater 1 the local container was deemed overload, outlined in equation 3.4. Any value above 1 indicated by how much the local container was overload as outlined in equation 3.5 where o is the overload amount . l_{total} was then used to determined what SEP entry was selected, such that the total processing duration cost of the volume of messages was not greater than l_{total}. The selection processes are outlined next in section 3.4.7.

$$l_{total} >= 1 \tag{3.4}$$

$$o = 1 - l_{total} \tag{3.5}$$

For example, if the local container has a 500ms end to end latency requirement but the total expected processing time for all messages within it's local remit equal to 600ms, then the local container is overloaded by 100ms of processing time.

Overload detection was preformed on a message by message basis as new messages entered the load handler. This has the benefit of providing for a quick detection of overload per our proposed approach and adaptation of the application can ensue quickly there after. However, the downside of this is that it may slow down the consumption rate of messages in the stream if the cost of measuring message volumes, preforming the overload calculation and any subsequent SEP selection activities are too high. This is important to note if l_{limits} is low where the cost of measuring the volume of messages in the application at this level of granularity leads to a delay in the stream and subsequent delayed output results.

Placing messages in a queue and measuring the average processing times of messages has a number of advantages:

1. It removes the need to measure the total processing cost directly, or CPU cycles needed to process each message within the stream. Given the messages can be of a complex nature, such as a set of Twitter Tweets used in our evaluation, as outlined in section 4.6. The cost to measure the total CPU cycles are significant at any moment in time for all messages in the system, which can greatly consume resources for non-functional tasks in an already overloaded system.

2. By having overload detection placed prior to it's associated PE and thus taking a measure of current and future work by measuring the volume of data in the queues, an approximation can be made of the amount of time required to process this workload. In the case of a detected late arrival of results an adaptation can occur in the processing of this data before it is processed.

3. Other programs operating within the same physical host of the application may consume resources from time to time for their actives and thus may slow down the processing of messages in the SSCF. Although this is not directly related to a bursty data stream, it can appear as though it is bursty and as such messages build up in the queues. At this point, CS detects that the application is overloaded, given that messages have begun to fill up in the queues, and adapts the subtask selection accordingly, reducing the cost of processing each message. Thus increasing message throughput rates. However, we also assumed that any application will be correctly deployed and provisioned through capacity planning as best possible such that bursty data streams and application *slow downs* are infrequent.

The downside of measuring a message to an approximate processing duration in time is the lack of specificity in how long it takes to process each message. The expectation is that message processing durations will fall into a normal distribution over time. As such we assume message processing times will average out over time as more and more messages are processed. Another downside, relates to the current model of how long a message takes to process, such that if it changes over time the systems expected view of message processing duration may change. For the purposes of this thesis, we assume that is not the case and the concept drift techniques could be applied where expectations are updated from time to time as within the data mining community [47].

3.4.7 SEP Selection

The last common element within each version of CS was how entries within the SEP are selected. The LH receives messages at higher input data rates and determines it cannot process the volume of messages in the local container on time. It is the responsibility of the local LH to determine how to act. In the case of a LC, the LC selects the most appropriate SEP entry such that combined processing duration of the included optional subtasks will ensure that the messages are processed on time in the local container. Thus a processing duration violation in the local container will not occur, l_{limits} will not be exceeded.

Each entry in the SEP table is iterated and each entry is checked to determine if the optional subtask subset provided in each entry provides for enough time to process all messages in the local container. For CS-A and CS-D in the SEP, equation 3.3 is recalculated where the $m_{average}$ is exchanged for the cost of processing the messages within the new subtask selection for each entry selected. If the calculation, as before, produces a value greater than 1, then the next iteration is checked in the table. If the value produced is less than 1, the entry is selected and the message is updated with the selected subtasks it will be processed with.

In SSCF as messages are passed from container to container, the set of subtasks which the message has been processed with are available in the *SubtaskSelection* field within the message itself. This is illustrated in table 4.1 in the message properties.

Selecting subtasks in this way and iterating the SEP using linear search is a naïve approach which leads to a worse case time complexity cost of $O(n)$ as each entry in the table may need to be checked to see if it's an appropriate selection for the given message volume in the local container. A best case complexity of $O(1)$ can occur if the first entry in the list is selected.

In all version of CS this is the process followed to select appropriate SEP entries, in CS-B the process is different but we will discuss this in section 3.8.

3.4.8 Section Summary

In section 3.4 we outlined the load handling approach taken in CS. Section 3.4.1 outlines the general approach such that a stream application constructed in a CS manner with a number of individual subtasks. Individual subtasks can be disabled temporarily to increase message processing throughput in the application by reducing the computational cost of processing messages with the remaining subtasks.

Section 3.4.2 outlines how this is implemented with a deterministic Subtask Execution Plan (SEP) where each entry in the table is a unique set of subtasks and each entry has different message processing duration costs as well as accuracy and error bounds. Section 3.4.5 outlines the subtask profiling approach taken to determine such costs for each subtask.

Finally, section 3.4.6 and section 3.4.7 outline how an overload is detected and how entries in the SEP are selected during application overload.

In the next four sections, we outline each of the four versions of CS which have been designed and explain their operations with examples.

3.5 CS - Duration Focused

CS-D was the first design of CS to validate the approach and as such it has a naïve design to measure the cost of processing all messages in the local container, as well as selecting an appropriate entry in it's SEP. This has been outlined previously in section 3.4.6. We briefly outline it's approach again before describing an example.

CS-D was designed and implemented in a LC, titled Load Classifier - Duration Focused (LC-D) which was responsible for determining which set of subtasks were selected for execution within the local PE at runtime. During periods of normal load, such as when the system was not overloaded, the selection was simply the full subtask set allocated to it's local container. However, during overload periods the selected subset subtasks were chosen from navigating the SEP, selecting a set of subtasks which can process the volume of messages in the local container on time.

CS-D performs it's work within a loop consuming messages from it's input queue. Output messages are placed on it's output queue as illustrated in figure 4.1. Upon each new message arrival at the load classifier, a new measure of the total processing time required to process all messages in the local container was determined, p_{total} as outlined in equation 3.2. If this value is greater than l_{limit}, (the processing duration limit of the local container) then the SEP was navigated to select an appropriate subset of subtasks which could ensure the current message volume would be processed on time.

The subtask execution plan as outlined in section 3.4.2 is navigated one entry at a time. For each entry in the plan, the combined subtask processing time s_{total} is multiplied by the current volume of messages in the local container m_{local}. This value was then measured against the local containers message processing duration limit l_{limit}. If total cost of processing all messages in the local container was greater than the processing duration limit assigned to the container, the loop iterates and moves to the next entry in the plan to make the same determination. This is expressed in the following equation:

$$s_{total} \times m_{local} <= l_{limit} \tag{3.6}$$

When an entry is found, the subtask selection is made. The incoming message is updated with the current subtask selection based on the contents of the entry in the SEP that is selected. This notifies the PE when it receives the message which subtask combination should be used to process it locally.

If no entry is found in the SEP which allows processing of m_{local} within the local container within l_{limit}, the selected subtasks default to a selection of all mandatory subtasks allocated to the local container. The message is updated with any subtask selection made and added to the load handlers output queue for processing in the local PE with the selected subtasks. If no mandatory subtasks are present in the local container, then the message will pass through the PE and on to the next container downstream.

To further explain this process, we provide a detailed example in the next section.

3.5.1 CS-D Example

Suppose we have an application which has ten subtasks of which three subtasks are mandatory number S1-S6, S4, S5 and S6 are mandatory (we choose smaller example values to illustrate a full example). Table 3.4 outlines the results of subtask profiling these subtasks upon a sample data set. These metrics were generated after profiling each subtask. The application has 500 ms to process a message as it's total processing duration limit.

Subtask Id	Accuracy Loss (%)	Accuracy Contribution (%)	Average Processing Time (ms)
1	80	20	1
2	90	10	2
3	85	15	3
4	NA	NA	4
5	NA	NA	5
6	NA	NA	6

Table 3.4: Subtask profiling example metadata

Further, suppose the subtasks are distributed amongst two containers, where container c_1, gets allocated a l_{limit} of 200 ms while container c_2 gets allocated 300 ms. This is allocated as such given the higher costs of processing messages in c_2. Further, c_1 is allocated subtasks S1-S3 and c_2 is allocated S4-S6. As a reminder, if message queuing and processing duration times are greater than a l_{limit} value in its respective local container, a message is deemed late. The SEP has been generated for the application which has a total of 8 entries, 2^3 as calculated from equation 3.1. This table was retrieved from Zookeeper and the local subtask profiler populated the local subtask selection and processing durations based on the assigned local subtasks.

Table 3.5 shows the full SEP for the subtasks outlined in this example. Note, per the priority order CS-D gives to message processing duration, the table has been ordered with decreasing order of message processing duration. Further for the purposes of this example, subtask processing order is irrelevant.

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Local Subtasks	Local Average Processing Duation Per Message (ms)	Accuracy (%)	Average Processing Duration Per Message (ms)
8	S1, S2, S3, S4, S5, S6	NA	S1, S2, S3	6	100	21ms
6	S2, S3, S4, S5, S6	S1	S2, S3	5	80	19ms
7	S1, S3, S4, S5, S6	S2	S1, S3	4	90	18ms
5	S1, S2, S4, S5, S6	S3	S1, S2	3	85	18ms
4	S3, S4, S5, S6	S1, S2	S3	3	70	18ms
3	S2,S4,S5,S6	S1, S3	S2	2	65	17ms
2	S1, S4, S5, S6	S2, S3	S1	1	75	16ms
1	S4, S5, S6	S1, S2, S3	NA	NA	55	15ms

Table 3.5: Example SEP for CS-D, ordered by message processing duration.

Under normal operations, c_1 can process a message within the data stream in approximately 6 ms, when no queueing time is considered. Table 3.5 outlines this with *Subtask Plan Id 8* where within this local container, S1-S3 are executing. Now suppose a burst occurs in the data stream and the volume of messages rise in the input queue for LC-D. Suppose, on the next iteration of LC-D's loop, the message received rate grows beyond what is previously was, not enough to cause overload but messages begin to queue up and spend some time within the input queue. On the next iteration after this, the load handler checks the count of messages in the local container and now shows that there are 50 messages within the queue.

Based on equation 3.2, p_{total} = 300 ms. Based on equation 3.3 and equation 3.4, CS-D determines that the local container is overloaded by 50% and an adaptation of the application must occur. From table 3.5 *Subtask Plan Id 8* is not capable of delivering messages on time further. CS-D begins to navigate the table, applying equation 3.7 to determine the next best subtask selection which can ensure l_{limit} is not violated. When applying equation 3.7 to *Subtask Plan Id 6*, the next entry in the table, CS-D takes it's local subtask processing durations into consideration. In this the local container can reduce the cost of processing each message locally by 1ms thus the new p_{total} equates to 250 ms which is still above the l_{limit} of 200 ms.

LC-D now moves on to the next entry in the SEP, *Subtask Plan Id 7* and makes the same calculations. This time p_{total} equates to 200 ms which satisfies equation 3.4.

LC-D selects this entry in the SEP and updates the message with subtask ids for S1 and S3 and passes the message to it's local PE for processing. Provided c_2 does not modify the subtask selection and in this case it cannot given S4-S6 are mandatory subtasks and must execute, the output result will be 90% accurate. On the next message arrival at LC-D, the same assessment of message volume is made. If the volume of messages in the queue has increased further, entries in the SEP are assessed to determine if each subsequent selection can meet the applications processing duration requirements.

3.6 CS - Accuracy Focused

The accuracy focused load classifier LC-A based on CS-A, follows the same overload determination and overload amount measures as CS-A outlined in section 3.5. In fact, the only different between the two approaches is the SEP ordering.

In CS-A the SEP is ordered giving a priority to accuracy rather than primarily to message processing duration. Having said this, CS-A is still concerned with processing messages on time but accuracy is a more important factor. This decision was made to determine if a focus on accuracy can lead to results being more accurate while also ensuring results are delivered on time, unlike CS-D which focuses just on result timeliness. The downside to this however, is that results may be more accurate when produced but may be late and such have little value.

An example is provided to show the differences between the SEP orderings between CS-D and CS-A in the next section.

3.6.1 CS-A Example

This example is based on the scenario as outlined in section 3.5.1, for the sake of brevity please refer back to this example. Table 3.6 shows the local SEP for c_1 after the Subtask Profiler has received the SEP from Zookeeper and added the local allocated subtasks to the plan. Here the SEP is organised by average accuracy contribution in descending order.

Continuing with the example, as with CS-D, CS-A's *Subtask Plan Id 8* is not capable of delivering messages on time, given there are now 50 messages in the queue. CS-A begins to navigate the table, applying equation 3.7 to determine the next best subtask selection which can ensure l_{limit} is not violated. When applying 3.7 to *Subtask Plan Id 7* , the next entry in the table, the CS-A takes it's local subtask processing durations into consideration. The local container can then reduce the cost of processing each message locally by 2ms, thus the new p_{total} equates to 200 ms which is equal to l_{limit} value of 200 ms for the local container.

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Local Subtasks	Local Average Processing Duation Per Message (ms)	Accuracy (%)	Average Processing Duration Per Message (ms)
8	S1, S2, S3, S4, S5, S6	NA	S1, S2, S3	6	100	21
7	S1, S3, S4, S5, S6	S2	S1, S3	4	90	18
5	S1, S2, S4, S5, S6	S3	S1, S2	3	85	18
6	S2, S3, S4, S5, S6	S1	S2, S3	5	80	19
2	S1, S4, S5, S6	S2, S3	S1	1	75	16
4	S3, S4, S5, S6	S1, S2	S3	3	70	18
3	S2,S4,S5,S6	S1, S3	S2	2	65	17
1	S4, S5, S6	S1, S2, S3	NA	NA	55	15

Table 3.6: Example SEP for CS-A, with an SEP ordered by average accuracy contribution.

Comparing CS-D's SEP in table 3.5 to CS-A's SEP in table 3.6, noticeable differences emerge in the ordering of both tables.

Take for example the second entry in both tables, in table 3.5 it can be identified as *Subtask Plan Id* 6 and in table 3.6 it can be identified as *Subtask Plan Id* 7. *Subtask Plan Id* 7 produces a result which is 10% more accurate on average and at a cost of 1ms less than *Subtask Plan Id* 6 in table 3.5 when navigating both tables in linear search order.

As such in certain cases, depending on how subtasks are weighted in the application in both processing duration costs and accuracy contribution, CS-A can produce a more accurate results in less time than CS-D. Given that in this case, CS-D had to check two entries before it found an appropriate value entry which can meet the application processing duration requirements. However, note this deduction is highly specific to an application and the subtask implementation, but analysis of the generated SEP after subtask profiling can elude to an ordering priority which yields the best results for the application.

3.7 CS - Cost Focused

As mentioned in section 3.3 and eluded to in section 3.5, the navigation of the SEP particularly when the number of subtasks in an application are large, can take some time due to linear search with a worst case time complexity of $O(n)$. As such, activities should be minimised to reduce any extra CPU time spend on activities which detract from processing messages in the application directly. This is the purpose of CS-C in a load classifier, titled LC-C.

During a bursty data stream, the application and system are already overloaded at 100% CPU saturation. As such, the purpose of Computational Shedding - Cost Focused (CS-C) and Computational Shedding - Binary Search (CS-B) for that matter, are to reduce the complexity and cost of determined overload as well as attempting to reduce the cost of selecting a subtask combination in the SEP at runtime. CS-A's and CS-D's approaches are preformed with the four equations outlined in section 3.4.6 for each message received by their respective load classifiers.

CS-C improves upon CS-D and attempts to reduce the cost of measuring overload in it's associated load handler using a difference approach. Rather than measuring the cost of processing every message in the local container with each entry in the SEP by the load handler, CS-C takes a measure of the maximum allowed processing time per message m_{max}, only once. It then measures this value against the expected local execution time of messages in the local container. As outlined by equation 3.7.

$$m_{max} = l_{limit}/m_{local} \tag{3.7}$$

$$s_{total} >= m_{max} \tag{3.8}$$

When m_{max} was calculated, each entry in the SEP was checked using linear search to determine if that particular entries estimated message processing cost was lower or equal to m_{max}, as outlined in equation 3.8. If so, that entry in the SEP was selected and the message was updated with the local subtasks indicated by this entry in the SEP. If not, CS-C moves on to the next entry in the SEP, to make the same determination. As with the other load classifiers, if no entry is found, all mandatory subtasks are added to the message. If no mandatory subtasks are allocated to the local container, the message is passed to the PE and passed on to any down stream containers.

3.7.1 CS-C Example

Using the details of the example provided in section 3.5.1 including the SEP in table 3.5, m_{max} is calculated to be $200ms/50 = 4ms$ for c_1 using equation 3.7. Given that, the first entry in CS-C's SEP takes 6 ms to process a message, c_1 cannot deliver the message within required processing durations.

CS-C preforms a linear search on the SEP navigating to the next entry, which again cannot process a message within the allotted time. CS-C checks the next entry in the table. Using equation 3.8, CS-C selects *Subtask Plan Id 7* with it's associated subtasks to process the message in c_1 as 4 ms is equal to the 4 ms required processing time for a message.

This is the same selection as CS-D but determined with a reduced cost since the cost of processing messages was calculated just once before each SEP entry was checked.

3.8 CS - Binary Search

The final version of CS which was designed was the Computational Shedding - Binary Search (CS-B) which further improves upon CS-C by reducing the cost of navigating the SEP to find an appropriate entry. The binary search load classifier LC-B based on CS-B, attempts to improve upon CS-C by reducing the cost of selecting an entry in the SEP. The worst case time complexity cost of the SEP was $O(n)$ in this case, using linear search, given that every entry in the plan may need to be checked to determine if it's an appropriate selection for the current message volume in the local container. This was the case for all versions of CS other than CS-B.

However, an improvement can be made in the search cost in navigating the SEP by using a Binary Search Algorithm. This algorithm has a worst case time complexity of $O(logn)$ and best case time complexity cost of $O(1)$[33]. CS-B has an SEP ordered by decreasing message processing duration times, supplied by the subtask profiler and as such a Binary Search Algorithm implementation was applicable to this ordered array. A sorted array in incrementing order being one of the requirements of the Binary Search Algorithm.

The operation of LC-B was very similar to LC-C where m_{max} was determined first, the Binary Search Algorithm was then performed as follows:

1. Compare m_{max} with the middle SEP entry message processing cost.
2. If m_{max} matches with middle entry, return the mid index.
3. Else If m_{max} was greater than the mid entry, then m_{max} can only lie in right half sub-array after the mid entry. So we search again for right half.
4. Else (m_{max} is smaller) so we search again for the left half.

After the particular SEP entry was selected the normal process of updating the message with the selected local subtasks was made and the message was added to the output queue of the load classifier for processing in the PE.

Using this approach has the benefit of searching a large SEP more quickly than a simple linear search which may iterate the entire table to find an appropriate entry. This has a major benefit in a load classifier, by reducing the effect of measuring the stream and thus reducing the effect of slowing it down. This is as a result of checking less entries in the SEP before a message is let through the LC and into it's local PE.

To illustrate the performance improvement, take for example an SEP with 100 entries. In this case linear search may check 100 entries to find an appropriate SEP entry where as binary search would have to make $log(100) = 7$ on average.

However, for the Binary Search Algorithm to be applicable, the SEP must be ordered and in this case it is ordered in messages processing duration, given the timely requirements of a stream computing application. Hence LC-B is based on the SEP ordering as supplied in LC-D rather than in LC-A as there is no apparent order in message processing duration there.

3.8.1 CS-B Example

To illustrate the improvement in the use of binary search, we again take the example as supplied in section 3.5.1. However, this time we suggest that the message volume in the queues of c_1 contain a total of 75 messages. This makes m_{max} this time 3 ms. With a linear search of the SEP supplied for CS-D in section 3.5.1, it would take CS-D 4 possible iterations of it's SEP to find an entry which can satisfy the processing duration requirements of c_1. Such that the 75 messages could be processed within 200 ms, using *Subtask Plan Id 5.* Using a binary search algorithm instead, the worst case time complexity is 2 ($log(3)$), and as such binary search could find entry *Subtask Plan Id 5* twice as fast as linear search, even in small SEP tables.

To illustrate this, we work through the Binary Search Algorithm to show that it's time complexity factor will be 2, using the algorithm as outlined in section 3.8.

1. Compare m_{max} with the middle SEP entry which in this case is the 4 entry, *Subtask Plan Id 5.*

2. Given that m_{max} we are searching for is 3 ms, we have found an appropriate entry in the SEP in constant time.

3.9 Chapter Summary

This chapter outlines the CS approach to managing bursty data streams by reducing the computational cost of processing messages at runtime and thus processing all messages in the data stream to a partial degree during intense but short-lived data bursts. This is achieved by constructing stream applications into individual subtasks such that optional subtasks can be temporarily disabled. The concept is outlined in section 3.4 along with a proposed approach on how subtask selections occur at runtime in the SEP and how overload is detected in the stream application.

Further, four versions, each which improve in some form on one another have been proposed. CS-D as outlined in section 3.5, priorities message processing duration and searches the SEP in linear time using a naiive approach to select appropriate SEP entries and ultimately, appropriate subtasks which can ensure message processing deadlines are met. A detailed example is provided of this approach in section 3.5.1.

CS-A improves upon CS-D, prioritising estimate application output result accuracy over message processing durations such that the SEP is ordered in descending order of expected application output accuracy. Although messages deadlines are the most important criteria of stream computing to honour, a focus on accuracy as well is an important element, the results of which need to be analysed against other CS designs. This design is outlined in section 3.6. A detailed example is provided of this approach in section 3.7.1.

CS-C was designed to help reduce both the cost of measuring local container utilisation and the local containers future work to help ensure message results were delivered on time, a design improvement of CS-D. It's design can be found in section 3.7. An example of the optimisation it provides can be found in section 3.7.1.

CS-B was the last improvement made in this thesis to help reduce the navigational cost in the SEP using a binary search algorithm and as such provides an improvement in load classifier processing times. The design of CS-B can be found in section 3.8 and a worked example showing it's improvement can be found in section 3.8.1.

In the next chapter, we outline the implementation details of the SSCF and explain the proof of concept application which was built on SSCF to evaluate the four implemented CS versions. Also two load shedders were also implemented to provide a comparison for CS, these are discussed also.

Chapter 4

Computational Shedding (CS) and The Simple Stream Computing Framework (SSCF)

4.1 Overview

This chapter contains implementation details of our Computational Shedding (CS) approach. To evaluate our approach a distributed stream computing framework was developed, titled Simple Stream Computing Framework (SSCF), where the CS approach resides as an alternative approach to load shedding when dealing with bursty data streams.

This framework has been used in both the data collection for our experimentation by extracting data from the Twitter Public API [66], and for the evaluation of our approach through a developed proof of concept search application, searching a Twitter data stream for relevant Tweets to an input search query. The application itself, including implemented subtasks built within SSCF, can be found later in the chapter in section 4.6.

This chapter begins with a description of the SSCF and its architecture in section 4.3. We provide a brief overview of the components contained within. The most important components and algorithms are discussed in some detail. We describe how CS was implemented in the framework and how the framework supported subtask selection, as outlined in chapter 3.

We have implemented both Random and Semantic Load Shedders based on work by the Borealis and Aurora Projects [111, 118]. CS is compared against other load handling alternatives, the results of which are described in the next chapter. Here, the implementation for both load shedders can be found in section 4.4.1 and section 4.4.2. An in-depth description of the work to implement CS in SSCF can be found in section 4.5.

Finally, the results of this work are presented with analysis in chapter 5.

4.2 Motivation

The primary purpose of this implementation was to assess the accuracy and processing duration of Computational Shedding against Random and Semantic Load Shedding during a varying degree of system overload rates. To that end, we chose to evaluate our strategy and implementation within the context of an experiment based around a real world use case, focusing on the effects each of the overload handling strategies has on the application.

The purpose of this is two fold. First, the availability of simulators tailored to measuring overload handling strategies are not available or have not been maintained throughout the years. For example, the research groups involved within the Aurora and Borealis project have been disbanded [116] and work on this system has ceased since 2008. Other frameworks such as S4 [94], have ceased development and have not developed any load handling strategies. Thus the work to implement overload handling strategies within this framework would far outweigh the benefit when compared to a framework developed with load handling strategies developed at it's core.

Further, current popular stream computing systems consist of trivial data types within messages, which are passed between local and remote stream operators in a DAG. For example, current position and temperature measures expressed as integer types [2, 13, 94]. We assert that, in a system representing a more complex real world application, a more complex data model is needed to correctly capture the nature and semantics of the application.

Consider the Twitter Tweet data model [65]. Twitter is a popular on-line news and social networking site where users interact by posted messages called Tweets, which contain more than 37 different properties including multiple data types such as integers, strings, locations, lists and dictionaries to capture the nature of a Tweet. Take for example a location which was represented as a set of longitude and latitude coordinates. This level of complexity in the data models was not represented by many popular existing works within the context of their simulators. Our aim within our experimentation was to mimic a real world experiment as best as reasonably possible while measuring system and application overload characteristics.

Second, benchmarks such as the Linear Road Benchmark [13] which simulate dynamic toll charges within a simulated high-way, have become popular to evaluate and compare stream computing systems in the state of the art. The benchmark specifies the following queries: (i) provide toll notifications to vehicles within 5 s; (ii) detect accidents within 5 s; and (iii) answer balance account queries about paid toll amounts. The goal of the benchmark was to support the highest number of express-ways L while satisfying the latency constraints. As mentioned in work with the Stream Processing Core [69] the Linear Road Benchmark took a focus on the application and it's output results, unlike other benchmark implementations in stream computing and data stream mining, which took a system focus measuring throughput and end to end message latency.

Our aim was to have the same application focus, however the Linear Road was tailored to measuring the scalability of stream computing systems without a direct focus on handling system overload and the effect this may have on the output of the executing application. Thus the L value for a stream computing system was assessed at the point the system cannot cope with more express-ways, or more data. In our work, this was where we wished to start our assessment and as such the benchmark was not appropriate for our evaluation in it's current form. A sentiment which is shared in other newer works [32].

With all this in mind, we developed a stream computing framework which fulfils the characteristics as a distributed message passing framework while also having load handling semantics at it's core as a primary means to handle system overload.

4.3 SSCF: Simple Stream Computing Framework

The Simple Stream Computer Framework (SSCF) can be described as a set of connected nodes running in sequential order to process messages in a data stream. Figure 4.1 outlines the typical architecture of a SSCF Processing Node (PN). An application built atop SSCF can contain multiple PNs, which in turn can contain multiple containers communicating between one another via a shared computer network. SSCF was built atop Microsoft's ASP.Net Core 2 framework and built using the C# language [46] using Object Orientated (OO) principles. The ASP.Net Core 2 framework includes native support for HTTP communications with the inclusion of HTTP clients and servers. ASP.Net Core 2 follows an MVC style design pattern for development of HTTP based applications which provide a clean software design strategy.

Next we discuss and describe all major components of the framework as outlined in figure 4.1.

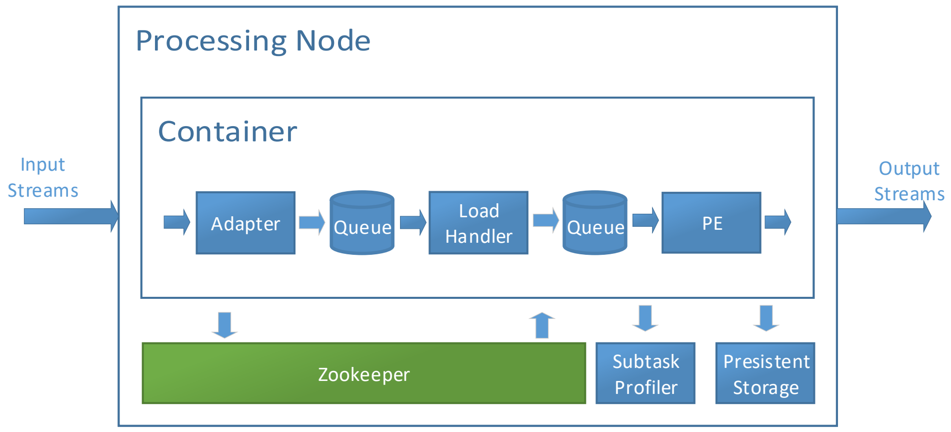

Fig. 4.1: A component diagram of SCCF, showing a Processing Node (PN) with a single Container. Typically more than one container executes within a PN at a time. Each container contains a SSCF stream computing pipeline including a load handler. Communications between containers are orchestrated with Apache Zookeeper [45] for service discovery. Blue components represent components that were constructed. Green components represent components or services which where implemented by a third party but used within this research

4.3.1 Processing Node

A Processing Node (PN) is a local association of one or more containers which reside on the same physical host. In the case of multiple containers either deployed locally within the same PN or remotely on other PNs, service discovery is facilitated via Apache Zookeeper [45]. Each PNs accesses to the subtask profiler for subtask profiling activities at application start. The PN also has an interface layer for access to persistence storage, be it local file system or database access. For example figure 4.2 shows a single PN having access to a database at the end of the application pipeline for persistence storage. The database here is used for result storage used in the analysis as outlined in chapter 5.

4.3.2 Containers

Each container, takes one input stream consumed by the adapter and can produce a single output stream within the PEs output stream handler. PE are discussed in section 4.3.8 later in this chapter. Containers are a logical grouping of components, an association based on their direct activities to process the incoming data stream. This grouping aids in the orchestration of both the creation of components and communications between components, such as referencing of the inner queues, or creation of the specified load handler. Within a PN, multiple containers can reside to process multiple data streams, allowing for parallelism of an application on a single physical host across multiple data streams.

4.3.3 Adapters

Adapters preform translation work from data outside the container to an understood format within the container. This translated model is processed by the PE downstream. Within SSCF, communications between adapters is preformed over HTTP to facilitate ease of use within other programming languages facilitated by constructs in many language frameworks, such as default http client and servers as mentioned in ASP.Net Core 2 [46].

SSCF contains three adapter types; first an adapter to consume data directly from a Twitter Stream. The adapter consumes the complex Twitter Data Model and translates it into an internal model which was persisted to local disk in JSON format. Section 5.2.2 in chapter 5 covers this process in more detail.

Second (Rate Adapter) an adapter to read the saved Twitter data from local disk and serve this data into the associated load handler and PE at a variety of data rates to simulate a real world data stream with bursty characteristics. Data rates are controlled using a Token Bucket protocol [11] where data is delivered as messages per second. SSCFs internal message model, as outline in table 4.1 is quite generic allowing any object payload to be attached as the payload to push through the framework.

This adapter facilitates the simulation of a bursty data stream which delivers the stored Twitter data at the application at higher and higher input data rates. This adapter was developed as a result of the public Twitter data stream serving only 1% of Twitter traffic on the free tier, which was far to slow for experimentation with SSCF. The full Twitter data stream can be purchased. Chapter 5 will further discuss the experimental methodology.

The final adapter is a Input Adapter which acts as a HTTP Server to consume messages which are produced by other containers in the application. In our application, all containers other than the first container in the application implement this adapter to pass messages from one container to another, as can be seen from figure 4.2. Communicates from one PE to another is facilitated by messages being sent from the output stream of one container over HTTP into other containers input adapter awaiting their arrival.

4.3.4 Queues

Within each container there are two in memory queues with first in first out semantics (FIFO), preserving input and output message ordering in SSCF. The SSCF queues, which are thread-safe are an abstraction built on top of *ConcurrentQueue* Collection class within .Net Core Framework [35]. Thread-safe queues are important here to ensure message ordering as well as speed by removing the need for thread locking semantics around the data structure in application code. The rational of using queues can be found in section 3.4.6. Fig 4.1 outlines the location of the queues between the input adapter and the load handler, and between the load handler and the PE.

Within each container there are two in memory queues with first in first out semantics (FIFO), preserving input and output message ordering in SSCF. The use of queues in the architecture provides for a number of benefits. First, queues provide a decoupling between the input data rate and shield component in which the queue is in front of. This is so to shield the component from any effects increased message volume would have on the component. For instance, large increase in input message rates may lead to larger interruption on the components normal operations. Queues are there to buffer against such interruptions.

Second in our design, as we discussed in section 3.4.6 queues provide for a means to measure current message volume and approximate the anticipated workload that local container will have to process. From this we can derive if the application is overloaded or not and thus instigate an adaptation of the local subtask within the local container.

Third, the use of queues allows for a message passing model such that SSCF components which operate within their own threads do not need communicate via a thread locking mechanism, and as such threads do not need to wait unit locks are released before it can process messages within the queue. Within SSCF the queues were developed on top of the *ConcurrentQueue*

Collection, a thread-safe abstraction in .Net Core Framework [35] which facilitates non-blocking queue implementations.

However, queues, unlike a application directly coupled to the input stream, can run out of storage space. This can occur when the input rate is so great that adaptation within the application cannot occur fast enough or if there is no further adaptations available such that only mandatory subtasks execute within the local container. As such the application throughput is slower than the rate at which input messages are received and queues begin to fill. Message delivery times may begin to become late. If the bursty continues, queues can overflow and result in uncontrolled message loss. Fig 4.1 illustrates the location of the queues between the input adapter and the load handler, and between the load handler and the PE.

4.3.5 Message Format

To communicate between containers in SSCF, a message passing model was used. Table 4.1 outlines the base message model used in SSCF.

Name	Description	Data Type
Id	Unique id of message	string
Payload	Payload of message defined as a Generic	Generic
InputTime	Time message entered the application.	DateTime
OutputTime	Time message has completed processing in the application	DateTime
LoadHandlerProcessingTime	Duration of time the load handler was processing the message (ms)	long
SubtaskSelection	Subtasks which were selected by the load classifier to process the message	List<int>
SlidingWindow	Optional Sliding Window attached when message is passed in the framework	SlidingWindow

Table 4.1: The structure of the internal message which is passed between containers in SSCF.

As a message is processed it's processing time within each PN is recorded and stored within the message. Additionally the length of time the message has spent processing in the load handlers is also recorded and stored within the message.

To allow a message to contain a flexible payload specific to each application, Generics in .Net and C# language [39] are used to define the type. Generics allow for the definition of type-safe data structures, without committing to actual data types, thus allow application specific data structures to be constructed for application built on SSCF. This was restricted to any object which implemented the IPayload interface. Thus any object which implemented the IPayload interface could be passed between components in SSCF.

In our proof of concept application, the payload property referrers to a type which represents an interface. With this we can pass in either a sliding window message or a serialised Tweet Message as the payload so it can be passed to the next component in the pipeline for processing. This allows for each subtask to generate it's output results and attach that result to the message before it is passed to the next subtask in the local container or sent from one container to the next in the application's pipeline. The data model for the *ParsedTweetMessage*, the most common payload type in this thesis and accompanying implementation can be found in table 4.4.

Additionally, a list of selected subtasks are attached to each message which passes through a load handler. In all cases but the load classifier, the selection of subtasks are the complete set of locally allocated subtasks to the container, ie: no subtask is shed. In the case of the load classifier, the selection was dynamic based upon the determination of current application load to expected processing message duration.

When the message is passed from load handler to PE, the selected subtasks are only activates if the message states as such. Subtasks not in the list are not activated and a result is not generated by that particular subtask.

Finally, messages which were processed by all selected subtasks in the local container are passed one at a time from the output of the PE to the input adapter in the next container in the chain.

4.3.6 Load Handlers

Load handlers as described in the previous chapter, attempt to control the flow of data coming into a PE during an overload situation.

Each load handler preforms it's work within it's own thread and within a loop continuously consuming messages from it's input queue, placed there by the adapter. Each load handler placed output messages within it's output queued for consumption in the PE.

In this thesis seven load handlers have been implemented, a load hander for baseline result generation, two load shedders and four difference implementations of CS. Each implementation of CS was an improvement upon it's predecessor in design, where each load classifier had a different focus. The aim with each was to asses which was best suited within the scope of this thesis. Each of the load handers are described in more detail below but we leave the implementation details of each load classifier for section 4.5 later in the chapter:

First, an empty load handler which merely takes messages from the adapter queue and adds these messages into the PE input queues. The purpose of this load handler was to create our initial benchmark application results. These ideal rules, results which have not been manipulated by load handling activities. These are compared to the results of each load handler varies output result within the remit of the experimental Twitter application described in section 4.6. By using a pass-through load handler, data would not be discarded or modified prior to being processed by the subtasks contained within the PE.

Second, we have implemented a Random Load Shedder based on work presented in the Borealis and Aurora systems [2, 28, 118] as well as within Tatbul's PhD Thesis[111]. The Random Load Shedders role is to stochastically discard incoming messages within the input data stream such that they are randomly selected. Thus the aim here was to produce a set of messages which were randomly selected, which were eventually processed by the PE. The Random Load Shedder implementation is discussed further in section 4.4.1 later in this chapter. All available subtasks in the application are executed with this load handler as it does not shed subtasks.

Third, a Semantic Load Shedder was also implemented from work in Tatbul PhD Thesis. The aim of a Semantic Load Shedder is to improve on a stream applications output result accuracy by improving upon the message selection process for discarded messages by attributed each message a utility value to the applications function. Thus during system overload the selected messages are of higher semantic value to the PEs workload than messages which are rejected. As loads increase messages with higher and higher semantic values are discarded. We discuss this further in section 4.4.2. All available subtasks in the application are executed with this load handler as it does not shed subtasks.

Finally, the load classifier which was implemented as part of the work of this thesis to uniquely classify the current load in the local container and determine a selection of subtasks from the SEP which best fit the current message volume in the local queues in the local container. The specific subtask selection was then used to process messages within the PE. As mentioned there are four specific implementations used in this thesis for CS which are discussed in more depth in section 4.5.

Another important function of the load handlers was overload detection as outlined in section 3.4.6 as outlined in chapter 3. Overload detection was implemented by taking a measure of the current volume of messages within the local container, within both it's input queue and output queue as well as sliding window and multiplying this by the average cost to process a message, a value determined by subtask profiling, which we discuss in section 3.4.5. Section 3.4.6 outlines the four equations used to determine overload in each of the load shedders and two of the load classifiers. Two other load classifiers used a difference approach in an attempt to reduce the processing time spend here but more on this will be covered in section 4.4.2.

As the application was given an total end-to-end processing duration for messages, this time was divided up between each of the container within the application. The division of this time, which we call l_{limit}, was based on the allocated subtasks to a container and the overall cost of processing messages within each subtask.

For example, suppose the application must process a message within one second. The application consists of three containers, each with an equal processing cost to process messages, regardless of the number of subtasks within a container. Then each container is given a l_{limit} of 0.33 seconds. If a message resides in any container for longer than this period of time, the adaptation process was initiated and in the case of a load classifier, searching the SEP to find an entry which would meet the new deadline. Further information on the overload detection and SEP selection is provided in section 4.5 later in this chapter.

4.3.7 Sliding Window

Subtasks execute within the confinement of a PE's input stream handler. A sliding window [125] is a custom data structure which holds the output results of the application, generated in the input stream handler of the last container in application, container 3. The sliding window "moves" with the data stream as it is processed giving a view of the current application and system at any point in time. Local container state is recorded after each message is processed. Table 4.2 as well as table 4.3 outlines the properties of the sliding window which are recorded.

In our proof of concept application, a sliding window represents the output results of the Twitter search application as well as the list of Tweets retained and their order, an order determined by the Tweets rank which was generated by the set of selected subtasks in each of the local containers. The sliding window was implemented with the IPayload interface which allows it to be the payload type for an SSCF message, a design pattern as outlined in section 4.3.5. As such it could be passed around the SSCF framework. In this application the sliding win-

Name	Description	Structure	Data Type
Items	list of application results	List<T>	Generic
GenerationTime	window generation Time	NA	long
LoadHandlerProcessingTime	average processing time of the load handler	NA	long
SubtaskSelection	list of selected subtasks ids used in window generation	List<T>	int
SubtaskMetadata	meta data of each subtask	Dictionary<int, object>	SubtaskMetaData
MessagesProcessed	number of messages processed by PE	NA	int
MessagesDiscarded	number of messages discarded from load shedder	NA	int
OverloadAmount	current system overload amount	NA	float
CurrentQueueCapacity	current aggregate queue capacity of containers	NA	int
CurrentWindowSize	count of application items in the window	NA	int

Table 4.2: An outline of the contents of the implemented sliding window object, which captures application results and system metrics on a per second basis.

dows are stateful, such that the list of valid messages grows and grows until the application has completed. This is opposed to a stateless sliding window, which after results are generated, the sliding window resets.

As mentioned previously, as messages move from one container to another, the results of the subtasks are attached to the message. When the message enters the last container, the results are normalised and aggregated within the bounds of the sliding window of time. From which a ranked order is generated and eventually persisted along with all other meta data in each message. The results of the sliding window are then persisted in the local PNs database.

Further we capture meta-data regarding data processing rates, the rates messages may have been discarded and meta-data in regard each subtask which has been selected for execution within the sliding window.

As mentioned data is persisted, this was the purpose of the storage container as outlined in figure 4.2, the storage container persists the final sliding window to disk which was used for analysis after the experiment had completed.

The results of the application and the output of each sliding window can be found in chapter 5. The results of the sliding window were shown to the user every second after application start for display purposes also.

Name	Description	DataType
Number of Messages Processed	Current number of messages processed by the subtask	int
Accuracy Lost	The percentage of accuracy lost to this subtask, measured by profiling	float
Total Latency in ms	Total processing thus far for this subtask	int

Table 4.3: An outline of the contents of the implemented subtask meta data object, which captures processing information on each individual subtask at runtime. Each subtask is updated as it preforms processing actives on incoming messages.

4.3.8 Processing Elements

A processing element (PE) is a logical association of an input stream handler, a possible output stream handler, a potential sliding window and any selected subtasks which have been chosen to process the next message received in the stream. The input stream handler executes in a loop with it's own thread, listening for messages from the input queue to the PE. When a message is received it is processed within each of the selected subtasks after which it is added to a sliding window. The selection of subtasks has already been made by the load handler associated with this PE in the same container. A common interface outlines the definition of a subtask which is then implemented in application code to implement a subtask. In the input stream handler, the *Subtask Factory* class, creates new instances for each selected subtask to process each message received. An instance of the Subtask Factory is Dependency Injected (DI) into the PE at application start [53]. DI is a technique for managing and controlling dependencies of a class in object oriented programming. Further, the input stream handler can process messages in groups should messages reside within a local sliding window of time, as is the case in container 3 in the application.

The results of any work preformed by each subtask was saved back on to the message. In the case of PN_3, producing results within the bounds of a sliding window, the aggregated sliding window results were stored in the sliding window payload of the message. Generics are used here allowing an SSCF message to hold a variety of object types as outlined in section 4.3.5 in chapter 3.

Within the PE, the output stream handler executes within a loop and listens to events from the input stream handler where messages are passed between them. The output stream handler in our Twitter search application had two functions;

First the output stream handler was responsible for key lookup for downstream containers in Zookeeper [45] to discover their IP address for communications. Zookeeper in this work was used for distributed key value storage and retrieval. Ever fives seconds key lookup was initiated. Communications was facilitated via HTTP and an encapsulated message was sent to the downstream container containing the contents of the containers work for that given message, results generated by the selected subtasks, as well as the message itself and any Twitter data parsed within it. An illustration of this can be seen in figure 4.2 in communications between the PN_1 and PN_2, and between PN_2 and PN_3. Remember, in our implementation a PN contains just a single container.

Second, as outlined in figure 4.2 PN_3, the final PN in the applications, it's output stream handler was responsible for persisted the generated sliding window and it's contents to the MySql database in Azure. This output handler was responsible for checked pointing sliding window state periodically for analysis of the search application, which we explain in section 4.6.

For this particular application to prevent any relative "slowness" in persisting results from PN_3 to the database, as relational databases can be relatively slow to write to, an additional queue was placed within the output stream handler. The input stream handler within PN_3 wrote sliding window results, at a rate of one per second into this queue. The output stream handler then picked up the sliding window message, writing the results to the database for analysis at a later time. Having a queue in this position, removes the slowness or backlog problem that might be present in waiting on writing data into the database. Additionally having sliding windows created just once a second, ensures that the queueing time of the messages in this queue does not grow given service time is much faster. As such the results of the application were not effected.

In terms of application development, both the input stream and output stream handlers are the location within the SSCF framework where application logic should reside for message and data processing.

4.3.9 Subtasks

Subtasks are built within the confinement of the *Subtask Abstract Class* and *ISubtask* interface where with each implementation the *bool Process(IPayload payload)* method must be implemented with functionality depending on the application. In our case, each of the 13 subtasks outlined in section 4.6.1 later in this chapter, had their own concrete type. Each subtask also has a boolean type to indicate if it is mandatory or not within the application.

After a subtask is constructed it is added to the Subtask Factory Class, a factory design pattern which instantiates each subtask outlined within it's *ISubtask GetSubtask(int id)* method. As mentioned in section 4.3.8, the Subtask Factory is dependency injected into the PE on startup. For each message received by the input stream handler within the PE, the indicated subtasks within the message payload indicate what subtask are to be instantiated by the Subtask Factory. After the subtasks execute and produce a result for the message, results are added back onto the SSCF Messages *Payload* Property, the creates subtasks are destroyed by the Garbage Collected given they are short lived objects.

4.3.10 Subtask Profiler

As mentioned in chapter 3 an off-line subtask profiler has been developed with the purpose of assess the processing duration costs and accuracy of each subtask within the application. For the purposes of subtask profiling, all subtasks within the application were executing within a single container in SSCF off-line to generate both accuracy and message processing duration metrics. Additionally, the SEP as mentioned in chapter 3 was also generated after profiling activities had completed.

The subtask profiler runs the application within a single container with all implemented subtasks to create a baseline set of results. These results contained the average subtask processing duration and average result accuracy when the optional subtask was not included to produce the application results. For each optional subtask this process is preformed. The results for each subtask can be find in section 5.2.7.1 in the next chapter.

After the profiler generates message processing duration times and accuracy measures of each optional subtask, the subtask meta-data, the profiler next moves on to generate the global SEP for the application. This was first described in section 3.4.2 as a structured list of subtask combinations including their average aggregate message processing duration, a time to process a single message in the sample data set as well as the overall estimated average output accuracy contributed to the ideal result of the application.

The SEP and subtask meta-data was distributed to each container via Zookeeper, where each container pulls this information in from Zookeeper on application startup, though the *execution_table* lookup key. Now the SEP was loaded into the local container, the local load classifier could navigate it to select appropriate subtasks for the current local container load.

For instance, in our implementation, subtask 1, 2, 3, 4 and 5 resides in the container within PN_1, five of the nine optional subtasks in the applications. In total the Twitter search application was constructed with 13 subtasks. The global SEP was loaded into PN_1 internal container on application startup, taken from Zookeeper. Using the equation 3.1, the total number of combinations here are 2^9 which equates to 512 entries in the SEP table.

The SEP was generated using a bit counting function seeded with the total possible combinations as profiled by the Power Set calculate to generate all possible combinations quickly. This process took on average of 5ms to generate the SEP. The SEP generation function can be found in section 3.4.2. Subtask 2 titled *Word Filter*, subtask 3 titled *Calculate Word Count*, subtask 12 titled *Normalisation* and subtask 13 *Rank* were defined as mandatory and as such were not part of the power set calculation. These mandatory subtasks were added to each entry in the table given they are mandatory operations and must run upon each message. A full list of subtasks are outlined in section 4.6.1 later in the chapter.

Although the container here does not contain subtasks outside of it's allocation, it can only execute subtasks which have been allocated to it locally. This may mean that it correctly selects an SEP entry for which only subtask 2 and 3 given they are mandatory and thus no optional subtasks execute upon the message. The message was then updated with subtask 2 and 3 before it is added to the output queue for processing in the PE. As mentioned in section 3.4.2 in chapter 3 the rational for this was to allow the load classifiers the greatest possible options for selecting subtasks combinations either locally or remotely rather than curtailing the SEP to a smaller set of entries.

The proof of concept application outlines in section 4.6 the results of the subtask profiling activity can be found in section 5.2.7.1 in the next chapter.

4.3.11 Service Discovery and Configuration Management with Zookeeper

For service discovery and to determine the location of distributed SSCF Containers, we have integrated a Zookeeper master [45] as part of the framework. Zookeeper is an Apache Open Source project, which provides services for leader election, key value store storage, service discovery and location services amongst others. Its role in SSCF is primarily to discover and locate each container as deployed in the network, as well as to provide a means to share configuration and

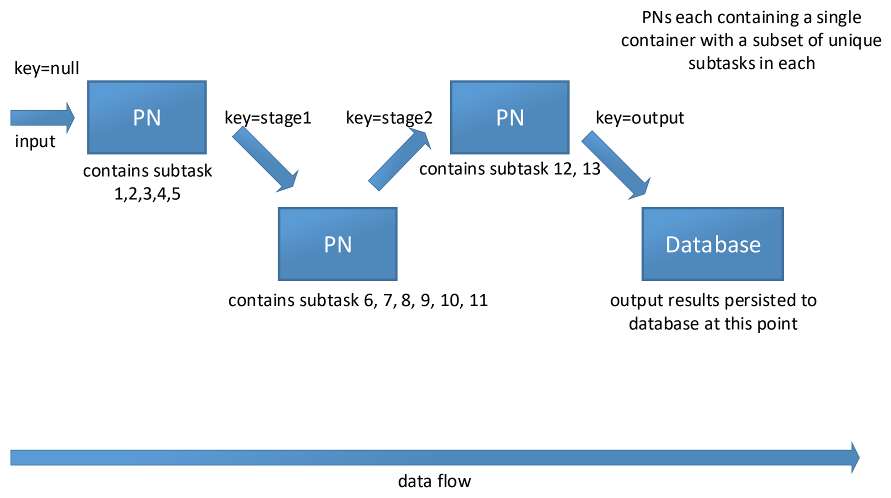
PNs each containing a single container with a subset of unique subtasks in each
key=null
input
PN
contains subtask 1,2,3,4,5
key=stage1
PN
contains subtask 6, 7, 8, 9, 10, 11
key=stage2
PN
contains subtask 12, 13
key=output
Database
output results persisted to database at this point
data flow

Fig. 4.2: An architectural diagram of our Twitter search application, showing the allocation of subtasks to PNs and containers. Also shown are the lookup keys used to locate upstream containers in the output stream flow from downstream containers. These keys are registered and de-registered in the Zookeeper Master on service startup and shutdown. Keys are registered in the application at design time.

subtask profiling and the generated SEP via it's key value lookup mechanism. Each container registered itself with Zookeeper on startup. On shutdown, containers de-register themselves from Zookeeper and removed themselves from the application. Communications to Zookeeper was facilitated using The Apache Zookeeper .Net Client [59].

Registration and de-registration is important here. When a PE to emits a message via it's output stream, it must understand where this message must go. To enable this, we periodically preform a key based service discovery lookup against Zookeeper to determine if there are other containers in the system wanting to consume the message via their input stream, via their input adapter. A temporary cached list of key mappings to ip addresses are maintained within each container. This list is updated every five seconds to get the latest entries in the Zookeeper ZNode for the key. The keys are designed as part of the application at design time.

The key structure for our application coincides with the structure of our PNs and containers. Figure 4.2, describes both the high level architecture of Twitter real-time search and rank based application, as well as the allocated subtasks to containers.

4.3.12 Section Summary

In section 4.3, we described the implementation of the major components in SSCF. SSCF was implemented as a distributed stream processing framework built on the .Net Core framework. The framework was built to aid in the development of stream based applications which require load handing functionality as their core.

The subtask profiler was described in section 4.3.10, in which each individual optional subtask in the application was measured in both terms of message processing duration and result accuracy when compared against a baseline result set. This information was then used to generate a subtask execution plan globally and locally within each local container, such that combinations of subtasks are generated which produce a processing time per message and estimated accuracy target towards the ideal application result.

Finally, load handling has been implemented with specific types of load handlers, which determine the current capacity of the system and if an overload is detected, initiate an adaptation in the processing of incoming messages within the application, specific to each load handler type. The next section covers the implemented load shedders and load classifier in more detail.

4.4 Load Shedders

In this section we describe the implementation used for the three types of load handers used in the thesis. This includes four different implementations of load classifiers which show the implementation of CS, with their selection of subtasks at runtime. The *Load Handler Class* is an abstract base class within SSCF which allows specific load handling functionality, applied to input messages as they are received in the input stream. The next three sections describe the load handers which extend the base class to manage application overload at runtime.

4.4.1 Random Load Shedding

The random load shedder preformed data load shedding on the input data steam prior to messages flowing into the PE. The load shedder received input data from the data adapter via it's shared input queue with the adapter. Messages which were not shed were sent to the output queue.

The messages were then consumed by the PE in it's input stream handler and processed in the application.

The random load shedders implementation is based on work by [111, 113, 116, 118]. In this work, at periodic intervals, the total volumes of messages were accounted for in the system as well as the executed aggregate amount of CPU cycles required to process all messages in the system. If the aggregate CPU cycle count was greater than the total CPU cycle count available, random input messages were shed as a result. The volume of messages to shed was based on the overrun of current total CPU cycle capacity, subtracting the total required CPU cycles required.

The selection of which messages to discard in the load shedder was determined by the probability of a weight toss coin. The weighting for this probability was based on a drop predicate for the overload amount of the system as outlined in equation 4.1. Messages were randomly discarded until the current total CPU capacity of the system was less than the total available CPU capacity.

The random load shedder as proposed by Tatbul was modified to fit within the design constraints of SSCF. SSCF does not measure CPU capacity directly as mentioned in section 3.4.5. The implementation of this detection was described in section 4.3.6 where load was measured by taking a measure of the average processing duration of all messages in the local container. If this processing duration cost was greater than the local containers allocated processing deadline and adaptation of the data stream was applied locally.

When an overload was detected and an overload amount generated for example, the random load shedding algorithm mandates that 10% of the system resources must be recovered to avoid uncontrolled data loss. Thus a random 10% of inbound messages were discarded. The implementation in Aurora and Borealis outline a drop predicate based on a Bernoulli Distribution [111]. We outline the equation used to determine random load shedding selection drop predicate as it was implemented in SSCF.

$$p = 1 - (x - 100 * (m/x)) \tag{4.1}$$

where P is the drop predicate value, x is the drop amount such that the volume of data dropped is enough to stave off overload effects and m is the lowest cost to preform a drop such that the volume of data selected to be dropped must outweigh the cost of dropping this data at this point.

For example, assume that the cost of processing a message is 1ms while the cost of dropping a message is 2ms, the lowest value for m accepted was 2ms such that the selection would have a lower limit of 2 messages.

Once the overload period has passed, import data flow returned to a level where random load shedding was no longer required.

4.4.2 Semantic Load Shedding

Our semantic load shedder implementation was also based on an approached suggested by Tatbul et al.[111, 113, 116, 118]. In this case, during overload input messages dropped were based on the perceived messages utility to the applications function. The approach suggests to use semantic filters to drop input messages with increasing application utility to offset system overload. The approach suggests the use of an ordered list or histogram of utility frequencies, sorted in order of importance. A pointer into the histogram bins indicated which bin or set of bins to drop at any point in system overload. As load on the system increases, the pointer moves up the histogram to other bins which indicates to drop more and more input messages. Each bin indicates higher utility to the application.

For example, if the system determines a 30% overload, 30% of the message must be dropped. If enough data is contained within bin 1 of the histogram, drops of message characterised by this selection occur on new messages as they are received by the load shedder. A pointer moves from bin to bin selecting more and more higher utility messages to drop until the number of messages accumulatively selected equates to 30% of the total messages within the load shedders view.

Our filter for this approach is based on word count in a Tweet as it pertains to one of the metrics which was used to Twitter Search application described in section 4.6. Tweets with higher word count are deemed more important than Tweets containing lower word counts. Given that Twitter had a limit of 140 characters, a histogram containing 140 bins was created. An allocation of frequency counts was attributed to each bin based on the count of words in a Tweet text which created a distribution of message word counts over time within the histogram. The load shedder preformed this classification as each new message arrived.

At any given point in time, t number of Tweets reside in b number of bins. With this model we can then discard input messages based on utility at a proportional amount to the overload.

For example if the application is 10% overloaded and 10% of messages were contained in the histograms first bin, ie: having a low word count, then new messages which were received and had word count classified by the first bin were discarded in the load shedder.

Should the overload volume increase to 40%, then bin 1, and any other bin, up to the point of having 40% of messages received in volume were dropped. This accounted for dropping messages with higher application utility but "saves" messages with higher word counts or higher application utility from being shed.

As the overload period passes, the histogram bin selection gradually drops to correlate with the overload amount. When the overload period has passed entirely, ie: the system is not overloaded further, messages are processed as normal without the need to shed further. The histogram bin pointer returns to zero.

Overload detection followed the same implementation outlined in random load shedding using the approached outlined in section 4.3.6.

4.5 Load Classifiers

In this thesis, four load classifiers were developed which contain the implementation of the four CS designs as outlined in chapter 3. Each load classifier was built on top of the *Load Handler* Abstract class in SSCF. Each implementation searches the respective local SEP and assess local container load (ie: the volume of messages in the PE and within both local queues) within the *Task HandleStreamAsync()* abstract method. Each instantiated load handler must implement this method with it's desired functionality.

4.6 Realtime Twitter Search Application

To evaluate our proposed design as outlined in chapter 3 and in section 4.3, we have developed a real time stream based application to search and rank a Twitter data stream. Our motivation to select this application can be found in section 4.2. Briefly, Twitter is a classic data stream application which was a good fit in this thesis to represent a real world application given the time based semantics and causal ordering of the data stream it produces.

In our application a user enters a search phrase at application start up. The search phrase is used to search a Twitter Data Stream for relevant Tweets. When a relevant Tweet was found in the stream it was processed by the selected subtasks, ranked and added to the sliding window within the application. In all, 13 subtasks were implemented to give a level of complexity to the application and to the number of optional subtask permutations in the SEP.

The contents of the sliding window were reported every second of application execution time. Tweets which do not fulfil the requirements of the subtasks are not processed and are simply discarded.

As mentioned previously the sliding window implementation here was stateful, such that the sliding window contains an ever growing set of input messages accepted by the applications subtasks.

With this approach, the contained Tweets which fulfilled the requirements of the 13 subtasks are maintained within the sliding window, where they were ranked and persisted via the output stream of PN_3's container as outlined in figure 4.2.

The ranking of each Tweet was based on a set of ten metrics intrinsic to the data contained within the data stream. A Tweet contains approximately 37 different properties as mentioned in section 4.2. For our purposes of subtask selection variety, ten metrics introduce variety to allow for different subtask selections at runtime. All subtasks developed in the application are outlined in section 4.6.1.

Name	Description	Data Type
TweetId	Unique Twitter Identifier per Tweet	string
Text	Parsed Text of the Tweet	string
RetweetCount	Tweet Retweet Count	int
VerifiedCount	User has a verified account	bool
FavouriteCount	User favourite tweet count for other tweets	int
FollowerCount	User follower count	int
ListedCount	Number of public lists user is in	int
StatusCount	Number of updates user has posted to Twitter Platform	int
WordCount	Number of works in the Tweet	int
QueryPosition	Position of search words in the Tweet	int
QueryFrequency	Frequey of search words in the Tweet	int
RankPosition	Ranked positon of Tweet in sliding window list of Tweets	int
RankValue	Normalied rank value in sliding window	float

Table 4.4: As part of the application, the SSCF message is extended to contain extra application properties needed to process a Tweet and rank it.

Of the 13 subtasks which constitute the search and rank application, there are nine optional subtasks and four mandatory subtasks which produce a ranked set of Tweets, ranked relative to each other by means of the selected subtasks. The results of which are contained within the sliding window.

The next section briefly describes the implementation of each of the subtasks which constitute the application.

4.6.1 Implemented Subtasks

Subtasks are implemented by extending the subtask base class and providing application specific functionality upon each message within the overloaded execute method. Subtasks also have access to the sliding window should they require access to all messages currently within the sliding window. Our proof of concept application contains 13 subtasks which operate on each message received in the PE.

Id	Subtask Name	Mandatory	Optional	Description
1	WordNormalisation		●	Normalises Tweet text to Unicode
2	WordFilter	●		Filters Tweets based on search query
3	CalculateWordCount	●		Counts the words in a Tweet text
4	CalculateWordPosition		●	Counts position of query works in Tweet Text
5	CountQueryFrequency		●	Counts search query word frequency in Tweet Text
6	FavouriteCount		●	Parses and counts users favourite Tweet Text
7	ListedCount		●	Parses and counts users listed count
8	RetweetCount		●	Parses and counts Tweet retweet count
9	Verified		●	Parses and asses if a user is verified
10	FollowerCount		●	Parses and counts the followers of the user
11	StatusesCount		●	Parses and counts users activity on Twitter
12	Normalisation	●		Normalises all other subtasks outputs
13	Rank	●		Ranks each Tweet in the Sliding Window on the normalised value

Table 4.5: The list of subtask names used in the application as well as their ids and an indication if the subtask is mandatory or optional in the application

In the ideal case, when the application is not overloaded, all subtasks execute in sequential order within each PE. The input message is updated with each subtasks output before being passed to the next subtask in the chain. After all subtasks have completed their work in each container, a checkpoint of the sliding window was created every second.

The processing order of subtasks here was important, in some cases. For example, all optional subtasks can process the message in any order. However, mandatory subtasks such as the normalisation and rank subtask, require other subtasks to produce a result first. Or else the relevant data they require will not be present within each message or within the sliding window. Further without a ranking subtask, there would be no ranked output result of the application. The ranking subtasks input was dependant upon the output result of the normalisation subtask. While the normalisation subtasks input was dependant on other optional and mandatory subtasks. This was the reason both the normalisation and ranking subtask are declared mandatory and defined in the SEP in the order of normalisation first and ranking second.

Next a brief outline of each subtask is presented. A common trait of each of the subtasks was to extend the *Subtask Base Class* and provide an implementation within the *ProcessMessage* method. Here each subtask receives the base message from SSCF which can be cast to the appropriate application type. The type of the case message in table 4.4. After the subtask preforms it's expected work upon the contents of the message, the output state was stored into the message itself before the message was passed to the next subtask in the chain.

The last subtask to process a message was the ranking subtask, after it's state was saved, a checkpoint of the application was made in the sliding window waiting to be check-pointed to the database.

The following sections briefly describe each subtask;

4.6.1.1 Subtask 1 - Word Normalisation

The Word Normalisation subtask normalises the Tweet Text to Unicode given that the message is generated from a third party service and thus needs to be normalised before it can be parsed. Unicode characters have multiple equivalent binary representations consisting of sets of combined and/or composite Unicode characters. Word Normalisation was performed to prevent both false positives and false negatives in comparison between input search query and Tweet text. The subtask additional parsed the Tweet and retrieved the Tweet text object for normalisation and saved it back into the input message for processing by the next subtask in the chain. This was preformed using the standard C# string normalisation functions [93].

4.6.1.2 Subtask 2 - Word Filter

A word filter subtask was implemented to parse out Tweet text and filter out Tweets from the data stream which do not contain any reference to the input search query. The search query was entered at application start. First the task tokenised the input stream text, and compares each word in the string array to the words entered in the search query. If a valid comparison was found, the Tweet and message were allowed to proceed to the next subtask. If no valid comparison was found, the Tweet and message was rejected and was not processed by any other subsequent subtasks.

4.6.1.3 Subtask 3 - Calculate Word Count

In subtask 3, the Word Count subtask parsed the tokenised Tweet text and updated a counter for the number of words in the Tweet text. Tweets with higher word counts were assumed to have more information than Tweets with lower word counts and as such were assumed to be more important. [95]. This count was stored on the message before being passed to the next subtask.

4.6.1.4 Subtask 4 - Calculate Word Position

Subtask 4 function was to calculate the lowest position of words in the Tweet text which contained the input search query. The lowest position of any word contained in the query string was retained within the message after discovery. The assumption here was that Tweets with the search text appearing closer to the start of the Tweet text, have a higher value than Tweets where the search text appeared further from the start.

4.6.1.5 Subtask 5 - Calculate Query Frequency

Subtask 5 purpose was to calculate the frequency of each occurrence of any of the search words in the Tweet text. The assumption here was that Tweets with higher frequency of the search query terms within the Tweet Text, may be more important than Tweets which have lower frequencies of the words. The output value was updated on the message after it was computed.

4.6.1.6 Subtask 6 - Tweet Favourite Count

Each Twitter Tweet contains a user meta-data child JSON object which contains meta-data about the user and their actives on the Twitter Platform. The favourite count is a measure of the users engagement within the Twitter platform. For our purposes a higher favourite count indicates a user which was more engaged with the Twitter platform and produced Tweets with higher value than users with a lower favourite count [65]. The purpose of this subtask was to parse the Tweet and to generate this value for each message received. The output value was saved onto the message after it was computed.

4.6.1.7 Subtask 7 - Tweet Listed Count

This subtask parsed user meta-data from each Tweet. Here the listed count pertains to the number of lists a user has been added to by other users of the Twitter Platfrom. The assumption here was that users which are contained in a large number of lists, more than other users have been deemed to be more important by other users of the platform [65]. The output value was saved onto the message after it was computed.

4.6.1.8 Subtask 8 - Retweet Count

Retweets are the act of sharing a Tweet which are created by another user on the Twitter Platform. The premise was that Tweets which have been retweeted more than other Tweets are assumed to be more valuable on the Twitter Platform than Tweets with a lower number of retweets [65]. The output value was saved onto the message after it was computed.

4.6.1.9 Subtask 9 - Verified Count

Twitter has a verification process indicating that the user tweeting is a person of public interest. Twitter defines this as “An account may be verified if it is determined to be an account of public interest. Typically this includes accounts maintained by users in music, acting, fashion, government, politics, religion, journalism, media, sports, business, and other key interest areas.” Based on this assumption a Tweet with user meta-data which has been allocated the verification badge was deemed to be more important than Tweets from users which do not have the verification badge. Messages which have a verification badge were attributed this value and saved onto the input message in the application [65].

4.6.1.10 Subtask 10 - Follower Count

A user on Twitter can "follow" another users Tweets and activities. Given the large volume of data streaming through Twitter a user does not get a view of all data on the Twitter platform, but rather is given a small subset of data at any point in time. A user can indicate their interest in certain types of Tweets by subscribing or "following" a user.

This is an indication to Twitter that the subscribing user wishes to see more Tweets from the subscribed user. From this we assumed that a Tweet from a user with a large number of followers has more importance than Tweets from a user with a lower number of followers [65]. This JSON property was parsed from the Twitter Tweet contained within the receive SSCF input message by the subtask and the follower count value was added to the message before it was passed to the next subtask in the execution plan.

4.6.1.11 Subtask 11 - Statuses Count

The "Statuses_Count" value is available on the User Object within the Tweet and it gives a count of all Tweets including retweets which the user has issued with the Twitter Platform. Users with higher status counts are assumed to be more active on the platform and thus may produce messages which are of higher value than users which have lower statuses counts [65]. Similar to other implemented subtasks which parse the User Tweet Model, this subtask parsed this property and saved the value onto the SSCF message.

4.6.1.12 Subtask 12 - Normalisation

This subtasks purpose was two fold. First, all metrics contained within the message generated by other subtasks were normalised within each metric ie: weighted against the upper bounds of the highest valuing metric of it's type. Second, all generated metrics must be normalised across one another. Allowing this subtask to compute an aggregate score for each message or in this case, each processed Tweet. This aggregate score was then used to rank each message against one another, which was the function of the rank subtask.

Each individual metric saved onto a messages was normalised if the optional subtask generating it was active in the application at the time. It may not be active due to the load classifiers decision to turn it off for a time during bursty data periods. For values which were not present in a message, the value was not computed for that message. This may mean that some messages in the sliding window have difference ranges, for example, some messages may have four metrics while other messages may have three metrics. For values not computed they were given a zero value. This may skew the results of some messages to have a greater rank value but additional it also means more message potentially have been accepted by the application because the subtask processing time was reclaimed.

The normalisation process works as follows:

1. For each message within the sliding window, determine the minimum or maximum value of the messages ranking metrics within the sliding window. The choice to take the minimum or maximum was based on the semantics of the metric. For example, in the case of word count, a higher value was deemed to be more important than a lower value. As such, we take the maximum count value between all messages within the sliding window for that metric.

2. For each metric a value was generated which represented the normalised value of this metric. To produce a normalised value between 0 and 1. This put all normalised metrics on an equal footing to allow them to be aggregated together per message.

3. Store each of the normalised values on the message and store the maximum or minimum values on the sliding window.

4.6.1.13 Subtask 13 - Rank

Finally, we arrive at the ranking subtask which has a function to aggregate each message's normalised metric values to produce a single ordinal rank score. When all subtasks produced a result, this accounted for nine individual metrics used to rank a Tweet against another set of Tweets within the sliding window. When message ranking scores collide, a fractional ranking methodology was used to rank the messages. The ranked score was added to each message in the sliding window.

4.7 Summary

In this chapter we described the implementation of Computational Shedding within SSCF. We discuss the major components of the SSCF framework, such as sliding windows, containers and subtask profilers. We also discuss our proof of concept Twitter search application, which consumed and ranked Tweets in real-time based on an input search query. The aim of this was to show how to implement Computational Shedding within the context of a real world application.

We describe the implementation of each load handling mechanism, namely random load shedding, semantic load shedding and our proposed alternative load classifiers which selects appropriate sets of subsets at runtime in an effort to forgo data loss in overload situations while continuing to produce an output result in the application.

Finally, we describe each of the subtasks developed to produce this application.

In the next chapter we outline our evaluation strategy and methodology to better understand the benefits and drawbacks of Computational Shedding compared to Random and Semantic Load Shedding alternatives.

Chapter 5

Evaluation

5.1 Overview

This chapter contains the evaluation details of the experiment undertaken to evaluate our proposed computational shedding approach with SSCF. Contained within it, is the experiment methodology we use to evaluate CS and it's implementations and the results of this experimentation. We evaluate our proposed approach by experimentation, in which we observe, compare and contrast the processing duration and output results of a real world application. The application in question is a Twitter Search Application built to search and rank Twitter Tweets in real time based on an input search query supplied at application start.

First, an outline of the experimental setup is provided, along with the experimental parameters chosen to run the experiments.

Next, we outline the experimental comparison methodology for the applications results. Given result accuracy tends to be application specific, we provide a detailed account of how results were measured and compared in each of the load handlers chosen for the evaluation.

Finally, the results of our experiments are discussed before the chapter is summarized.

5.2 Evaluation Setup

This section outlines how our experiments were setup. First we describe our objective of this evaluation. how data was collected from Twitter which was used as the basis for our experimental analysis in section 5.2.2.

Next we describe the deployment of PN in Azure, Microsoft's Cloud Platform, where our experiments were executed in section 5.2.3. Further, we describe the experiment distribution model explaining the variety of data rates used to stream the collected Tweets into the deployed application and explain how this occurred in section 5.2.4.

Finally, we outline the measured outputs of the experiment in section 5.2.6, in addition to outlining the experimental parameters in section 5.2.7 chosen to run the experiments with. For example, the input search query, queue total capacities and data rates used to run the experiments with.

5.2.1 Objective

Our objective in this evaluation is to determine if CS can be considered as a viable alternative to classical load shedding approaches in terms of end-to-end processing duration and application accuracy (or how close an output result was to the ideal application output). Further, we aim to measure the improvements made in the four variants of CS as outlined in chapter 3 through their effect on a proof of concept application.

Further, we aim to understand the limitations of the CS approach, in terms of its effects on queue capacity during overload, the duration of each load handlers execution time, as well as subtask execution plan selection times. For all scenarios evaluated, we wish to compare CS and it's variants against existing load shedder implementations, and determine whether our approach exhibits advantageous characteristics.

5.2.2 Data Collection

As mentioned in chapter 4 our evaluation of CS is based on observations made with simulated real time Twitter search application. To facilitate this methodology, data from Twitter, in the form of a set of three million Tweets were collected from Twitter Public API or Firehose [66] and later stored in Azure Storage in a serialised JSON format.

Many researchers have used the Firehose in the past for their research into interesting areas, such as social computing, emergency management, determining political election appointments and stock market trading speculation example [64, 70, 86, 99, 103]. Although the Twitter Firehose API is public, the free tier serves Tweets at a rate of 1% of all Twitter Tweets sampled at a random rate. Paid services allow for a greater level of consumption from Twitter. While 1% is valid for evaluation in this thesis, the data rate is not enough to overload modern day computers and applications. Further, to ensure a valid comparison between load shedding and computational shedding implementations, replay of the same data is required to create a valid statistical comparison.

This large data set, consisting of three million Tweets was broken down into 30 subsets containing 100,000 Tweets, each stored as a serialised JSON structure. An experimental run in this thesi, can be described as serving one of the subsets consisting of 100,000 Tweets at a requested data rate into the deployed application until all 100,000 Tweets have been processed.

5.2.3 Deployment

The developed application was deployed to Azure, Microsoft Cloud Platform. Figure outlines the deployed application. Each node in the cluster was deployed as a *D2 v2* instance size, which gives each node 2 CPU cores, 7.00 GB of memory and 100GB of local disk storage at a price of €0.196 per hour of use [38]. In total the application was deployed across three virtual machines, an Infrastructure as a Service (IaaS) offering in Azure. IaaS provides virtualised cloud computing resources. Each of the machines communicate with one another with service discovery facilities provided by Zookeeper as outlined in chapter 4 section 4.3.11. Node 2 in the network was allocated as Zookeeper master. A visualisation of the deployment as well as the allocated subtasks to nodes can be found in figure 4.2 in chapter 4.

A single MySql Database [36] instance was deployed in Azure to retain application and service output results. From there, node 3 populated the database with each sliding window output from the application on a per second basis from it's output stream handler. The data was then downloaded locally soon after the experiment was run for analysis. Further details on this process are defined in section 5.2.4.

Finally, all data sets used in the experiment which are mentioned later in the next section were saved to Azure Storage [37] and mounted as a remote drive to the first node in figure 4.2. This allowed the first node to consume the data from an abstraction which behaves as a local file system to load the data file by file into local memory before it was served to the application at designated data rates.

Given IaaS resources are charged on a per minute basis, the three instances and database were running only when the experiment was executing. All resources were destroyed outside of this time to remove the possibility of incurring additional monetary costs.

5.2.4 Experiment Distribution

In any real world application, a large distribution of variation in output results can be observed. As was the case when running the application atop SSCF within Azure. Although this thesis is not concerned with their causes some examples are provided to provide context. First, context switching and resources starvation within the local virtual machine. Second, network congestion between PNs causing network delays and perhaps TCP/IP packet redelivery. Finally noisy neighbours caused by other virtual machines on the same physical host consuming local physical resources at a high rate and starving the experiments application [51].

As a consequence of these events, random variations in application measurements reduces the accuracy of statistical tests and can only be overcome by running the experiment multiple times. This is the rational for running each load handler type with thirty individual data sets, as outlined in section 5.2.2.

As mentioned, each data set contains 100,000 Tweets which were served into the deployed application at a variety of data rates. For each data rate selected, thirty data sets were served one by one until all Tweets were processed by the application with the selected load handler. Giving a total of three million unique Tweets served into the application during each experimental run.

In the application, an experimental run took on average of approximately 90 seconds to process 100,000 Tweets when data was served at 1000 messages per second (MPS). In addition subtask profiling, to create the SEPs was also preformed in this application within the same environment.

To facilitate the processing of this vast quantity of data, an automated system was developed to process the data sets at a variety of data rates. The data rates chosen create load on the application below and well above the applications overload thresholds. This equated to an input data rate of approximately 3500 MPS. The threshold was determined by the overload calculation as outlined in chapter 3, but simply put is a proxy approach for within input queues start to fill. The specific data rates are discussed later in section 5.2.7.

As well as running each data set at designated data rates through the application, each experimental run is executed with a selected load handler.

A light weight control application was developed called *Mission Control* which allowed remote control of the deployed application. Communications were facilitated though *Azure Service Bus Queues*, a remote Queue as a Service (QaaS) which provided First in First out (FIFO) queue semantics. The first PN in the cluster would listen to the queue for messages, consume them and execute the requested instructions contained within each message, one at a time.

Missions Control is a command line application which takes command line arguments which are then used as application configuration settings for the application running in the cloud. Configurations include the number of data sets to process, a comma separated list of load handler types to use in each experimental run, the range of data rates to service data at the application and the search query itself to search the simulated Twitter data stream. Messages containing the application configuration data are then sent to the Service Bus Queue from Mission Control.

When the application starts messages are consumed from the *Service Bus Queue* containing the application configuration settings. Messages are received by the first PN in the cluster where it was executed, one at a time. For example, the PN might receive a message to load data set 4, running at a data rate of 1000 MPS using a random load shedder. For each data set, data rate and load handler type, a message was added to the *Service Bus Queue* such that for 10 data sets, 10 data rates and 4 load handler types, 400 messages would be sent to the queue from Mission Control. Each message within this queue was processed one at a time in each stage of the application pipeline. Additionally each message included a unique id which represents a deterministic value for the experiment run so it can be identified later. The configuration data was added as metadata to the SSCF message for distribution to all PNs in the cluster within the first message sent to each PN.

When a full data set was processed by the file loader which loads the JSON files off disk containing serialised Tweets, the last message sent into the application contains a bool value to indicate that this is the last message in the data set. As this message is processed by each PN and there in each load handler and PE, the asynchronous threads running in each component stop to allow the application to end. Each PN then resets, ready to receive the first message again for processing.

Mission Control contains a command line argument to generate application results. The R language along with R Studio were used to create a set of scripts to generate the set of graphs outlined later in section 5.3. With the deterministic id for each experiment mentioned earlier, Mission Control can check the database to determine if the application had completed it's requested work, processing all messages. If not, a message was presented on screen, showing the state of the last 10 experimental runs. If the application was complete, Mission Control, would take a MySQL dump of the contents of the remote database and restore it to a local database on the client machine where Mission Control was executing. From here the set of created scripts were run to preform data analysis on the results for each data rate and load handler type to generate a statistically accurate picture of the measured metrics. The diagrams were then saved to disk in pdf format for later analysis and are presented later in this chapter.

5.2.5 Evaluated Load Handlers

As outlined in chapter 4 an evaluation is provided in relation to a number of load handler types within the context of the Twitter Search and Rank application.

The computational shedding (CS) approach was implemented within four implementations, one for each version of CS outlined in chapter 3:

- A load classifier with a SEP ordered by descending order of accuracy to the expected applications ideal result, per the design of CS-A. This will be known as LC-A.
- A load classifier with a SEP ordered by descending order of processing duration to produce a result in the application. This will be known as LC-D.
- A load classifier which reduces the cost of determining current load and selection of subtasks at runtime per the design of CS-C. This will this be known as LC-C.
- A load classifier which attempts to reduce the searching time to select appropriate subtask combinations in the SEP by using a binary search algorithm per the design of CS-B. This implementation will be known as LC-B.

Also, as outlined in chapter 4 two load shedder types have been implemented and evaluated:

- A random load shedder which randomly selects which input messages in the data stream to drop. This load shedder will be known as LS-R.
- A semantic load shedder which selects which input messages within the data stream to drop based on semantic value to the application. Here the semantic value of a Tweet was

measured in terms of the word count contained within it. Such that higher word counts were more important than lower word counts. This load shedder will be known as LS-S.

Multiple CS implementations were chosen to best understand the positive and negatives approaches when compared to one another. A summary of the evaluated load handlers along with a short description of each is given in table 5.1.

Load Handler	Overload Handling Method	Description
Empty Load Handler	N/A	Implemented to provide a baseline method of obtaining ideal results in the Search and Rank Application. It preforms no load handling duties.
LS-R	Message Discarding	Random Load Shedder which works at the message level to discard messages randomly at a proportial rate to precieved application overload.
LS-S	Message Discarding	Semantic Load Shedder which works at the message level to discard messages based on a semantic utility to the application. Retains an in memory distribution of messages to asses utlity and value of drops.
LC-A	Subtask Discarding	Load classifier with a subtask exeuction plan order by accuracy in comparion to a baseline benchmark. Exeuction plan is order in in descending order, optimised for accuracy selection.
LC-L	Subtask Discarding	Load classifier with a subtask exeuction plan order by latency in comparion to a baseline benchmark. Exeuction plan is order in in descending order, optimised for latency selection.
LC-C	Subtask Discarding	Load classifier which reduces the cost of measureing current application load coupled together with LC-L implemenation to optimise for message processing time reduction.
LC-B	Subtask Discarding	Built upon LC-C with a reduce cost of measureing current application load, the search time in the subtask execution plan can be reduced using a Binary Search Algorithm.

Table 5.1: The suite of load handlers evaluated in this chapter along with a short description.

All load handlers were evaluated under the same conditions as outlined in section 5.2.4. Further a common set of measurements and metrics were used to evaluate each load handler within the confinement of the application as outlined next, in section 5.2.6.

As outlined in chapter 2, we expect LS-R to be outperformed by LS-S in terms of accuracy because of it's semantic view within the application. However, the bin selection mechanism may lead to dropping more data than is required as a results of the histograms bin distribution.

As outlined in chapter 3, CS asses the current local container load and selects an entry in the local SEP in a best effort to address the available time to process data and thus meet application duration requirements. We expect LS-A to take the longest time to select the most appropriate execution plan entry as it iterates through a list of subtask combinations which are not optimised for processing duration. LS-L should have good responses to application processing duration required but at the expense of application accuracy.

LC-C should preform better than LC-A and LC-D in terms of both accuracy and processing duration given the cost of the selection process is reduced coupled with the subtask execution plan being ordered by message processing duration, however this is again at the expense of application result accuracy. However given that subtask selection is essentially faster this should provide a greater level of accuracy than both LC-D and LC-A.

LS-B should preform the best of all consider that the selection process is an improvement upon LC-C and thus should provide better results than LS-L and LS-B. However this may not produce the most accurate results compared to LS-A with it's SEP focused on accuracy.

We expect that CS will out preform both classic load shedders in application processing duration requirements given the reduced computational requirements to process each message as subtasks are discarded to offset increased input message volumes. However, we expect this is at the expense of application accuracy given the load shedders will be processing with all 13 subtasks for each message that is accepted. With LS-S we expect this to produce the most accuracy application results when compared to the baseline given the semantic utility of the shedder is also one of the ranking metrics of the application. A decision to implement this as such was to give a fairer comparison in LS-S to other load handlers.

Finally, given CS cannot discard all subtasks as the application is limited to only discarding optional subtasks, four mandatory subtasks will remain. Our expectation is that there are benefits to the approach up to the point where all optional subtasks have been discarded but just the mandatory subtasks remain within the application, from this point on, no more adaptations can be taken. After this point, queues volumes begin to rise and message queueing times will increase.

5.2.6 Measured Outputs

The key measurements we aimed to gather during this experimentation related to both application end to end processing duration and application output result accuracy.

Stream computing applications must attempt to meet their required message processing duration deadlines, as should a message arrive late, it's value is efficiently zero. Additionally, if a system becomes overloaded and load handling activities are engaged to reduce the effect of that load, the result of the subsequent application operation must be of value. Application accuracy is very much application specific and to this end, what application accuracy means in our proof of concept application is outlined in section 5.2.6.1.

In stream computing both metrics are very important to a load handlers effectiveness. A load handler should ensure an application can continue to meet the application required processing duration as well as attempting to keep application output result accuracy high in light of data or application level adaptations. Thus we measure both here.

Further, we assess the processing duration of each load handler type to infer how long it can take a load handler to decide what it will do with the stream. Slow load handlers can cause an adaptation which is otherwise slower than one may like and thus a system which modifies it's behaviour as fast as possible is preferred to fend off late application results. For example in a load classifier there are a list of subtask selections to navigate in the subtask plan, this can take time particularly in cases where the system is extremely overloaded and a large list is iterated through in it's entirety for each message processed.

Finally, queue data retention is also measured. This is important to gain an understanding of the effects each load handler type as on a data stream and how long messages may remain in the application queues during overload.

5.2.6.1 Application Accuracy - Spearman Footrule

Accuracy is a measure as to how close a value is to an ideal result and very much intrinsic to the context of the application. In our case, pertaining to ranked sets of Tweets, accuracy is defined as how close one ranked set of results is to the ideal baseline result set.

In any ranked set, order of items is important such that items at the top of the set are more important than items lower down. To that end, Spearman's Footrule (SF) [74] was used to measure the displacement or movement of items produced in the application in one ordered set compared to the position of the item in the ideal ordered set produced by the baseline[75]. For every sliding window, the aggregate displacement was calculated for all tweets to give a total displacement. The decisions of the load handler at the time of sliding window creation determined the order of the items in the output application result. Or in the case of load shedding, determined if a valid Tweet would make it into the sliding window or not, given it may be shed.

Displacement and Spearman's Footrule are defined by the following equations:

$$\sigma = |i - \sigma(i)| \tag{5.1}$$

Spearman's Footrule for total displacement is calculated as:

$$F(\sigma) = \sum_i |i - \sigma(i)| \tag{5.2}$$

In our case, for subtask profiling, a measure of total displacement for each optional subtask was generated to give an approximation as to it's accuracy value in the applications ranking algorithm. Using one sample data set from the experiments data set, the average displacement for each subtask was calculated by running the experiment without that particular optional subtask. Then by comparing the order of each item in each sliding window output to the corresponding sliding window within the baseline case, a measure of average displacement was calculated for every optional subtask. The baseline case produced sliding windows with output results which were calculated using all subtasks, both mandatory and optional.

SF requires the compared two lists to have the same items and the order of items in each list can be different compared to one another. All ranked tweets are found in the baseline sliding window. However, in load shedding cases a valid tweet one might expect to be in the sliding window may not be there as a result of shedding. As such from the literature, using an approach titled *Last observation carried forward* LOCF, [72, 9], the approach suggests that when a data point is missing from a result, it can be substituted for using the last observation made. Here to remove undue bias, when a tweet is not present in the output load handler list, we give it a position at the end of the list, rather than no position at all. Using this approach should allow for statistical comparisons such that a value which has been shed, would have a displacement position from a position in the baseline result set. Obviously this has little value for a user observing application output results but for the purposes of measuring accuracy between approaches, this methodology should be sufficient.

To illustrate what this result would look like take for example a sliding window contained 10 items in the baseline case and in the profiled case it contained just 9, the missing item would be given a position of 10 and so on, such that the count of items in the output sliding window would be equal to the count of items in the baseline sliding window list.

As mentioned in section 3.4.5 this methodology was also used in subtask profiling to measure the accuracy effects each subtasks has on the result of the application. However, given that no data was shed, substituted values were not used.

5.2.7 Experimental Parameters

SSCF, proof of concept application and load classifiers must be configured with a number of parameters before the application can be executed. This section covers the experimental parameters used including samples of the subtask execution plan generated and used for subtask selection at application runtime.

Table 5.2 outlines the experimental parameters used in the application and infrastructure to run the experiments. Queues play an important part in the SSCF framework however here their underlying implementation in .Net, given the queues are linked-lists, connects the queues directly with available memory on the underlying machines the application is executing on. Further as the application and framework are executed on 64-bit architectures as well as the size of the data sets used, the applications sliding window limit and the processing speed of data in the system, the queues have more then enough memory to fulfil the requirements of the application.

To measure the output results of the load handlers in the system, a variety of data rates are used to induce load and overload volumes of data within the application. During profiling it was observed that the application can process approximately 3,500 MPS without being overloaded. As such, the data rates used to induce load are below this rate and well beyond this data rate such that load on the application can be observed up to 10 times this load. This gives a good range of application and system load to see how each load handler operates at increasing input data rates.

Table 5.3 outlines the load handler parameters used in the experimental application. In the case of LS-R, the probability of drop selectivity is derived firstly from prior work, as a weight toss of coin, weighted by current system load.

LS-S selectivity was based on the utility of word count in the application. Given the application choice a measure of works in the application is a good selector for semantic value. Further the choice of 140 bins within the internal histogram gives a good level of selectivity range, such that the selection algorithm has a good range of choice in determining which bins to drop as load on the system increases. A small bin selection can lead to the possibility of over dropping incoming messages which would not lead to a fair comparison in the upcoming results.

Regarding the load classifiers, thirteen subtasks of which nine are optional give a good variety

Category	Name	Value	Description
Infrastructure	Queues	7GBs Azure VM	Based on .Net ConcurrentQueue<T> which are based on a linked list implementation thus are limited by available main memory on the hosted machines in Azure Cloud.
	Containers	1	The number of Containers allocated to each Processing Node in the application.
	Nodes	3	The application has been deployed across three nodes with subtask allocation for each.
Application	Search Criteria	"Just in Time"	Arbitrary search criteria used to filter, search and rank input Tweet messages against in the application.
	Subtasks	13	13 subtasks within the application, 4 of which are mandatory and 9 are optional.
	Subtask Allocation	NA	Subatsk 1,2,3,4 and 5 were allocated to node 1. Subtasks 6,7,8,9,10 and 11 were allocated to node 2. Subtask 12 and 13 were allocated to node 3.
	Message time	2 seconds	An arbitrary value indicating the maximum amount of time a message should remain in the system. If message total time is greater than this value, adaptation of the application in some form must occur ie: LS or CS.
	Checkpoint	1 second	The checkpoint time used to create the sliding window in the last container for analysis and visualisation.
Data	Data Rates	1,000 to 50,000 (TPS)	Data rates used to service data at the application at increments of 2000TPS.
	Data Set Size	100,000 Tweets	The volume of Tweets within each data set file.
	Data Set Count	30	Number of individual data files used in the experiment.

Table 5.2: A table of experiment parameters used in the application and their configured values or ranges.

Type	Name	Value	Description
Data Shedders	Random Load Shedder	Weighted Probably based on toss of a coin	Weighted probablity of a message drop, weighted by current system overload value.
	Semantic Load Shedder Bins	140	140 bins selected based on the maxiumum number of characters with each Tweet text.
	Semantic Load Shedder Selection	Pointer to bin	A pointer into the current bin, selection based on current overload amount and distribution of data in each bin.
	Semantic Load Shedder Algorithm	Word Count	Semantic Utility based on word count of each Tweet text.
Load Classifers	Subtask Execution Plan	512	The number of entries in the plan based on the count of optional subtasks. Mandatory subtasks are always included in the subtask plan entry.

Table 5.3: A table of load handler parameters used in the evaluation characterised by shedding operation.

of choice in creating the SEP when aggregating all local SEPs into one table. As can be seen from table 5.3 nine optional subtasks give a possible combination of 512 entries including one additional entry where all optional subtasks are excluded and just the mandatory remain for selection in the execution of the application during overload. A more in depth discussion of the subtask execution plan is provided in the next section.

5.2.7.1 Subtask Profiling and The SEP

To facilitate the selection of subtask combinations to construct local and aggregate SEPs, subtask profiling was preformed on each subtask constructed as part of the experiment. Table 5.4 outlines the results of subtask profiling which include all mandatory and optional subtasks within the experiment. Note, values marked as NA are not applicable as a result of subtasks which are marked as mandatory in the application. These subtasks are critical to the application and cannot be shed, thus a value for fields marked as NA cannot be computed as to generate a value for the field would require the subtask to be removed from the application.

Subtask Id	Accuracy Loss (%)	Accuracy Contribution (%)	Average Processing Time (ms)	Displacement (Spearman)	Messages Processed
1	99.9808	0.0191698	0.00819911	2	5050181
2	NA	NA	0.0002978	NA	809268
3	NA	NA	0.000012	NA	809268
4	99.9612	0.0388034	0.000472032	5	809268
5	86.1223	13.8777	0.000274322	1795	809268
6	79.391	20.609	0.0103822	2666	809268
7	73.1391	26.8609	0.00757104	3475	809268
8	99.9694	0.0306099	0.00755992	3	809268
9	99.9993	0.0007	0.00768102	1	809267
10	74.5905	25.4095	0.00833841	3287	809267
11	86.8458	13.1542	0.00750432	1701	809267
12	NA	NA	0.000694456	NA	809267
13	NA	NA	0.000925529	NA	809267

Table 5.4: Outlined here are the results subtask profiling for a stateful sliding window in which the results of a search query against a sample data set produced the results. NA values represent values which were not computed given the subtask was designated mandatory and was not excluded from the application.

The profile results presented in table 5.4 feed into the Subtask Profiler which created all possible combinations of subtasks from the available set of optional subtasks. Mandatory subtasks are added to each entry in the plan, given mandatory tasks must execute in the application for each message processed. In table 5.5 and table 5.6 sample sets of entries in the aggregated plans are presented in the order they were generated by the profiler. For each entry there is the subtask plan id, the total accuracy contribution of the plan entry when producing a result in the application. Finally, also illustrated is the average message processing time of the combined subtasks in the plan entry, this was generated from the average message processing time generated per each subtask duration profiling. Messages processed account for all messages processed in each profile run for all optional subtasks.

Both tables represent the aggregation of all local SEPs generated in the application. After the raw subtask meta data is published to Zookeeper after profiling, as outlined in section 4.3.10 in chapter 4, each local container generates it's local SEP based on both the allocated Subtasks to it and this data. What is described here is the aggregation of all SEPs generated for each of the three containers for illustration and discussion purposes.

As can be see from table 5.5 and table 5.6 there are thirteen subtasks in total, of which four are mandatory, leaving nine optional subtasks, which gives a total SEP size of 512 entries generated by the subtask profiler, including the fall back to mandatory subtasks, which for the PowerSet combination is the null set. Entry 512 in the plan includes all subtasks thus it's estimated accuracy towards the applications results is 100% and takes 0.0599 on average to process a message, processed by all 13 subtasks.

For example, entry 191 in table 5.5 contains all subtasks with the exception of *subtask 5* which in the Twitter search application represented the search term query frequency subtasks. Details of each subtask including their identifier (id) can be found in section 4.6.1 in chapter 4. The application with the remaining subtasks was able to achieve a 86.1223% level of accuracy on average when compared to the ideal output result contained within the baseline experiment for this SEP entry.

Table 5.5 and table 5.6 illustrate a sample set of the SEP ordered by processing duration and accuracy outputs respectively. For each table, the first 15 entries are displayed only to provide a sample of trends observed which are described next.

From table 5.5 a general trend can be observed where the average processing duration of the entries in the table reduce while moving from one entry to the next in descending order. Further, to facilitate the processing duration reduction, the included subtasks in the plan as well

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Local Subtasks	Local Average Processing Duration Per Message (ms)	Accuracy (%)	Average Processing Duration Per Message (ms)
512	1,2,3,4,5,6,7,8,9,10,11,12,13	NA	1,2,3,4,5	0.009255	100	0.0599002
191	1,2,3,4,6,7,8,9,10,11,12,13	5	1,2,3,4	0.008981	86.1223	0.0596259
254	1,2,3,5,6,7,8,9,10,11,12,13	4	1,2,3,5	0.008783	99.9612	0.0594282
62	1,2,3,6,7,8,9,10,11,12,13	4,5	1,2,3	0.008509	86.0835	0.0591538
422	2,3,4,5,6,9,12,13	1,7,8,11	2,3,4,5	0.001056	86.8458	0.0523959
167	1,2,3,4,5,6,7,9,10,11,12,13	8	1,2,3,4,5	0.009255	99.9694	0.0523403
189	1,2,3,4,5,6,8,9,10,11,12,13	7	1,2,3,4,5	0.009255	73.1391	0.0523291
163	1,2,3,4,5,6,7,8,10,11,12,13	9	1,2,3,4,5	0.009255	99.9993	0.0522192
192	1,2,3,4,6,7,8,9,10,12,13	5,11	1,2,3,4	0.008981	72.9681	0.0521215
231	1,2,3,4,6,7,9,10,11,12,13	5,8	1,2,3,4	0.008981	86.0917	0.0520659
239	1,2,3,4,6,8,9,10,11,12,13	5,7	1,2,3,4	0.008981	59.2613	0.0520548
227	1,2,3,4,6,7,8,10,11,12,13	5,9	1,2,3,4	0.008981	86.1223	0.0519448
126	1,2,3,5,6,7,8,9,10,12,13	4,11	1,2,3,5	0.008783	86.807	0.0519238
38	1,2,3,5,6,7,9,10,11,12,13	4,8	1,2,3,5	0.008783	99.9306	0.0518682
47	1,2,3,5,6,8,9,10,11,12,13	4,7	1,2,3,5	0.008783	73.1003	0.0518571

Table 5.5: Table showing the first 15 entries in the SEP for LC-D.

as the excluded subtasks can be observed. Finally the generated average accuracy impact to the applications output results are displayed to show the potential impact to the output result when selecting the plan entry at runtime.

In chapter 3 a description of graceful degradation was first described where the accuracy and processing duration of the application reduces gracefully from one selection of the subtask entry plan to the next. Table 5.5 illustrates the observation of this assertion as the entries in the plan show a graceful reduction in processing duration as implemented in LC-D, LC-C and LC-B. From the graph, degrading the latency requirements to process each message has a knock on effect on the output accuracy as assumed in chapter 3. However in overload situations this is the expected behaviour of CS implementations, with the assumption that discarding subtasks at runtime reduces the cost of processing each message and thus more input messages can be processed within the local container. However, the question as to the volume of messages which are processed in this situation and the accuracy of the result to the ideal are yet to be answered, they are covered in section 5.3

A similar observation can be made from table 5.6 when observing the graceful degradation of application accuracy when moving from one entry in the table to the next, as implemented in LC-A. In this case, this is at the expense of application message processing duration, where to

Subtask Plan Id	Included Subtasks	Excluded Subtasks	Accuracy (%)	Average Processing Duration Per Message (ms)
512	1,2,3,4,5,6,7,8,9,10,11,12,13	NA	100	0.0599002
163	1,2,3,4,5,6,7,8,10,11,12,13	9	99.9993	0.0522192
255	2,3,4,5,6,7,8,9,10,11,12,13	1	99.9808	0.0517011
419	2,3,4,5,6,7,8,10,11,12,13	1,9	99.9808	0.04402
167	1,2,3,4,5,6,7,9,10,11,12,13	8	99.9694	0.0523403
171	1,2,3,4,5,6,7,10,11,12,13	8,9	99.9694	0.0446592
34	1,2,3,5,6,7,8,10,11,12,13	4,9	99.9612	0.0517471
254	1,2,3,5,6,7,8,9,10,11,12,13	4	99.9612	0.0594282
423	2,3,4,5,6,7,9,10,11,12,13	1,8	99.9502	0.0441412
427	2,3,4,5,6,7,10,11,12,13	1,8,9	99.9502	0.0364601
290	2,3,5,6,7,8,10,11,12,13	1,4,9	99.942	0.043548
385	2,3,5,6,7,8,9,10,11,12,13	1,4	99.942	0.051229
38	1,2,3,5,6,7,9,10,11,12,13	4,8	99.9306	0.0518682
42	1,2,3,5,6,7,10,11,12,13	4,8,9	99.9306	0.0441872
294	2,3,5,6,7,9,10,11,12,13	1,4,8	99.9114	0.0436691

Table 5.6: Table showing the first 15 entries in the SEP for LC-A.

have a focus on the output accuracy of the result, subtasks are removed from the application in a different order than mentioned in table 5.5.

As a result of this difference in ordering when comparing message processing duration costs in both tables against one another, the cost of an accuracy focused load handler is higher processing duration then load handlers which have a focus on processing duration. As such our expectation is that processing duration load classifiers will have a higher message processing count and can withstand the effects of bursty data streams to a greater effect however this may be at the expense of application accuracy.

Finally, the relationship between application accuracy and processing duration within the aggregate SEP can be seen in figure 5.1 for the application. A Loess Curve or fitted regression line was plotted to show the relationship between processing duration and average accuracy output for each subtask subset within the subtask execution plan. From it an almost linear relationship can be observed as lower and lower processing duration values equate to lower and lower estimated levels of accuracy within the output result of the application. As such one can conclude in this experiment that there is a direct relationship between application accuracy and message processing duration, such that higher levels of accuracy require greater amounts of subtasks and greater amounts of time.

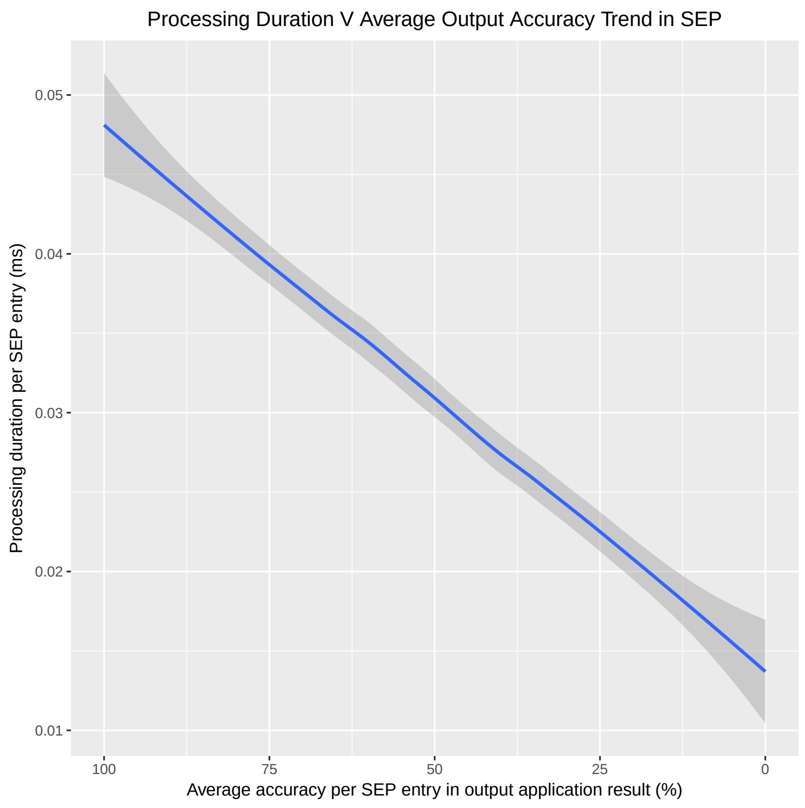

Fig. 5.1: Showing the relationship between output accuracy vs processing duration of messages in the aggregate SEP. From this figure a positive correlation relationship can be observed between the message processing duration time and the output accuracy of the applications expected result. Such that as message processing duration decreases so to does the accuracy of the produced result. The gray region represents 95% confidence interfaces for the Loess Curve or fitted regression line.

5.2.8 Evaluation Setup Summary

This section outlines the evaluation setup used to evaluation CS within the proof of concept application presented in this evaluation. First the objective of the evaluation are discussed. Next an outline of the deployment of the application and the data collection process was provided.

Further, the load handlers and their parameters are discussed as well as all experimental parameters used in the application. Analysis was also provided on the application profiling activities which generate the SEP for the application used for subtask selection. The metrics of the application measured are discussed as well showing how the proof of concept application was analysed.

The results of our Twitter search application implemented on top of SSCF with CS and both LS are provided in the next section.

5.3 Experiment Results

This section outlines the results of the experimental Twitter search application implemented to validate and asses CS approaches as well as to compare CS against load shedding alternatives. The metrics used for this analysis can be found with a discussion of the rational in section 5.2.6. In the following sections all results are described in aggregate form, such that all three container results have been aggrgrated into single view for discussion and graphing purposes, unless otherwise stated. For example, an SEP would represent the aggregate SEP generated within each local container based on the subtasks allocated to it. In the next section, load handler processing duration measurements will be described.

5.3.1 Load Handler Processing Duration

First we look at load handler processing duration as illustrated in figure 5.2. Load Handler duration is an important factor to asses given that load handlers take time to asses the current containers utilisation, and thus determine if messages residing in their queue can meet the applications output latency requirements. Given load handlers preform some operation with the input data stream and thus take time, time is taken away from processing the input streams into PEs, hence lower load handler processing times are desired.

From the graph which represents the aggregate time spent in all containers in the application, we can see that both approaches of subtask selection in LC-A and LC-D preform worst. By taking the most amount of time to make an appropriate selection in the SEP as input data rates grow and application utilisation rates increase. Given the aggregate SEP has a sizeable entry count of 512 entries, traversing the table takes time, which can be quantified with a worst case complexity of $O(n)$ for naïve linear search.

In the case of LC-D specifically, it's SEP is organised by lowest processing duration first with increasing duration for each entry there after, as such at low input data rates in the application entries in the table at the lower end can be selected quickly. At high data rates, the entire table needs to be traversed for each incoming message to find an appropriate SEP entry or eventually defaulting to just the mandatory subtasks. This gives the worst case complexity of $O(n)$. Additionally for each new message received the total cost of processing messages in the current container were assessed for each entry in the SEP, increasing the burden upon the load classifier. This is in contract LS-C and LC-B which preforms a similar assessment of container

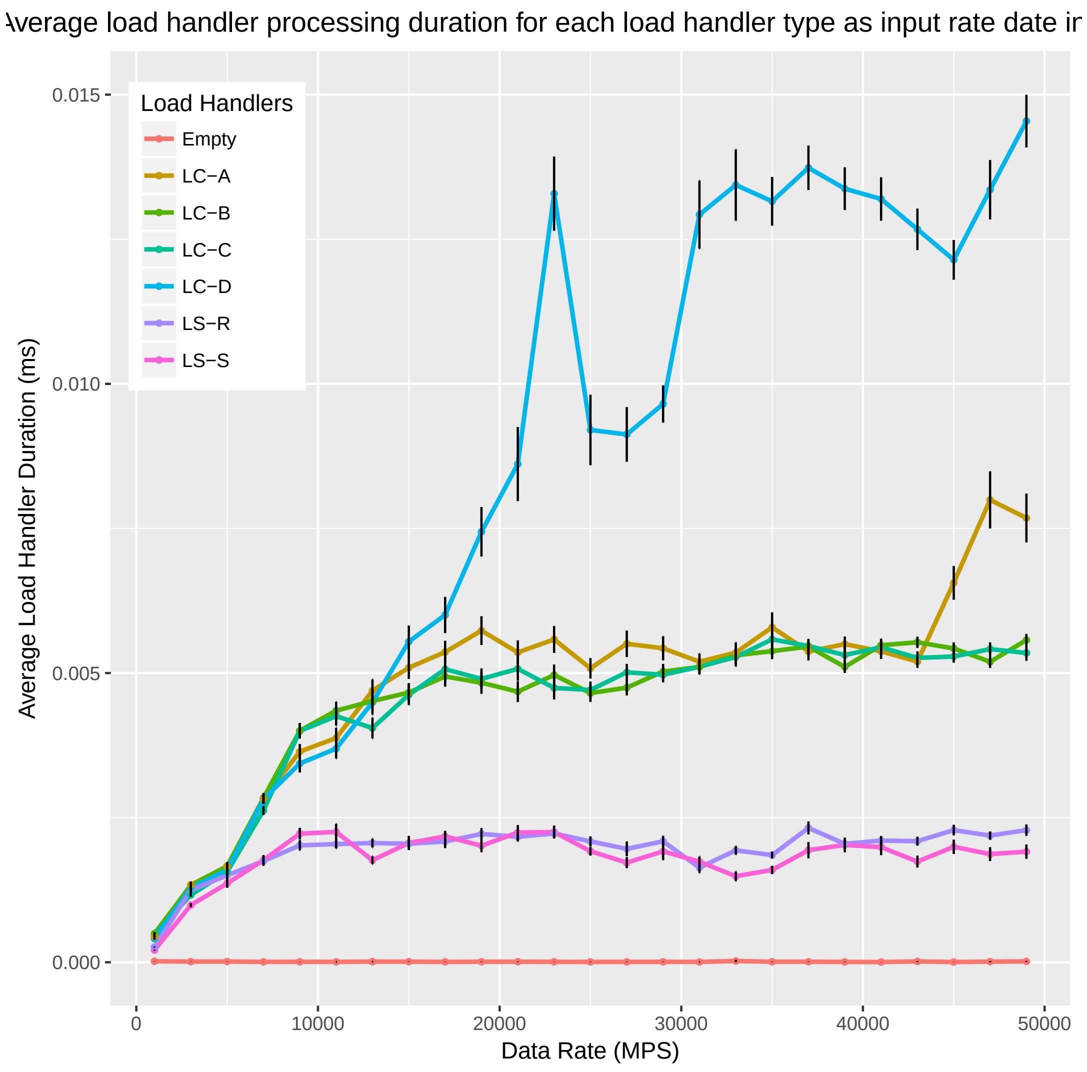

Fig. 5.2: Showing the average aggregate load hander processing duration in all containers as load increases in the application due to increasing input data rates. The shaded region around each trend line represents 95% confidence interval. Data points have been removed for clarity.

load but does this once per message received rather then for each entry in the SEP.

Interestingly, LC-A seems to out preform LC-D in terms of reduced load handing processing times beyond the 10,000 MPS mark where all load handlers except LC-C and LC-B appear to diverge. This was not expected as LC-D SEP has been optimised for end to end processing duration. This can be rationalised by the distribution of entries within the SEP for LC-A, specifically the distribution of average processing duration. In table 5.6 a sample of the SEP for LC-A is shown. Observing average processing duration, a non-linear relationship between one entry in the table to the next appears, this is in direct contrast to LC-D table which shows an ever decreasing processing time. Such that if a message or set of messages appear to miss their processing duration deadline, LC-D may have to traverse the entire local SEP table to find an entry to meet that deadline. Within the next container downstream in the pipeline the SEP must be traversed in the same regard causing large load handler processing times. However, in the case of LC-A there was a possibility that a valid SEP entry would better fit the current latency expectations, given the SEP entries arrangements are non-linear in processing duration.

LS-R and LS-S in this case preform best of all, showing a general trend of duration time increasing only to 0.0025ms during the entire experiment. This can be attributed to how *cheaply* they can select a value to determine if the incoming message should be discarded. LC-S would have been susceptible to the worst case complexity of traversing a list, if not for the pointer which maintains the selection of the current bin in the histogram. Thus this point becomes the next starting point for the selection of which values to shed. LS-R works in constant time to determine how much data to drop and determine to shed the message or not. As such it preforms well here.

Further LC-B and LC-C perform well compared to LC-A and LC-B, showing their improved designs to reduce subtask selection costs have a real world improvement. LC-C improves upon the design in LC-A and LC-D by reducing the cost to measure the total processing cost of all messages in the local container against the containers allocated total processing time.

Finally as expected in the experiments baseline the empty load handler implementation which merely takes messages from one queue and places them into another for processing in the PE, spends very little time preforming this task. In the next section, queue capacity measurements will be described.

5.3.2 Queue Capacity

Figure 5.3 outlines the average queue capacity of all queues and sliding windows in the application as an aggregate capacity value as system load increases for each load handler type. The graph is illustrated as a trend line each load handler type showing the general trend between higher and higher data rates and the capacity of queues and sliding window in the application.

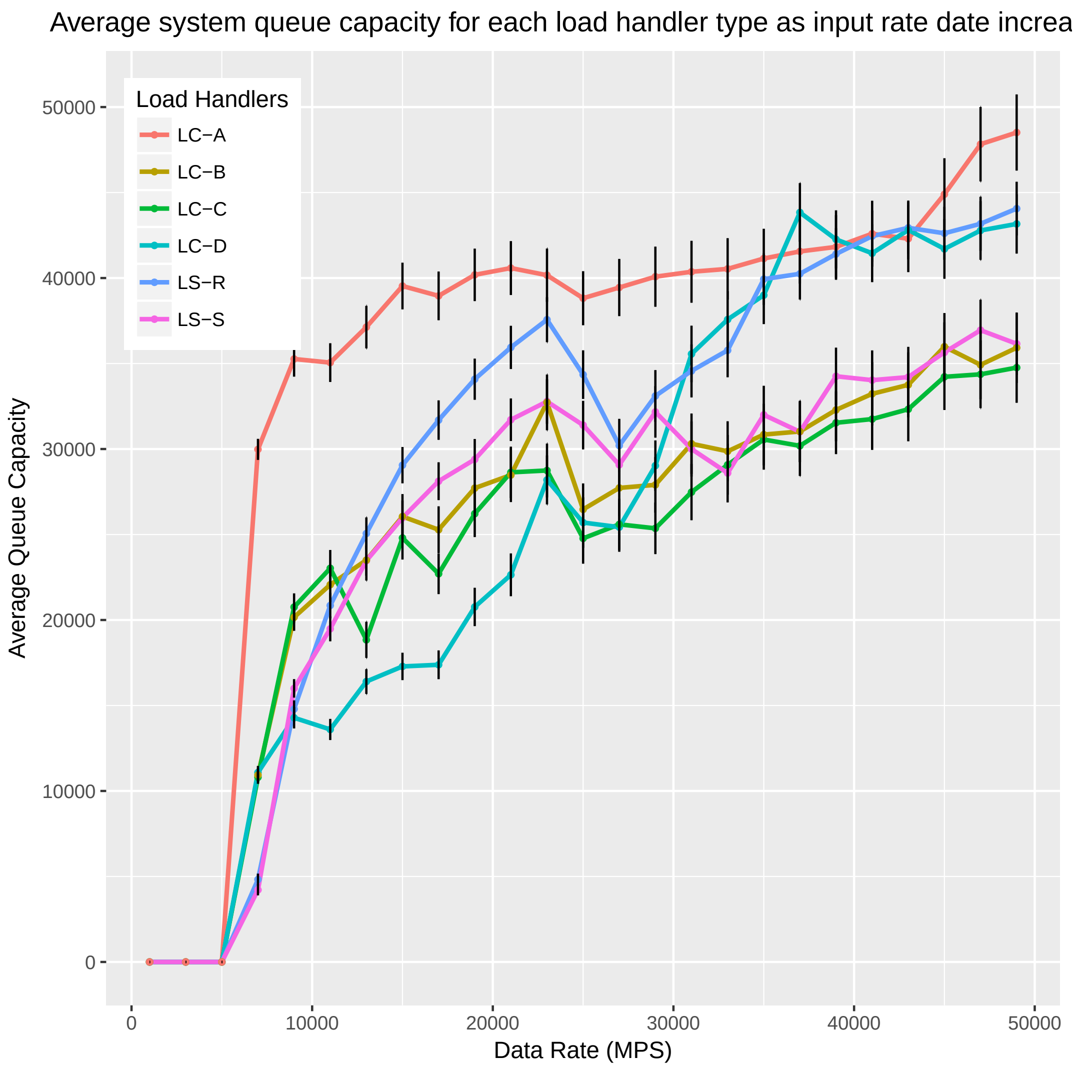

Fig. 5.3: Showing the volume of data within all queues in the application as system load increases by load handler type. The shaded region around each trend line represents 95% confidence interval. Data points have been removed for clarity.

As data rates increase within the application due to ever increasing levels of input data from a simulated bursty data stream, capacity of all the queues in the system increases as a consequence. As such ingestion times or message processing times increase given messages remain within the containers for longer periods of time. Adaptation in the application must be preformed to ensure messages meet their end-to-end processing duration deadlines, a requirement of the application here of two seconds as mentioned in section 5.2.7. At approximately 3,500 MPS the application becomes overload such that the volume of messages in the queues exceed the total end-to-end required processing duration of two seconds.

In the case of the load shedders, LS-R preforms worst of all, having the largest amount of messages residing in the queues. Although shown in figure 5.2, LS-R and LS-S have a low processing duration relative to other load handlers. As LS-R determines the cost of shedding on a message per message basis, it would appear this granularity is too small and the shedder cannot discard data at a fast enough rate. As such, more and more messages entered the queues and the LS-R determines whether to keep the next message or discard it randomly. LS-S appears to benefit from the bin selection strategy it provides. In this groups of messages can be discarded and so it has a more favourable message queueing profile then LS-R as any of the load classifiers, except for LC-A and LC-C.

Interestingly, LC-D, LS-S and LC-B appear to have a very similar relationship between data rate increase and queue capacity as the application experiences. LC-C appear to perform best overall showing the slower rate of queues filling as data rates increase. Initially it appears that LC-D preforms best but it's trend appears to be greater than other load classifiers graphed and as such would indicate that queues fill with this approach faster than other load classifiers. LC-C appears to preform best of all load classifiers. It appears that LC-C has a slower startup time hence the offset from zero where it starts to process messages. LC-B appears to preform better than LS-R but only as good as LS-S which is surprising. More surprising it would appear that LC-B is slowing at processing messages than LC-C which would suggest that the binary search of the SEP here has slowed down queue consumption rates and an iterative approach is faster a processing messages in the load classifier. Figure 5.2 suggests the rate at which LC-C and LC-B process messages are quite similar. Investigating further, the differences can be accounted for in what subtask have been selected within each strategy and as such one strategy selects an SEP entry which processes messages faster than the other on average. Table 5.7 outlines on example of the differences with a sample SEP selection from the experiment but we leave this discussion for the next section.

LS-A does not appear to preform well. As mentioned previously, this can be attributed to it's SEP selection strategy which is optimised for accuracy rather than latency and as such it can take longer to iterate the table to find the appropriate entries to fit the containers current view of the application load.

Irrespectively, LS-S appears to preform as well as LC-B overall and as well as LC-D at higher data rates as a result of its bin based selection shedding capabilities, a capability missing from LC-R.

An important aspect as to why the load classifiers perform well in this regard was based on the selected subtasks and the reduced processing cost of messages. As fewer and fewer subtasks were selected at runtime as input data rates increased during the simulated burst. As mentioned previously this is a consequence of adapting to the volume of data within the queues and the average processing time of each message, with the aim of trying to ensure all messages are processed within application processing deadlines. To further justify this assertion, the next section, section 5.3.3 provides sample sets of subtasks as selected from the SEPs by each load classifier during application runtime.

5.3.3 Subtask Selections

As mentioned in the previous section, LC-C, LC-D and LC-B preformed best, and in some cases as well as LC-S as queues in the application fill with incoming messages. To understand why this is the case, this section provides an overall analysis of the SEP. Additionally this section outlines with a sample, the entries in the plan which were chosen at runtime by the four load classifiers. A choice that was made based on the containers local view of application load.

As per design, CS was preformed to reduce the cost of processing each input message such that during application overload, message throughput could be increased. During the experimental runs each load classifier attempted to continue to produce a timely result with a subset subtask selection chosen from each load classifiers respective SEP. We illustrate this through two tables for clarity purposes.

In table 5.7 and table 5.8, two samples of the applications sliding windows output are provided at different data rates. Within each table the same data rate, data set and sliding window are used to construct the table. This provides an equal footing to compare and contract the load handlers against one another in terms of subtask selection and the effect this has on the application.

First, table 5.7 provides a view of a sample set of the output of a specific sliding window,

Load Handler	Queue Capacity	Messages Processed	Sliding Window Size	Overload Amount	Load Handler Duration (ms)	Subtask Count	Subtasks Selected	Subtasks Excluded
LC-A	51440	20561	38	0.211711	0.00710048	12	1,2,3,4,5,6,7,8,10,11,12,13	9
LC-B	12057	38944	47	0.0025882	0.0062633	13	All 13	NA
LC-C	38789	12212	34	0.461764	0.00693035	10	1,2,3,5,7,8,9,10,11,12,13	4,6
LC-L	29810	21191	38	0.192842	0.00850077	13	All 13	NA
LS-S	32622	14684	36	0.277062	0.00377603	13	All 13	NA
LS-R	31790	15619	35	0.252143	0.0019038	13	All 13	NA

Table 5.7: Showing a sample output of a sliding window pertinent to the current application load and subtask selection choice. This data was taken from data rate 17,000 MPS and dataset 15 for each load handler within the output application results.

sliding window number 3 for all load handlers within the 17,000 MPS and dataset 15 experimental run. Shown within it is the average processing duration of the each load handler, how much data has been processed in the application thus far and how many messages reside in the queues at this point in time. Also shown are the subtasks which were selected in the applications as the messages flew through each container. When this sliding window was sampled, the application was overload given the data rate which was being sent to the application by the data loader as outlined in chapter 4. The load handlers confirm this with their own approximation of application overload, given the value of overload being non-zero. Above zero possible adaptation occur in the application should the expected processing time of the quantity messages in the load queues of the container exceed it's locally allocated end to end processing duration. At this overload amount the adaptation in subtask selection can be seen for some load handlers, namely LC-A and LC-C but not others.

Load Handler	Queue Capacity	Messages Processed	Sliding Window Size	Overload Amount	Load Handler Duration (ms)	Subtask Count	Subtasks Selected	Subtasks Excluded
LC-A	59591	42410	43	0.886024	0.00534063	8	2,3,6,8,10,11,12,13	1,4,5,7,9
LC-B	55774	45227	48	0.79031	0.0046847	9	2,3,4,5,6,9,11,12,13	1,7,8,10
LC-C	52225	46776	43	1.16367	0.00453696	9	1,2,3,4,5,6,10,12,13	7,8,9,11
LC-L	61403	32616	40	1.33812	0.00969357	8	2,3,6,7,8,11,12,13	1,4,5,9,10
LS-S	50874	34101	44	0.823741	0.00182393	13	All 13	NA
LS-R	53594	40748	51	0.905175	0.00117329	13	All 13	NA

Table 5.8: Showing a sample output of a sliding window metadata pertaining to the current application load and subtask selection choice. This was taken from data rate 33,000 MPS and dataset 15 for each load handler within the output application results.

LC-A and LC-C have modified the subtask selection, which can be inferred from both the volume of messages in the queues as well as the average processing rate of the load classifiers. This ensures end-to-end message processing durations are meet. SEP entries are navigated to select an appropriate subtask sub selection in their respective SEPs. LS-S and LS-R also modify the application by shedding input messages but here the focus is on subtask selection for which they do not modify. The purpose of the two load shedders in this table is to shown they too experience application overload as input data rates increase.

Interestingly, at this stage the quantities of valid messages in the sliding window can be seen to be different for each load handler type. Both load shedders here appear to be slower at processing data as well as having a lower sliding window count than all other load handlers. LC-A appears to be quite slow too in this regard given the amount of data remaining in the queue, which again is a result of the navigation time of the SEP table, given it's optimised for application accuracy over message processing duration.

LC-B appears the fastest at processing messages from this sample sliding window as both the amount of valid entries in the sliding window and the amount of messages which have been processed are higher than others. This can be attributed to the load classifiers selection speed, selecting appropriate entries in the SEP for the perceived given load. This algorithm exhibits a worst case complexity cost of $O(logn)$ in navigating the local SEPs with a binary search algorithm rather than an iterative form as seen in all other load classifiers with a worst case complexity of $O(n)$.

When comparing the contents of table 5.7 against the contents of table 5.8 an interesting picture of how load handlers preform under higher and higher input data rates comes to light. There is an obvious trend that subtask selection in all load classifiers has increased as more message enter the queues but are not processed fast enough, thus inducing overload in the application. To adapt to this, each load classifier navigates it's own SEP and decides which subtasks to retain and which to discard to ensure processing durations times are adhered to for each message in the application.

The overload amount of the application, as in the volume of messages in the queues and messages processed in the application have all increased considerably as the data rate has increased, as one would expect at higher input data rates. Further, the amount of valid messages in the sliding window which have met the applications input search query requirements have also increased with the higher data rate.

In the next section, sliding window size will be discussed.

5.3.4 Sliding Window Size

In figure 5.4 the average sliding window size is shown or said another way, the average amount of valid search results are shown. This is illustrated as a trend line for each load handler type as input data rates increase. This helps visualise the general trends of the size of the sliding window as data rates increase and the effect each load handler has on the results of the application.

Sliding window size represents the amount of valid search results which have been accepted by the application after the subtasks have filtered out results and ranked them using the selected subtasks. Tweets which are not of value to the input search query to the application are rejected prior to entering the sliding window. Another important point to mention as the sliding window was a stateful sliding window, such that the item count increased as the application processed more and more messages in the input data stream, as more valid results were retained in the sliding window. As mentioned in section 5.2.7 sliding windows are processed every second, illustrated here are the aggregate results of all sliding windows within the application for each data rate and load handler type.

The empty load handler sends every Tweet in the data set to the first processing element where it may be rejected or accepted based on the search filter. In the case of load classifiers the processing duration of each message was reduced when subtasks were removed from the application as load increases. Thus in the load classifier case, more data can be processed within the same time frame it takes to produce a sliding window than in all other load handler types.

From figure 5.4 both load shedders perform worst of all, having the lowest amount of messages in the window as data rates increase, with LS-R preforming worst of all, a result which was not unexpected given it's random probably of discarding messages. LS-S preforms better here as expected given it considers incoming messages utility to the application.

Interesting, LC-A exhibits poor performance when compared to other load classifiers but still retains more valid Tweets in the sliding window than both load shedders. It's poorer performance can be rationalised as we have seen in previous sections where LC-A favours accuracy over application latency and at times it's SEP organisation can be advantageous but here it exhibits slower processing. When considering LC-A, it's SEP was arranged to optimise application accuracy at the expense of processing duration, as described in section 5.2.7.1. As such message processing

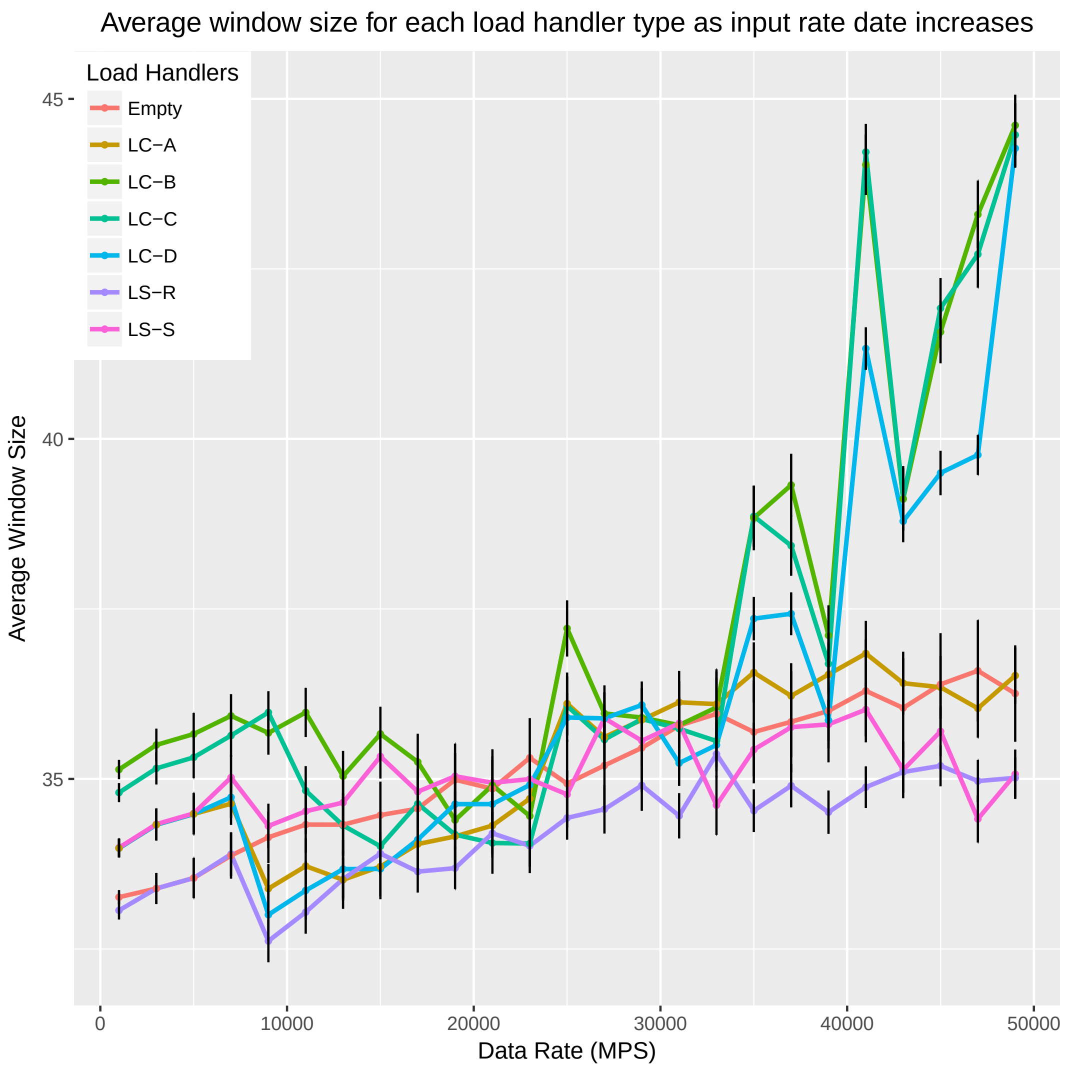

Fig. 5.4: Showing the volume of messages within each sliding window output as system load increases. This is illustrated by load hander types. The shaded region around each trend line represents 95% confidence interval. Data points have been removed for image clarity.

duration times in the application were sacrificed at times for application accuracy and as such lower numbers of valid Tweets were accepted into the sliding window before output results were produced.

Further, LC-D, LC-B and LC-C preform with a very similar trend with each having very similar sliding window item counts. LC-B preforms best of all here which can be attributed to the faster selection process of entries in the SEP as well as an optimised approach in determining current system load. This is the same approach as used in LC-C. LC-B has a worst case time complexity factor of $O(logn)$ which improves upon the selection process of all other load classifiers. Other load classifiers have a worst case time complexity of $O(n)$ and as such LC-B can found a valid SEP entry faster than other load classifiers on average. Note both types have a best case complexity of $O(1)$ hitting the first entry in the table of in the case of binary search hitting the entry at the half way mark in the table, given how binary search operates. This results in more time spent processing input messages than time spent search the respective SEP, causing less of a delay in the application.

Another interesting point to consider is the processing cost of load handlers as outlined in 5.3.1. LC-C and LC-B have lower processing costs than other load classifiers and as a result the container had more time to process messages given the lower load handler processing time. As a result, more messages get through to the local container in the same time frame as with other load classifiers, such as LC-A and LC-D. These messages are added to the sliding window given the large sliding window count in this case.

Finally, the empty load handler preformed no load handling work and as such all Tweets in the data stream reach the PEs and are processed by all subtasks. This has the consequence of slowing down the data stream as input data rates increase, causing a backlog in the queues, given the constant processing rate of he application. The results of this are shown in section 5.3.5, in the next section, where application duration times will be discussed.

5.3.5 Application Duration

In previous sections, a number of interesting measurements have been discussed, including sliding window size and subtask selection, but two overriding factors in stream computing lead to the best understanding as to which load handing strategy is best during bursty data events. How long it takes to produce a result as well as how accurate that result will be. If a application is late, it has little value or no value, if the result is largely inaccurate, how can it be trusted and acted upon. This sections pertains to the duration element of this assertion, the next section, section 5.3.6 details with the accuracy element of this claim.

In terms of processing time or processing duration per message, in section 5.2.7 and section 5.3.3 samples of the SEPs for each load handler type, which show the reduced cost of processing messages when subtasks are removed from the application. However what is missing from this picture is a complete understanding of how various subtask selections effect the overall processing duration of the application compared against the baseline.

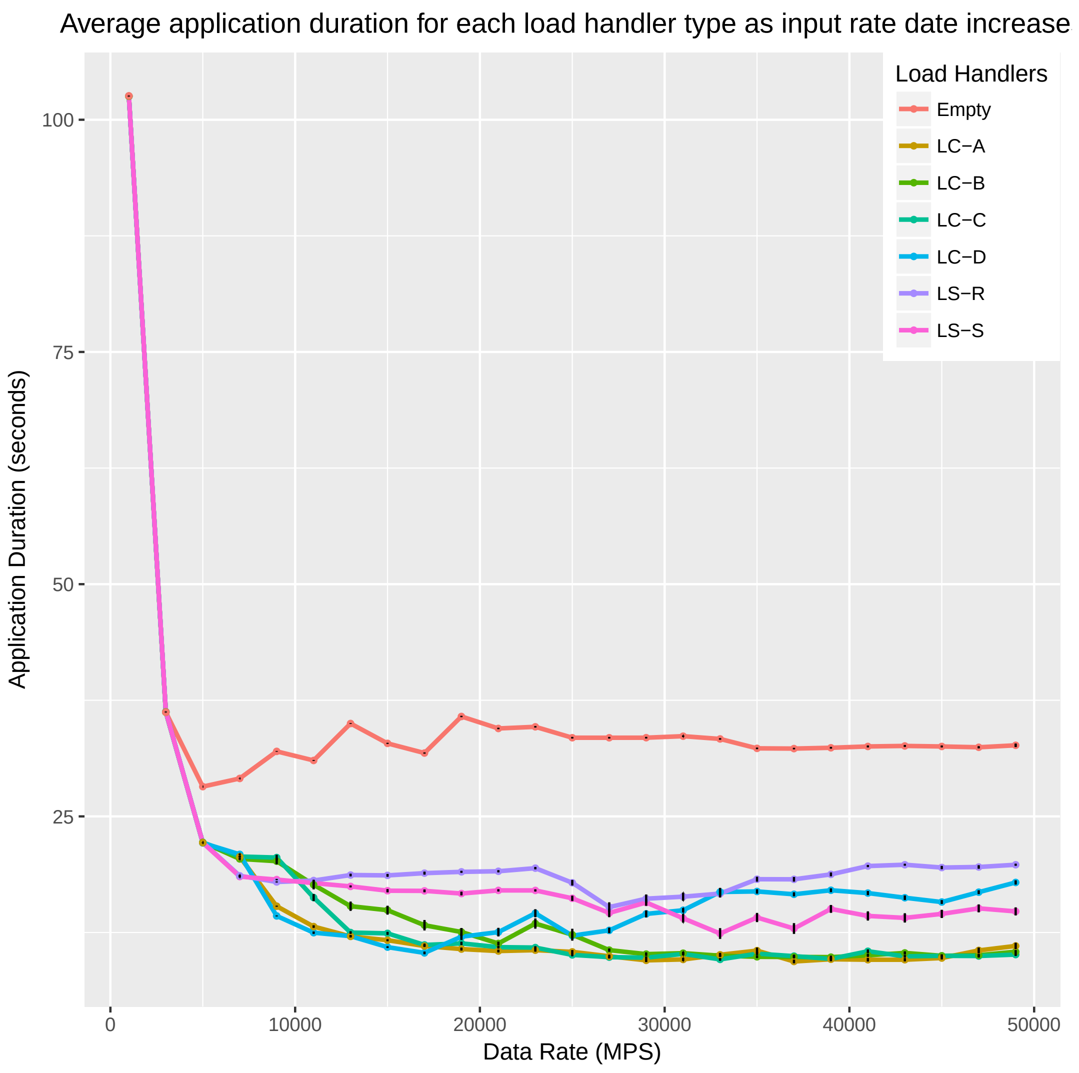

Fig. 5.5: Showing the average duration of the application to process 100,000 messages in each of the 30 data sets, as input data rates increase to simulate a bursty data stream. Error bars describe 95% confidence intervals for each data point in the figure

Figure 5.5 shows the duration it took on average to process all 100,000 messages in each data set when processing messages with various load handlers available to the application, over a variety of increasing data rates. From this figure we can see that the baseline case has the worst effect on application duration given that no adaptation takes place and the application must process all messages in the stream using the full set of subtasks available to the application.

Further, LS-R and LS-R perform worse here on average than the CS implementations which tend to preform best. Interestingly LC-D performs worse than LS-S at 30,000 MPS. Investigation into this appears to show what has been seen previously with LC-D. At higher input data rates to the load classifier, traverse the entire SEP before it will then select the just the mandatory subtasks for execution in the application.

As expected, LC-C and LC-B perform well here given their SEP has been focused on message processing duration rather than accuracy as is the case with LC-A. In addition to their reduced cost in finding an appropriate entry in the SEP in both LC-C and LC-B because of their respective improved designs.

Surprisingly LC-A preforms well here given it's focus on accuracy rather than message processing duration. In section 5.3.1 a similar surprising outcome was also noted. In the SEP for LC-A, the subtask combinations are first ordered by subtask accuracy and not message processing duration. As such the distribution of SEP entries in the table in terms of message processing duration is non-linear, in that from one entry to the next the proceeding duration may not reduce but could increase. Examples of this can be seen in table 5.6 and as such it may select entries in it's table which can meet application deadlines more quickly. Consequently, although LC-A and LC-D have a worst case complexity of $O(logn)$ to iterate their SEP, LC-D in general will iterate through more entries to find an appropriate entry than LC-A, thus taking longer.

In the next section, application accuracy measurements and results will be described.

5.3.6 Application Accuracy

Finally, this section discusses the accuracy of the application results for each load handler type in the application. To measure this value, each sliding window created within each experimental run was compared against the corresponding sliding window in the baseline experimental case for each load handler type. The methodology used to generate this value was also addressed in section 5.2.6.1. With this methodology, the accuracy of all load classifiers can be compared to both load shedder implementation using Spearman's Footrule. As a reminder, accuracy was measured as displacement or movement of Tweets in one ranked sliding window set in an experimental run measured against the positions of the same Tweets in the corresponding baseline sliding window. A displacement value tending towards zero equates to low levels of displacement, a favourable outcome.

Figure 5.6 shows the displacement value for each load handler type for each experimental run as input data rates of the data stream increased within the application. Here, the linear regression lines for the relationship between application accuracy and input message data rates are shown for each load handler type in the experiment. This figures shows both the individual data points as well as the general trend of each load handlers effect on accuracy in the application. Interestingly the overall displacement trend is bound between 30 and 45 approximately. Showing that all load handlers tend to produce a result similar to one another. Also considered here is the evaluation methodology as outlined in section 5.2.6.1 where in the case of load shedders, an approximation is used to show values which may not be part of the table to satisfy Spearman's Footrule set count requirement.

Interestingly and perhaps as expected LC-A appears to perform best of all load classifiers from a closer view of it's general displacement trend as observed in figure 5.6. LC-A appears to preform best which can be expected given it's SEP has been organised to focus on accuracy. As such LC-A discarded subtasks which have little or low effect on application accuracy first, to reduce overall message processing times as input data rates increase.

Further, LS-S has performed well as can be seen from figure 5.6. The semantic or value utility nature of it's selection strategy appears to have positive results in discarding large amounts of message to stave off the effects of the bursty data stream. This further helps retain messages of value to the application which contain the input search query. LS-R has also preformed well here given it was expected that due to it's random nature of discarding data it may not in fact produce accuracy results.

LC-D appears to have the poorest result in terms of output result accuracy as the poorest performing CS implementation in terms of application accuracy, with LC-B coming in a close second. The expectation here was that a Binary Search Algorithm with a better worst case time complexity than pure linear search would yield better results due to spending less time searching for an appropriate entry in the SEP. This expectation was not met. This was primarily due to

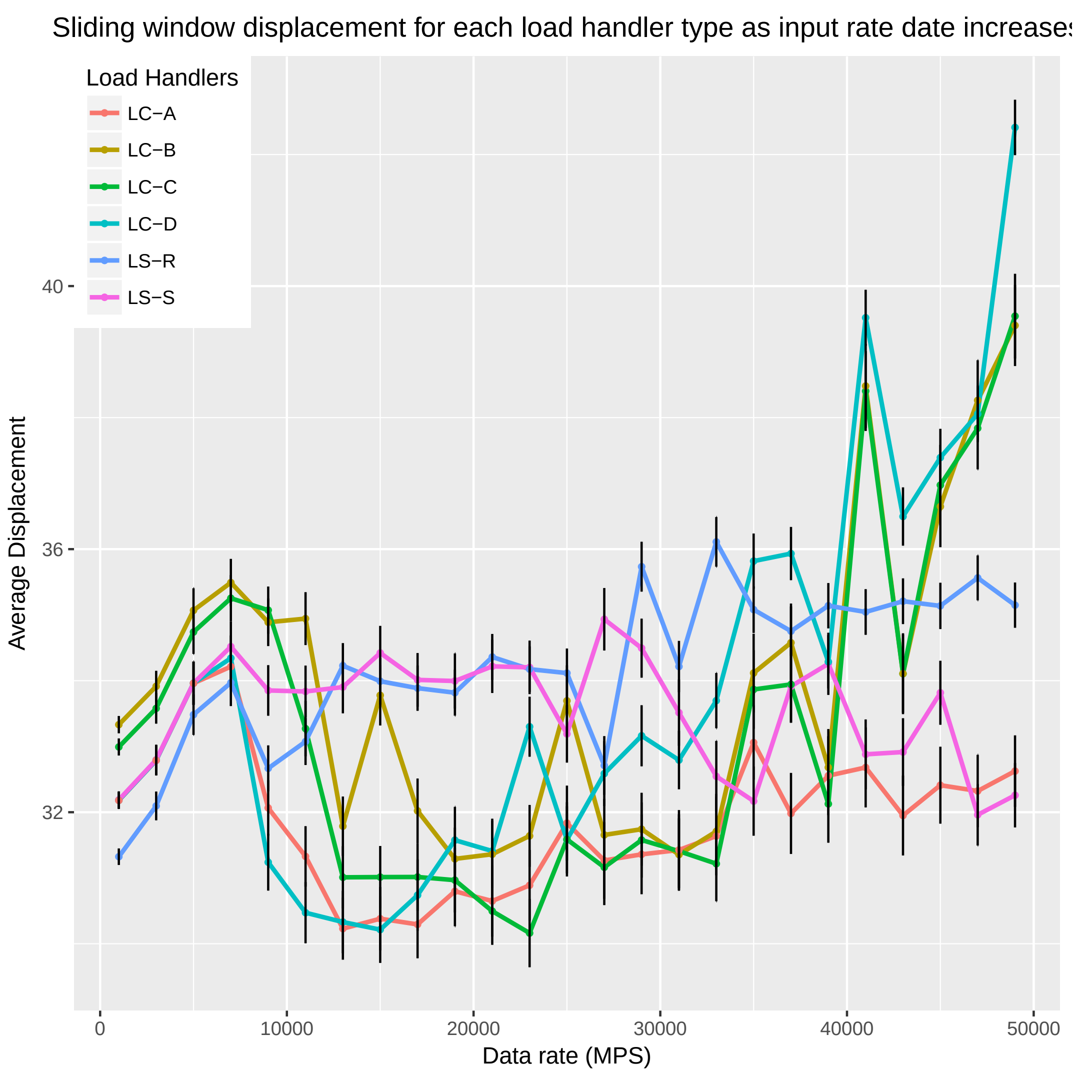

Fig. 5.6: Showing the Spearman's Footrule or total aggregate displacement per each sliding window as input data rates increase. These trend lines represent how much of the ideal results data is contained within the sliding window output in an experimental run with each selected load handler as well as the difference in rank order of each output Tweet. Lower value results are better. The shaded region around each trend line represents 95% confidence interval.

their focus on message processing times and organising the SEP as such. At higher data rates LC-D has to traverse the entire SEP table in each container to eventually select just the mandatory tasks alone being the last entry. This uses up time in the LC and thus allows messages to queue up further, resulting in poor SEP selections which effect result accuracy. This has a knock on effect with the applications result accuracy, such that when compared against the baseline sliding window Tweets the displacement value for each sliding window yielded the worst results.

5.3.7 Results Summary

The results in section 5.3 attempt to illustrate a picture of how load classifiers and load shedders perform as the Twitter search application receives more and more data as a result of a simulated bursty data stream. CS has an effect on the output results accuracy in the application, as message processing times were reduced when subtasks were temporarily removed from the application. This caused larger and larger inaccuracies in the output results as more and more subtasks are removed from the application corresponding with increased message ingestion rates. On a positive note, CS also has the effect of increased message throughput rates, such that as more messages are processed in a period of time, even to just an approximate level, as intended, application duration times decrease.

This is not surprising and is per design. However, the focus here of CS was it's comparison against it's various implementations and primarily against load shedding and as such some interesting results emerged.

First, generally sliding window output size produced by load classifiers were greater than that of results produced by load shedding. This shows that more results are produced from the application as a result of reducing the cost of processing each message by shedding subtasks. Although load shedding discards input messages, in the random case discarding messages that are both valid and invalid to the application without discrimination, and in the semantic case by utility selection. Both have an effect on the applications results. The increased utility selection in LS-S, coupled with a grouping strategy for message rejection produces sliding windows with more results but not to the same degree as with CS implementations. CS implementations processed messages the fastest but generally each result is produced with at a reduced accuracy level (incorrect position in the ranked output set). Interesting LC-A which attempts to focus on both application accuracy and message processing duration, tends to out preform all other load classifiers and load shedders in this regard.

Secondly, adding load shedders and load classifiers into the application adds a processing duration cost to the applications overall processing duration per message which is not present otherwise. Comparing load shedders against load classifiers here shows that load classifiers generally take longer to produce a result than load shedders. This is primarily due to their time complexity cost to navigate their respective SEPs, searching for an entry in the plan which was appropriate for the current system load. However, this increased cost appears to be balanced against the reduce messages processing duration in the PE due to the sub selection of subtasks. Conversely, load shedders do not have this benefit. The subtask selection in the LC experimental runs were constant with the same 13 subtasks selected for each message received. Load shedders do introduce an additional processing cost but not to the same degree as load classifiers.

Thirdly, the effect load handlers have on the data stream by slowing it down must also be considered from a message queuing and subtask range perspective. Queues were used to buffer against data stream bursts and thus smooth out the data stream input rate. In this case, if the queues fill beyond capacity, data is unintentionally lost unlike load shedding where data is lost controllably to some degree. From figure 5.5, CS generally preforms best when compared to LS in terms of message processing times and application duration. However CS has a finite range in which it has a controlled effect on the data within the stream. Said another way, CS cannot adapt the application beyond a range greater than removing all optional subtasks where it selects all mandatory subtasks for the application to process data. Should queueing times remain the same or worse, increase as more messages are received, the application will eventually feel the effects of queue overflow. As mentioned in section 1.5 we scope our bursty characteristics to short-lived but intense bursts given CS may not be applicable in longer term bursts.

When considering overall application duration as is outlined in section 5.3.5 optimised forms of CS implementations, namely LC-B, LC-C and LC-A, perform best against load shedding alternatives. A major requirement of stream computing system is to produce timely results which in this case, CS has achieved with the LC-A implementation of CS-A showing the most promising results when both application processing duration and application accuracy are considered. This suggests that a further optimised SEP and selection strategy focusing on application accuracy may lead to more timely and accuracy results during overload. We leave this suggestion for future work.

5.3.8 CS Drawbacks

As outlined in section 5.3.7 and as mentioned in chapter 3, CS has a number of drawbacks which should be discussed both in terms of it's applicability to a given application as well as it's range of operation as is pertinent to bursty data stream.

The underlying premise of stream computing is to produce timely results as late results are deemed of little value. As such, accuracy of the application results tend to reduce during bursty data events, as accuracy is traded for reduced message processing durations. Section 5.3.6 and section 5.3.5 show the results of the experimentation from both an accuracy and duration point of view. In cases where CS implementations optimise for message processing duration (LC-D), it is clear this has the worst effect on the results of the application results and how accurate those results might be.

Further, when all optional subtasks have been removed from execution in the application as a result of large overloads, only the mandatory subtasks remain for execution. In such a case where the bursty data stream rate is greater than CSs ability to offset it, the application will feel the effects of this by not being able to process messages any faster as can be seen in figure 5.5. There data rates above 20000 MPS are almost 3.5 times the applications 100% CPU utilisation. However, in cases where there is sufficient selectable options in the SEP, CS can offset higher input data rates by appropriate optional subtask selections.

The applicability of the CS strategy to any application should also be mentioned. Unlike LS which tends to take a data centric view to manage busty data stream (a selection of Random or Semantic is still application specific), CS takes an application centric view. As such CS requires an application to be constructed in such a way where the application can be broken down into a set of individual tasks or subtasks, of which some must be optional in generating a result for the application. Without this requirement, CS is simply not applicable. Applications types such as web based search or application search as outlined in this evaluation can contain such structures. Where the search index is based on multiple metrics and CS and remove tasks which create metrics in this index. Other examples with this structure include rule based systems, such as authorisation rule or classification rule systems as outlined in section 1.2.1 tend to have sufficient complexity and structure for CS approaches to be applicable.

Finally, CS requires data and application profiling prior to the start of an application. Profiling allows tuning of the subtask's accuracy contribution and message processing duration which are then used to generate the SEP. Should a LC's understanding of the cost of processing the data change, the expected processing costs of each subtask may not be applicable any further. Similarly, LS-S's filter must be tuned to the application and data ingested in question for best results to filter out low utility data.

5.4 Chapter Summary

This chapter contains the evaluation methodology and results of the Twitter search application used to evaluate the proposed CS strategy. CS and it's four implementations were compared against both LS-R and LS-S. Using an empty load handler each implementation was compared against a baseline result set for comparison of accuracy in the applications results. Described in depth in section 5.2 was the evaluation setup, how data sets were collected and organised as well as how the application and experiments were run in Microsoft's cloud platform, Azure. In the same section, detailed explanations of all application and experimental parameters were defined as well as analysis for samples of Subtask Execution Plans (SEPs). Finally, Spearman's Footrule was also described which was used in accuracy evaluation portion of this experimentation.

Section 5.3 outlined each of the facets of the evaluation and their results used to compares CS implementations against LS Implementations.

Section 5.3.7 provides a discussion of the results of the experimentation and section 5.3.8 provides from insight into the drawbacks of CS.

In the next chapter, the conclusions and proposed further work of this thesis are discussed.

Chapter 6

Conclusion and Future Work

6.1 Overview

This thesis presented CS, an alternative approach in tackling bursty data streams in stream computing applications. This approach has the benefit of not discarding input messages intentionally during bursty data events and has been shown to provide results faster and with more accuracy than load shedding alternatives. This chapter summarises the most significant achievements of the work described in this thesis, assesses this work and it's contribution to the research area as well as providing indications for further research work in this area.

6.2 Achievements

This work was motivated by the recent and rapid rise of stream computing applications to power the near-real-time requirement of today's applications. To facilitate this requirement, application architectures have changed from a pull data approach to a push data approach and with it have brought numerous issues in how data at volume is managed and processed.

One sure problem are bursty data streams, where due to phenomenon in the real world, the volume of data generated at source exceeds the applications ability to ingest and process it quickly enough before either results are stale or data is unintentionally lost as input buffers fill and overflow.

A review of existing approaches in solving this problem has shown that some solutions discard input data to alleviate load. Others, where time, resources and cost permit, scale the application to meet new input data rate demands. However, when scaling is not an option, the alternative is to discard input data which in some applications is not an appropriate solution. In particular, in these solutions, data is discarded at a proportional amount and rate to the overloaded input data stream. Thereby leaving a large amount of error in the output application result and large amount of data loss, all in an attempt to ensure result deadlines are upheld.

We defined an alternative novel approach, Computational Shedding (CS), which in appropriate applications can reduce the cost of processing messages in the data stream, thus allowing more messages to be processed in the same time frame during a bursty event. CS improved result accuracy by retaining all input messages when CS was optimised for accuracy targets, while also reducing the cost of processing retained and new messages received by the application. More importantly by reducing the cost of processing messages in the stream, CS can help ensure application deadlines are met while still producing an application result during bursty periods. We further outlined two other selection approaches for SEP entries at runtime which have been shown to reduce message processing times and reduce the cost of SEP selections at runtime.

To implement CS and evaluate it against LS alternatives, a novel stream computing framework, SSCF was designed and developed which has bursty data stream handling semantics designed into it's core unlike other stream computing frameworks which develop them after.

CS was evaluated through a comparison with CS implementations and against a number of existing approaches, through a proof of concept Twitter search application, involving a large real world data set extracted from Twitter. This data set was replayed over a variety of increasing data rates to induce a simulated bursty data stream of various velocities. CS out-performed current load shedding approaches in message processing duration as well as data processing volumes for any given time period. CS also produced an approximate output result, particularly in scenarios of high input data rates which were more accuracy than load shedding alternatives.

CS was developed in a number of versions where each version incorporate a different enhancement to the base concept, attempting to increase accuracy of the applications result during bursty data events as well as reducing the cost of processing messages in time. This approach was used so that the effect of each enhancement could be discerned through comparison with the baseline approach. Each enhancement was shown to produce a significant decrease in message processing duration while also processing more valid messages for a given time frame than load shedding alternatives.

Finally, ultimately CS-A produced the most accuracy results of all load classifiers during overload while also processing input messages faster than other load classifiers or load shedder alternatives.

6.3 Future work

The development and evaluation of computational shedding raised many interesting questions that are worthy of further investigation. This work presents a quantitative analysis from the perspective of a real-time search application ranking real world data from Twitter, replayed to simulate a live data stream. As outlined in chapter 5 section 5.3.8, the general applicability of the CS technique needs to be better understood when applying to a large area of problems given it's strategy is based upon adaptation within the application rather than the data stream. Chapter 1 suggests some application types but this is not a complete list. As such there may be types of applications for which CS is applicable and applications where it is not. As such, a taxonomy of applications types where CS is applicable should be investigated.

6.3.1 SEP Checkpointing

As suggested in chapter 5 the CS implementations could benefit from a position improvement when navigating their respective SEPs. There was a time cost within the load classifiers to traverse which can be further reduced by maintaining the current SEP entry from one input message to the next and determine which SEP should be selected from this starting position rather than the first entry in the SEP each time. The expectation here is with further improvement in reducing the cost of determining which SEP entry to select, application throughput and accuracy can increase.

6.3.2 SEP Organisation

As suggested in chapter 5, LC-A produces accuracy and timely results primarily due to it's focus on accuracy when organising it's respective SEP. Consequently the organisation of message processing duration as a result of the focus primarily on accuracy in the same SEP. The organisation approaches presented here could further be improved upon to produce results which are more accuracy to the applications output result with the addition to an appropriate timely selection strategy. Note, Binary Search may not be applicable in this case due to the lack of sorting in the SEP relative to message processing duration.

6.3.3 Online Profiling

Application profiling is required to generate the SEP but currently this remains relatively static after it is initially generated. This is based on input data which, if not reflective of data which the application will process at runtime can skew the determination of processing costs of messages and accuracy in each subtask. Further, given the data stream is assumed to be constantly evolving with no end, the costs attributed to each subtask can drift unintentionally. As such, online application profiling should be investigated to periodically assess if the accuracy and processing duration costs attributed to each subtask in the local container are consistent. Should these values change, the SEP should be regenerated at periodically intervals to allow the load classifiers to select the most appropriate SEP entries which is most relevant for the data in the stream currently.

6.3.4 Bursty Data Stream Benchmark

Now there are an abundance of stream computing applications in both open and closed source. Although, there currently does not exist a common benchmark or set of tools to compare and contrast the performance of such systems, particularity during burty data events. The Linear Roadway Benchmark [13] has long since been abandoned. This is particularly true when measuring adaptation techniques due to overload. A common set of metrics and an accompanying tool set is required to quantify techniques. Further a set of guidelines should be devised as to suggest what techniques should be used when presented with particular types of applications. With an aim to provide the most appropriate technique based on the requirements of the application. For example, maximised message throughput, minimises message processing durations, highest accuracy application results and finally single or multi-threaded local or remote environments.

6.3.5 Computational Shedding And Elastic Scaling

Finally, as the use of cloud computing and container technologies becomes more wide spread, cloud users are becoming acutely aware of the large costs of running in this environment as well as the degradation in application performance during the transition state as stream tasks scale up. An observation first outlined in section 2.6. A combined approach of applying CS and elastic scaling techniques working together could provide for a solution in which fast application adaptation is possible and thus "buy time" for any scaling approach that may be needed there after. A similar suggestion is mentioned in Rivetti et al. [101].

As scoped in chapter 1 CS is concerned with the immediate stream bursty and assumes it is short-lived, while scaling approaches can solve for the longer term burst effects with additional resources. Interestingly, if the bust could be managed using CS techniques alone, scale adaptations may not be necessary and as a result could save monetary costs.

6.4 Summary

This chapter summarised the motivations for, and the most significant achievements of, the work described in this thesis. In addition, suggestions for future work resulting from this thesis were presented.

Bibliography

[1] Daniel Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, C Erwin, Eduardo Galvez, M Hatoun, Anurag Maskey, Alex Rasin, et al. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666. ACM, 2003.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The International Journal on Very Large DataBases*, 12(2):120–139, 2003.

[3] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465355. URL `http://doi.acm.org/10.1145/2465351.2465355`.

[5] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, et al. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884. ACM, 2005.

[6] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, et al. Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884. ACM, 2005.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536229. URL http://dx.doi.org/10.14778/2536222.2536229.

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824076. URL http://dx.doi.org/10.14778/2824032.2824076.

[9] Mayer Alvo and Paul Cabilio. Rank correlation methods for missing data. *Canadian Journal of Statistics*, 23(4):345–358, 1995.

[10] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-443-X. doi: 10.1145/1289612.1289615. URL http://doi.acm.org/10.1145/1289612.1289615.

[11] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 71–71. IEEE, 2006.

[12] H. Andrade, B. Gedik, K. Wu, and P. S. Yu. Scale-up strategies for processing high-

rate data streams in system s. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1375–1378, March 2009. doi: 10.1109/ICDE.2009.116.

[13] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[14] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[15] B Babcock, S Babu, M Datar, and R Motwani. Models and issues in data stream systems. In *Proceedings of the twenty*, January 2002.

[16] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proceedings. 20th International Conference on Data Engineering*, pages 350–361, April 2004. doi: 10.1109/ICDE.2004.1320010.

[17] B Babcock, M Datar, and R Motwani. Load shedding in data stream systems. *Data Streams*, January 2007.

[18] Brian Babcock, Mayur Datar, Rajeev Motwani, et al. Load shedding techniques for data stream systems. In *Proc. Workshop on Management and Processing of Data Streams*. Citeseer, 2003.

[19] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[20] Cagri Balkesen and Nesime Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *International Workshop on Data Management for Sensor Networks (DMSN)*, 2011.

[21] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, New York,

NY, USA, 2013. ACM. ISBN 978-1-4503-1758-0. doi: 10.1145/2488222.2488258. URL `http://doi.acm.org/10.1145/2488222.2488258`.

[22] Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure, Haris Koutsopoulos, and Carlos Moran. IBM infosphere streams for scalable, real-time, intelligent transportation services. *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 1093, January 2010.

[23] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, F. Ye, S. Shao, and D. Schlosnagle. Data stream processing infrastructure for intelligent transport systems. In *2007 IEEE 66th Vehicular Technology Conference*, pages 1421–1425, Sept 2007. doi: 10.1109/VETECF.2007.303.

[24] Eric Bouillet, Mark Feblowitz, Zhen Liu, Anand Ranganathan, Anton Riabov, Schuman Shao, Don Schlosnagle, and Fan Ye. Stream processing based intelligent transport systems. In *Telecommunications, 2007. ITST'07. 7th International Conference on ITS*, pages 1–6. IEEE, 2007.

[25] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003.

[26] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[27] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465282. URL `http://doi.acm.org/10.1145/2463676.2465282`.

[28] Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Mike Stonebraker, Nesime Tatbul, Ying Xing, and Stan Zdonik. *The Aurora and Borealis Stream Processing Engines*, pages 337–359. Springer Berlin Heidelberg,

Berlin, Heidelberg, 2016. ISBN 978-3-540-28608-0. doi: 10.1007/978-3-540-28608-0_17. URL `https://doi.org/10.1007/978-3-540-28608-0_17`.

[29] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X. doi: 10.1145/872757.872857. URL `http://doi.acm.org/10.1145/872757.872857`.

[30] Raymond Cheng, William Scott, Paul Ellenbogen, Jon Howell, Franziska Roesner, Arvind Krishnamurthy, and Thomas Anderson. Radiatus: A shared-nothing server-side web architecture. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 237–250, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987571. URL `http://doi.acm.org/10.1145/2987550.2987571`.

[31] Yun Chi, Philip S. Yu, Haixun Wang, and Richard R. Muntz. *Loadstar: A Load Shedding Scheme for Classifying Data Streams*, pages 346–357. doi: 10.1137/1.9781611972757.31. URL `https://epubs.siam.org/doi/abs/10.1137/1.9781611972757.31`.

[32] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, May 2016. doi: 10.1109/IPDPSW.2016.138.

[33] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, pages 286–299. MIT Press, Cambridge, third edition, 2009.

[34] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[35] Microsoft Corporation. Microsoft .Net Core 2.1 ConcurrentQueue Class. https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1?view=netcore-2.1, . Accessed:2014-3-20.

[36] Microsoft Corporation. Azure Database for MySQL. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/, . Accessed:2018-03-05.

[37] Microsoft Corporation. Azure Storage. https://azure.microsoft.com/en-us/services/storage/, . Accessed:2016-03-05.

[38] Microsoft Corporation. Azure Virtual Machines. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/, . Accessed:2016-03-05.

[39] Microsoft Corporation. An introduction to c# generics. https://msdn.microsoft.com/en-us/library/ms379564%28v=vs.80%29.aspx?f=255&MSPPError=-2147217396, . Accessed:2013-02-23.

[40] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677. URL http://doi.acm.org/10.1145/2187671.2187677.

[41] Xuan Hong Dang, Wee-Keong Ng, and Kok-Leong Ong. Adaptive load shedding for mining frequent patterns from data streams. In *Data Warehousing and Knowledge Discovery*, pages 342–351. Springer, 2006.

[42] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[43] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[44] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, August 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137786. URL https://doi.org/10.14778/3137765.3137786.

[45] Apache Foundation. Apache Zookeper. http://hadoop.apache.org/zookeeper/, 2010. [Online; accessed 19-July-2010].

[46] .Net Foundation. ASP.NET Core 2. `https://blogs.msdn.microsoft.com/webdev/2017/08/14/announcing-asp-net-core-2-0/`, 2017. [Online; accessed 20-October-2017].

[47] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):44, 2014.

[48] B. Gedik, K. Wu, P. S. Yu, and L. Liu. A load shedding framework and optimizations for m-way windowed stream joins. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 536–545, April 2007. doi: 10.1109/ICDE.2007.367899.

[49] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376729. URL `http://doi.acm.org/10.1145/1376616.1376729`.

[50] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[51] Lazaros Gkatzikis and Iordanis Koutsopoulos. Migrate or not? exploiting dynamic task migration in mobile cloud computing systems. *IEEE Wireless Communications*, 20(3):24–32, 2013.

[52] Lukasz Golab and MT Özsu. Issues in data stream management. *ACM SIGMOD Record*, pages 1–10, January 2003.

[53] Rafał Grycuk, Marcin Gabryel, Rafał Scherer, and Sviatoslav Voloshynovskiy. Multi-layer architecture for storing visual data based on wcf and microsoft sql server database. In *International Conference on Artificial Intelligence and Soft Computing*, pages 715–726. Springer, 2015.

[54] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 126–137, June 2010. doi: 10.1109/ICDCS.2010.72.

[55] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2012.24.

[56] Aditi Gupta, Hemank Lamba, Ponnurangam Kumaraguru, and Anupam Joshi. Faking sandy: Characterizing and identifying fake images on twitter during hurricane sandy. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 729–736, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2038-2. doi: 10.1145/2487788.2488033. URL `http://doi.acm.org/10.1145/2487788.2488033`.

[57] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. International World Wide Web Conferences Steering Committee, 2013.

[58] Christopher B Hauser, Jörg Domaschka, and Stefan Wesner. Predictability of resource intensive big data and hpc jobs in cloud data centres. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 358–365. IEEE, 2018.

[59] Shay Hazor. Apache zookeeper .net client. `https://github.com/shayhatsor/zookeeper`. Accessed:2018-02-23.

[60] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2737-4. doi: 10.1145/2611286.2611294. URL `http://doi.acm.org/10.1145/2611286.2611294`.

[61] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. Ibm streams processing language: analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[62] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014. ISSN 0360-0300. doi: 10.1145/2528412. URL `http://doi.acm.org/10.1145/2528412`.

[63] Ulrich Höhle and Stephen Ernest Rodabaugh. *Mathematics of fuzzy sets: logic, topology, and measure theory*, volume 3. Springer Science & Business Media, 2012.

[64] Amanda Lee Hughes and Leysia Palen. Twitter adoption and use in mass convergence and emergency events. *International Journal of Emergency Management*, 6(3):248–260, 2009.

[65] Twitter Inc. Twitter tweet data model. `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/user-object.html`, . Accessed:2013-09-03.

[66] Twitter Inc. Twitter public streams. `https://dev.twitter.com/streaming/public`, . Accessed:2013-09-03.

[67] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 195–202, July 2011. doi: 10.1109/CLOUD.2011.11.

[68] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *Proceedings of the 31st international conference on Very large data bases*, pages 157–168. VLDB Endowment, 2005.

[69] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM, 2006.

[70] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, pages 56–65. ACM, 2007.

[71] Theodore Johnson, Muthu S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*,

SIGMOD '08, pages 1135–1146, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376730. URL `http://doi.acm.org/10.1145/1376616.1376730`.

[72] Hyun Kang. The prevention and handling of the missing data. *Korean journal of anesthesiology*, 64(5):402–406, 2013.

[73] Balakumar Kendai and Sharma Chakravarthy. Load shedding in mavstream: Analysis, implementation, and evaluation. In Alex Gray, Keith Jeffery, and Jianhua Shao, editors, *Sharing Data, Information and Knowledge*, pages 100–112, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70504-8.

[74] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938. ISSN 00063444. URL `http://www.jstor.org/stable/2332226`.

[75] Maurice G Kendall, Sheila FH Kendall, and B Babington Smith. The distribution of spearman's coefficient of rank correlation in a universe in which all rankings occur an equal number of times. *Biometrika*, pages 251–273, 1939.

[76] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. Cloud migration: A case study of migrating an enterprise it system to iaas. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 450–457, July 2010. doi: 10.1109/CLOUD.2010.37.

[77] Alireza Khoshkbarforoushha, Rajiv Ranjan, Raj Gaire, Ehsan Abbasnejad, Lizhe Wang, and Albert Y Zomaya. Distribution based workload modelling of continuous queries in clouds. *IEEE Transactions on Emerging Topics in Computing*, 5(1):120–133, 2017.

[78] Jooho Kim and Makarand Hastak. Social network analysis: Characteristics of online social networks after a disaster. *International Journal of Information Management*, 38(1):86–96, 2018.

[79] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[80] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter

heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742788. URL `http://doi.acm.org/10.1145/2723372.2742788`.

[81] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, pages 300–312, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35173-0.

[82] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *International Semantic Web Conference*, pages 280–297. Springer, 2013.

[83] Kwei-Jay Lin, Swaminathan Natarajan, and Jane W-S Liu. Imprecise results: Utilizing partial computations in real-time systems. Technical report, Illinois Univ.; Dept. of Computer Science.; Urbana-Champaign, IL, United States, 1987.

[84] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1143-4. doi: 10.1145/2320765.2320789. URL `http://doi.acm.org/10.1145/2320765.2320789`.

[85] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1143-4. doi: 10.1145/2320765.2320789. URL `http://doi.acm.org/10.1145/2320765.2320789`.

[86] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410, June 2015. doi: 10.1109/ICDCS.2015.48.

[87] Steven H Low and David E Lapsley. Optimization flow control—i: basic algorithm and convergence. *IEEE/ACM Transactions on Networking (TON)*, 7(6):861–874, 1999.

[88] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628144. URL `http://doi.acm.org/10.1145/2628136.2628144`.

[89] Michael Mathioudakis and Nick Koudas. Twittermonitor: trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1155–1158. ACM, 2010.

[90] Alexandru Moga, Irina Botan, and Nesime Tatbul. Upstream: Storage-centric load management for streaming applications with update semantics. *The VLDB Journal*, 20 (6):867–892, December 2011. ISSN 1066-8888. doi: 10.1007/s00778-011-0229-7. URL `http://dx.doi.org/10.1007/s00778-011-0229-7`.

[91] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 76–88, March 2010. doi: 10.1109/ICDE.2010.5447867.

[92] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005. ISSN 1551-305X. doi: 10.1561/0400000002. URL `http://dx.doi.org/10.1561/0400000002`.

[93] Microsoft Developer Network. Normalization and sorting. `https://msdn.microsoft.com/en-us/library/ms230117(v=vs.100).aspx`. Accessed:2014-10-03.

[94] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010. doi: 10.1109/ICDMW.2010.172.

[95] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. (1999-66), November 1999. URL `http://ilpubs.stanford.edu:8090/422/`. Previous number = SIDL-WP-1999-0120.

[96] Leysia Palen, Kate Starbird, Sarah Vieweg, and Amanda Hughes. Twitter-based information distribution during the 2009 red river valley flood threat. *Bulletin of the American Society for Information Science and Technology*, 36(5):13–17, 2010.

[97] S Parsons, Mark Weal, Nat O'Grady, and Peter Atkinson. Social media in emergency management: Exploring twitter use by emergency responders in the uk. *International Journal of Emergency Management*, 2017.

[98] Thao N. Pham, Nikos R. Katsipoulakis, Panos K. Chrysanthis, and Alexandros Labrinidis. Uninterruptible migration of continuous queries without operator state migration. *SIGMOD Rec.*, 46(3):17–22, October 2017. ISSN 0163-5808. doi: 10.1145/3156655.3156659. URL `http://doi.acm.org/10.1145/3156655.3156659`.

[99] Owen Phelan, Kevin McCarthy, and Barry Smyth. Using twitter to recommend real-time topical news. In *Proceedings of the Third ACM Conference on Recommender Systems*, RecSys '09, pages 385–388, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-435-5. doi: 10.1145/1639714.1639794. URL `http://doi.acm.org/10.1145/1639714.1639794`.

[100] E. Poormohammady, J. H. Reelfs, M. Stoffers, K. Wehrle, and A. Papageorgiou. Dynamic algorithm selection for the logic of tasks in iot stream processing systems. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–5, Nov 2017. doi: 10.23919/CNSM.2017.8256009.

[101] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. Load-aware shedding in stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 61–68, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4021-2. doi: 10.1145/2933267.2933311. URL `http://doi.acm.org/10.1145/2933267.2933311`.

[102] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: Real-time event detection by social sensors. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 851–860, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772777. URL `http://doi.acm.org/10.1145/1772690.1772777`.

[103] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web*, pages 851–860. ACM, 2010.

[104] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[105] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 53–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370826. URL `http://doi.acm.org/10.1145/2370816.2370826`.

[106] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 25–36, January 2003.

[107] Mehul Arunkumar Shah. *Flux: A Mechanism for Building Robust, Scalable Dataflows.* PhD thesis, University of California, Berkeley, 2004.

[108] Georgios L Stavrinides and Helen D Karatza. A cost-effective and qos-aware approach to scheduling real-time workflow applications in paas and saas clouds. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 231–239. IEEE, July 2015.

[109] Georgios L Stavrinides and Helen D Karatza. Scheduling real-time parallel applications in saas clouds in the presence of transient software failures. In *2016 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–8. IEEE, Aug 2016.

[110] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, December 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107504. URL `http://doi.acm.org/10.1145/1107499.1107504`.

[111] Emine Nesime Tatbul. *Load Shedding Techniques for Data Stream Management Systems.* PhD thesis, Providence, RI, USA, 2007. AAI3272068.

[112] N. Tatbul. Streaming data integration: Challenges and opportunities. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)(ICDEW)*, volume 00, pages 155–158, 03 2010. doi: 10.1109/ICDEW.2010.5452751. URL `doi.ieeecomputersociety.org/10.1109/ICDEW.2010.5452751`.

[113] N. Tatbul and S. Zdonik. Dealing with overload in distributed stream processing systems. In *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, pages 24–24, April 2006. doi: 10.1109/ICDEW.2006.45.

[114] Nesime Tatbul. Qos-driven load shedding on data streams. In *XML-Based Data Management and Multimedia Engineering—EDBT 2002 Workshops*, pages 566–576. Springer, 2002.

[115] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 799–810. VLDB Endowment, 2006. URL `http://dl.acm.org/citation.cfm?id=1182635.1164196`.

[116] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003. ISBN 0-12-722442-4. URL `http://dl.acm.org/citation.cfm?id=1315451.1315479`.

[117] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[118] Nesime Tatbul, Yanif Ahmad, Uğur Çetintemel, Jeong-Hyon Hwang, Ying Xing, and Stan Zdonik. Load Management and High Availability in the Borealis Distributed Stream Processing Engine. *Lecture Notes in Computer Science*, pages 66–85, February 2008.

[119] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156,

New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5. doi: 10.1145/2588555.2595641. URL `http://doi.acm.org/10.1145/2588555.2595641`.

[120] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132750. URL `http://doi.acm.org/10.1145/3132747.3132750`.

[121] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544, June 2014. doi: 10.1109/ICDCS.2014.61.

[122] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, volume 00, pages 22–31, April 2016. doi: 10.1109/IC2E.2016.38. URL `doi.ieeecomputersociety.org/10.1109/IC2E.2016.38`.

[123] Dave Yates and Scott Paquette. Emergency knowledge management and social media technologies: A case study of the 2010 haitian earthquake. In *Proceedings of the 73rd ASIS&T Annual Meeting on Navigating Streams in an Information Ecosystem - Volume 47*, ASIS&T '10, pages 42:1–42:9, Silver Springs, MD, USA, 2010. American Society for Information Science. URL `http://dl.acm.org/citation.cfm?id=1920331.1920391`.

[124] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *VLDB Endowment*, 4(11), 2011.

[125] L. Zhang, J. Lin, and R. Karim. Sliding window-based fault detection from high-dimensional data streams. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(2):289–303, Feb 2017. ISSN 2168-2216. doi: 10.1109/TSMC.2016.2585566.