Performance Modelling of Distributed Stream Processing Topologies

Thomas Cooper

Submitted for the degree of Doctor of Philosophy in the School of Computing, Newcastle University

August 2020

For Amanda and Gillian

Abstract

Distributed stream processing systems (like Apache Storm, Heron and Flink) allow the processing of massive amounts of data with low latency and high throughput. Part of the power of these systems is their ability to scale the separate sections (operators) of a stream processing query to adapt to changes in incoming workload and maintain end-to-end latency or throughput requirements.

Whilst many of the popular stream processing systems provide the functionality to scale their queries, none of them suggest to the user how best to do this to ensure better performance. The user is required to deploy the query, wait for it to stabilise, assess its performance, alter its configuration and repeat this loop until the required performance is achieved. This scaling decision loop is intensely time consuming and for large deployments the process can take days to complete. The time and effort involved also discourage users from changing the configuration once one is found to satisfy peak load. This leads to the over-provisioning of resources.

To solve these issues a performance modelling system for stream processing queries is required. This would allow the performance effect of changes to a query's configuration to be assessed before they are deployed, reducing the time spent within the scaling decision loop. Previous research on auto-scaling systems using performance models has focused either; on queueing theory based approaches, which require small amounts of historical performance data but suffer from poor accuracy; or on machine learning based approaches which have better accuracy, but require large amounts of historical performance data to produce their predictions.

The research detailed in this thesis is focused on showing that an approach based on queueing theory and discrete event simulation can be used, with only a relatively small amount of performance data, to provide accurate performance modelling results for stream processing queries.

We have analysed the many aspects involved in the operation of modern stream processing systems, in particular Apache Storm. From this we have created processes and models to predict end-to-end latencies for proposed configuration changes to running streaming queries. These predictions are validated against a diverse range of example streaming queries, under various workload environments and configurations. The evaluations show that for most query configurations, our approach can produce performance predictions with similar or better accuracy than the best performing machine learning based approaches, over a more diverse range of query types, using only a fraction of the performance data those approaches require.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Thomas Cooper

August 2020

Acknowledgements

I would like to start by thanking my supervisor, Dr. Paul Ezhilchelvan, for his advice, patience, good humor and honesty (even if sometimes I didn't want to hear it). It has been a journey.

Thanks to my fellow MSc Computer Science students for making my introduction to the field so much fun and for their continued friendship and support. Special thanks to Soo Darcy for her excellent proof reading skills.

My thanks to the staff of the Newcastle University Centre for Doctoral Training (CDT) in Cloud Computing for Big Data: to Drs. Paul Watson and Darren Wilkinson for leading the CDT and taking a chance on a masters student with some funny ideas; to Drs. Matt Foreshaw and Sarah Heaps for giving me a great start and support along the way; and thanks especially to Oonagh McGee and Jen Wood, without whom chaos would have reigned.

Thanks to Karthik Ramasamy, Ning Wang and all the Twitter Real Time Compute team for giving me a chance to test my ideas and learn all I could from them.

Thanks to my fellow CDT PhD students for sharing their knowledge, experience and friendship. Special mentions go to Naomi Hannaford and Lauren Roberts for being far more patient with the loud guy sitting next to them than they needed to be; and to Dr. Hugo Firth who is a good friend and fellow shaver of yaks.

Thank you to my family, who have supported me throughout my (multiple) academic endeavours: to my father and grandfathers, for giving me my love of technology; and to my mother Amanda, who gave me the means to start my journey toward this PhD and who never wavered in her belief in me.

Finally, love and thanks to Gillian, without whom none of this would have been possible and who already knows everything I want to say.

Contents

List of Tables

List of Figures

Glossary

API application programming interface.

ARIMA auto-regressive integrated moving average.

bolt The main type of component in a topology query plan. They contain the user defined tuple processing code. Each executor assigned to a topology will run the code for a single bolt.

CDF cumulative distribution function.

CEP complex event processing: A set of concepts and techniques for processing real-time events and extracting information from event streams as they arrive. The goal of complex event processing is to identify meaningful events (such as opportunities or threats) in real-time situations and respond to them as quickly as possible.

complete latency The time taken for all child tuples, produced from a source tuple, to be processed fully.

component The elements of a topology's query plan. The spouts or bolts which are defined by the user.

CPU central processing unit.

CSV comma separated values.

DAG directed acyclic graph.

DBPS distributed batch processing system.

DES discrete-event simulation: A simulation which models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next.

DiG directed graph.

Disruptor queue Apache Storm's custom queue implementation using the LMAX Disruptor data structure which allows non-blocking movement of items on and off the queue. These queues are used throughout the Storm system in both the executor and worker processes.

DSPS distributed stream processing system.

end-to-end latency The time taken for a tuple to traverse the entire topology from being issued by the source spout to completing service in the final sink bolt.

ERQ executor receive queue.

ESQ executor send queue.

EST executor send thread.

ETIP estimated task input proportion.

ETOP estimated task output proportion.

executor The processing units within a topology's logical plan. They contain the tasks which do the actual processing of tuples and coordinate the input and output tuples for those tasks.

GP Gaussian process.

ground truth latency The latency measure used in the evaluation experiments. It is the time from the moment a message is pulled off the Kafka Message broker in the spout executors of the test topology to just before being sent back to the message broker in the final sink component of the test topology.

GRP global routing probability.

I/O input to output.

ICMP Internet Control Message Protocol.

JSON JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serialisable value).

JVM Java Virtual Machine.

Kappa Architecture A replacement for the Lambda Architecture which removes the batch processing element and uses a single unified stream processing approach, relying on a fault tolerant and accurate system to process all queries.

Lambda Architecture The combination of a fast and potentially inaccurate stream processing system, to provide rapid responses, with a periodically run batch processing system that will provide accurate responses at a slower rate.

logical plan A representation of a topology showing how the executors assigned to each component are connected together.

LTF local transfer function.

MAPE monitor, analyse, plan, execute.

MPC model predictive control.

MTIP measured task input proportion.

MTOP measured task output proportion.

MVA mean value analysis: A recursive technique for computing expected queue lengths, waiting time at queueing nodes and throughput in equilibrium for a closed separable system of queues.

OS operating system.

parallelism When used in reference to a Storm component (spout or bolt), the parallelism of the component refers to the number of executors assigned to that component.

physical plan A representation of a topology showing how the executors assigned to each component (spouts or bolts) are distributed across the various worker processes which are running on the cluster of worker nodes.

QoS quality of service.

query plan A high level representation of a topology showing how the user defined components of the streaming query are connected together. This will also show the type of connection (stream grouping) between components.

queueing theory The mathematical study of waiting in lines or queues. From this theory queueing models are produced so that waiting times and queue lengths can be predicted.

RAM random access memory.

reinforcement-learning A supervised learning approach where a cost function is applied to previous decisions in order to inform future decisions.

RMSE root mean squared error.

RTM real time monitor.

scheduling The act of creating a physical plan by assigning operator replicas to workers. This process is carried out by a *scheduler* implementation.

SerDes Short from of serialisation/de-serialisation and refers to the process of converting native language objects into byte stream representations (and back again) for transfer across a network connection..

SLA service level agreement.

sojourn time The total time spent at a queueing node, including the queue waiting time and the service time.

SOP stream output proportion.

spout A special type of component in a topology query plan which is responsible for receiving events from outside of the system and creating tuples to be passed into the topology. Spouts are also responsible for acknowledging the completed processing of source events to external systems.

SRP stream routing probability.

Storm-Tracer The name for the Apache Storm performance modelling system developed as part of this doctoral research.

SVM support-vector-machine.

SVR support-vector-regression.

task Tasks contain the user defined tuple processing code and are used to facilitate stream and state partitioning with topology. Multiple tasks can be housed within an executor and the number of tasks a component has is set for the lifetime of the topology.

topology A directed graph of operators which perform actions on a stream of continuously arriving data.

topology configuration A set of integers indicating the parallelism hint, the number of replicas (executors), for each component of the topology and the number of worker processes assigned to the topology.

TRP task to task routing probability.

TSDB time series database.

tuple In mathematics, a tuple is a finite ordered sequence of elements. In Apache Storm a tuple is the data structure which passes between the logical operators (executors)

in the topology. A Storm tuple has one or more *fields* which are keys linking to data values.

tuple flow plan A representation of a topology which includes all the internal Storm elements a tuple will pass through as well as the logical operators.

ts^{-1} tuples per second.

tweet A short 140 (now 280) character message posted to the micro-blogging platform Twitter (https://twitter.com). These messages can contain "hashtags" which denote topics of interest, location information, images, videos and other meta-data. Tweets are publicly visible by default and analytics and real-time or historic tweet feeds can be purchased from Twitter.

ULT user logic thread.

VM virtual machine.

worker process A JVM instance which contains and coordinates the executor threads and various internal queue implementations associated with them.

worker node A virtual or physical machine which runs one or more worker processes.

WPST worker process send thread.

WPTQ worker process transfer queue.

Chapter 1

Introduction

Processing big data is not a new phenomenon in computing, it has been present since its earliest days. The only thing that has changed is what we consider *big* to mean. Towards the end of the last century, hundreds of thousands of items and multiple megabytes were pushing the boundaries of what was possible to process. Now, thanks to research projects like the Large Hadron Collider and the ambitions of companies like Google, trillions of items and petabytes of data need to be processed on a regular basis.

In the past, the solution to *big* data issues was typically to scale vertically, where the code is run on machines that have faster processors, more memory and/or larger storage. However in recent years, thanks in part to the advent of cloud computing, horizontal scaling — where computation is spread across several relatively low powered machines — has become the norm. This has led to the emergence of many distributed big data processing frameworks such as Google's MapReduce (Dean & Ghemawat, 2008), Apache Hadoop (Hadoop, 2019) and Apache Spark (Zaharia et al., 2010). These distributed batch processing system (DBPS) allow for huge amounts of data, 20 petabytes a day or more (Zaharia et al., 2008), to be processed. However, these systems are focused on processing large *volumes* of data, not on processing high *velocity* data.

In recent years it is not only the size of the data that needs processing which has required innovation, but also the rate at which this data arrives and the timeliness of operations performed on it. Social networks, such as Twitter[1], are focused on "What is happening now?" and so need to analyse trends across the hundreds of millions of daily posts their users produce in real time (Toshniwal et al., 2014). This real time requirement means that batch processing systems which run daily, hourly or even every minute are not appropriate. This has led to the development of distributed stream processing systems (DSPSs) such as Apache Storm (Toshniwal et al., 2014), Spark Streaming (Zaharia et al., 2013) and Apache Flink (Carbone et al., 2015).

[1] see: https://about.twitter.com/

DSPSs are often combined with the DBPSs mentioned above into what is commonly referred to as the *Lambda Architecture* (Marz & Warren, 2015). This architecture allows operators to leverage the low latency of a stream processing system for queries which are time, but not accuracy, critical and use the slower batch processing system for those queries that require accurate results across a whole dataset. The Lambda Architecture has been widely adopted in industry, however running two concurrent systems obviously involves duplicated effort, additional resources and increased costs. To address this, industry has started to move towards eliminating the batch processing elements of the Lambda Architecture and using the stream processing system for all queries. This approach, known as the *Kappa Architecture* (Kreps, 2014), requires solving several issues with the current DSPSs around their message delivery guarantees, fault tolerance and automation. This focus on improving stream processing systems has led to increased interest in research related to DSPSs and their operation, which in turn provided the impetus for this PhD research.

1.1 Distributed Stream Processing

DSPSs allow operations on streams of constantly arriving data to be distributed across a cluster of workers (physical or virtual machines). The operators form a directed graph, commonly referred to as a *topology*, which defines the sequence of operations on the incoming data. Figure 1.1 illustrates a DSPS topology, with a source component (S) that inserts data packets into the topology and several operators which consume these data packets, perform operations on them and produce zero or more new data packets to be passed *downstream* to the next operator in the topology.

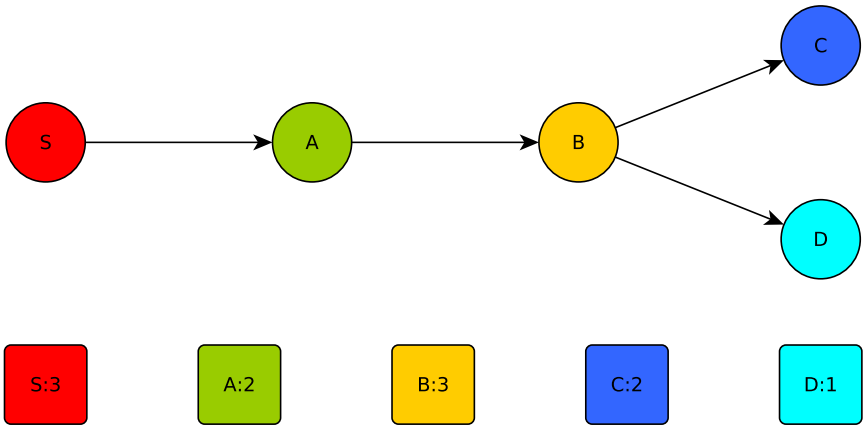

Figure 1.1: An example stream processing topology and replication level of each operator.

If needed, operators can be replicated across the cluster of workers to reduce latency and increase throughput. Slow operators and/or those expecting high traffic can have higher numbers of replicas, to increase parallel processing, while fast or low traffic operators

may require fewer replicas to keep up with the incoming data packet arrival rate. The numbers, shown in the squares below the operators in figure 1.1, give an example level of replication for each operator. Figure 1.2 shows an example of how the copies of each of those operators could be distributed across a cluster of three workers.

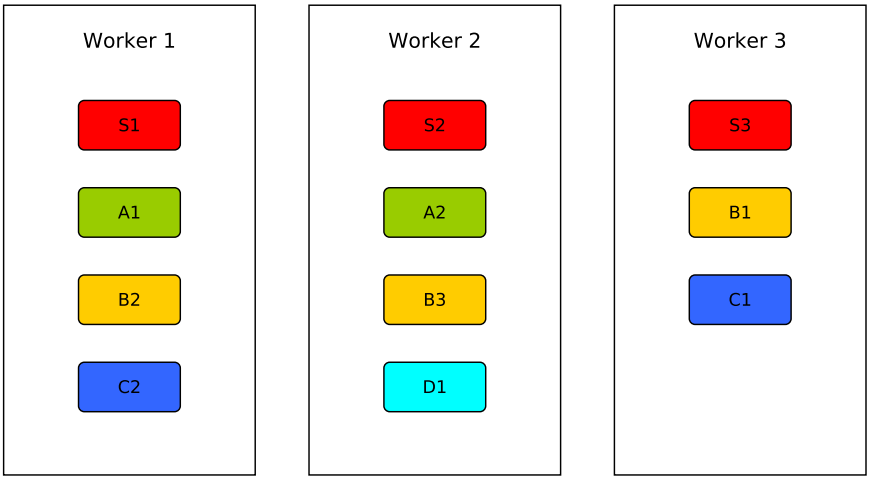

Figure 1.2: A cluster layout for the operators of the topology shown in figure 1.1.

The creation of this operator layout on the cluster, commonly called the topology's *physical plan*, is referred to as *scheduling*. The physical plan of a topology can have a significant effect on the performance of that topology. Placing replicas of two connected operators, which send large amounts of traffic to one another, on separate worker nodes will mean that traffic has to travel across the network, incurring additional transfer latency. Placing high traffic operators on the same node may remove the network transfer latency between them but, as these operators are now on the same machine, they are competing for the same finite resources and so this may result in slower processing of the data packets.

1.2 Topology Scaling Decisions

Many of the popular DSPSs provide functionality to change the physical plan of a topology whilst it is running. However, to the best of our knowledge, all but one[2] of the mainstream DSPSs[3] have no mechanisms to *automatically* scale or reschedule their resources in response to the size and rate of the incoming data (the topology's incoming workload). This means that the current DSPSs rely on a human administrator's domain knowledge and experience in order to scale a topology to the correct size to meet peak incoming workload. The administrator will typically engage in an iterative approach using the following steps:

1) Choose the parallelism of each topology operator.

[2]Apache Heron — see chapter 3 for more details
[3]Apache Storm, Spark or Flink

2) Use their DSPS's scheduler to create a physical plan for their chosen configuration. Certain schedulers may give additional control over worker node resources and operator replica placement as well as operator parallelism.
3) Deploy that physical plan to the DSPS cluster.
4) Wait for the deployment to finish and for the data flow through the topology to stabilise. Depending on the size of the topology this can take a significant amount of time and is often overlooked in DSPS design (Heinze, Jerzak, et al., 2014).
5) Wait for a further period to gather performance metrics. This can involve waiting for a sufficiently high input workload to arrive (which may require waiting for a specific time of day) or the creation of an accurate test load generator.
6) Assess if the topology performance is meeting the specified targets.

This *schedule* → *deploy* → *stabilise* → *analyse* loop (see figure 1.3) is a time intensive process. In the worst case, for large, complex and high-traffic topologies (like those used in production environments), finding a configuration that will maintain performance in the presence of peak input workload can take several weeks to complete (Graham et al., 2017).

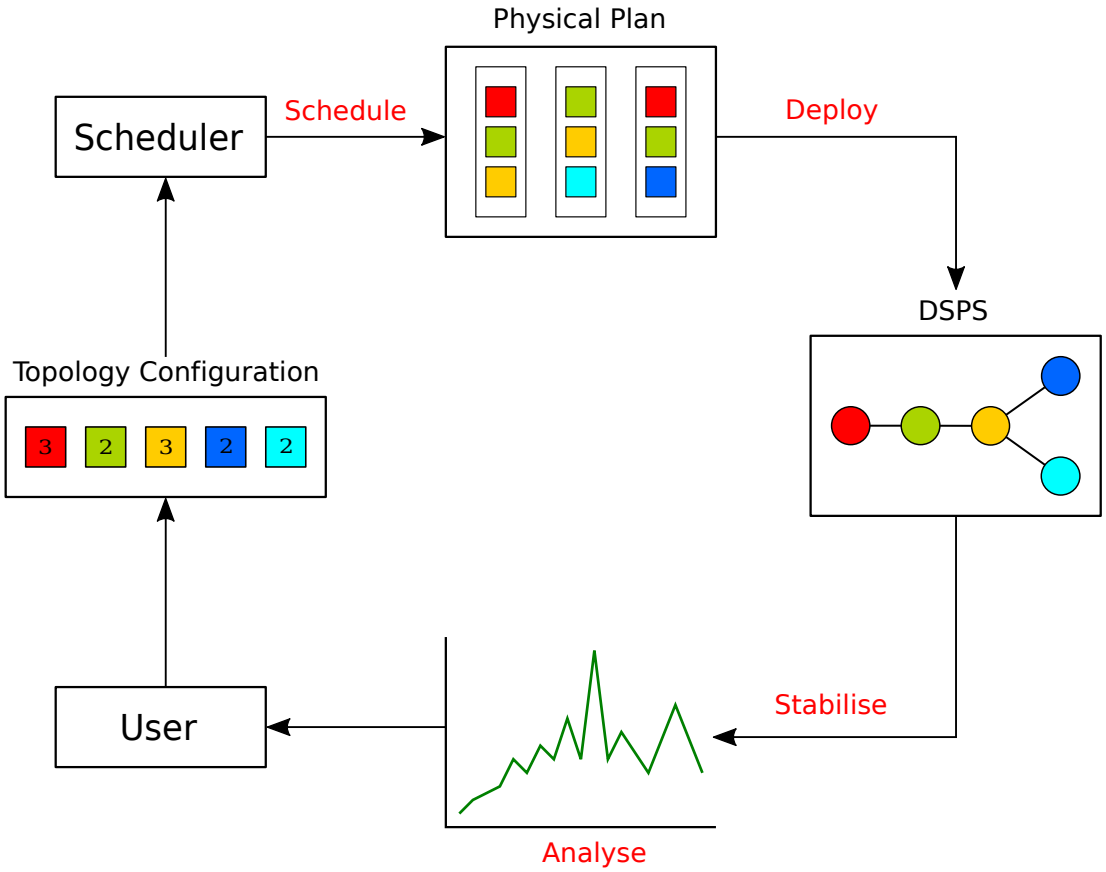

Figure 1.3: The schedule → *deploy* → *stabilise* → *analyse* scaling decision loop with a human user.

Scheduling a topology's physical plan is a known NP-complete problem (Fernandez-Baca, 1989), where multiple optimum plans exist for a given set of constraints. There are numerous examples in the literature of automatic scheduling algorithms that use a variety

of approaches to obtain an optimum plan (see chapter 3 for more details). However, whilst these systems allow the removal of the human administrator from the scaling decision loop, they still require multiple iterations of the full loop in order to converge on a valid plan (see figure 1.4). Even the advanced machine learning algorithms, described in chapter 3, cannot avoid the time penalty of repeated iterations of this loop.

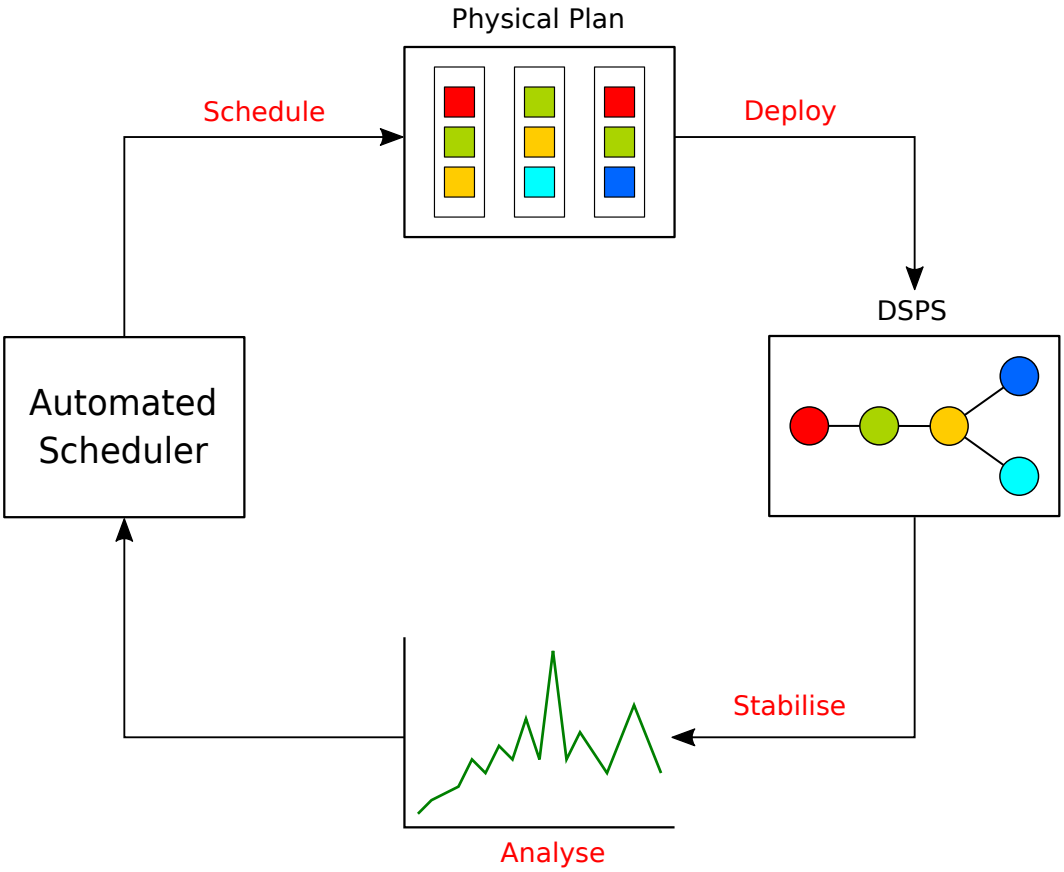

Figure 1.4: The schedule → *deploy* → *stabilise* → *analyse* scaling decision loop with an auto-scaling system included.

1.3 Model-Based Scaling Decisions

In order to reduce the time taken to make scaling decisions, the loop shown in figure 1.4 needs to be shortened. Ideally, the time costly phases of deployment, stabilisation and analysis should be performed as few times as possible, preferably once. What is required, therefore, is a way to assess if a physical plan is likely to meet a performance target *before* it is deployed to the DSPS cluster.

This implies the need for a performance modelling system capable of assessing a proposed physical plan. This system should be able to use recent metrics data, collected from a running topology, and predict the end-to-end latency and throughput of that topology as if it were configured according to the proposed plan. Such a system could not only address the time taken to converge on a valid physical plan (as discussed in section 1.3.1)

but also provides several other advantages (described in sections 1.3.2, 1.3.3, 1.3.4) over the current topology scaling approach described in section 1.2.

1.3.1 Faster convergence on a valid plan

The inclusion of a performance modelling system into the scaling decision loop would allow schedulers to decide whether a plan is appropriate before it is deployed, avoiding the significant costs that are incurred for that operation (Heinze, Jerzak, et al., 2014). This would also allow an auto-scaling system to quickly iterate to an initial deployment plan based on some preliminary metrics rather than iterating over the full scaling loop or, in the case of reinforcement learning auto-scaling approaches, waiting for an extended training period (with a sufficient variation in states) to be completed.

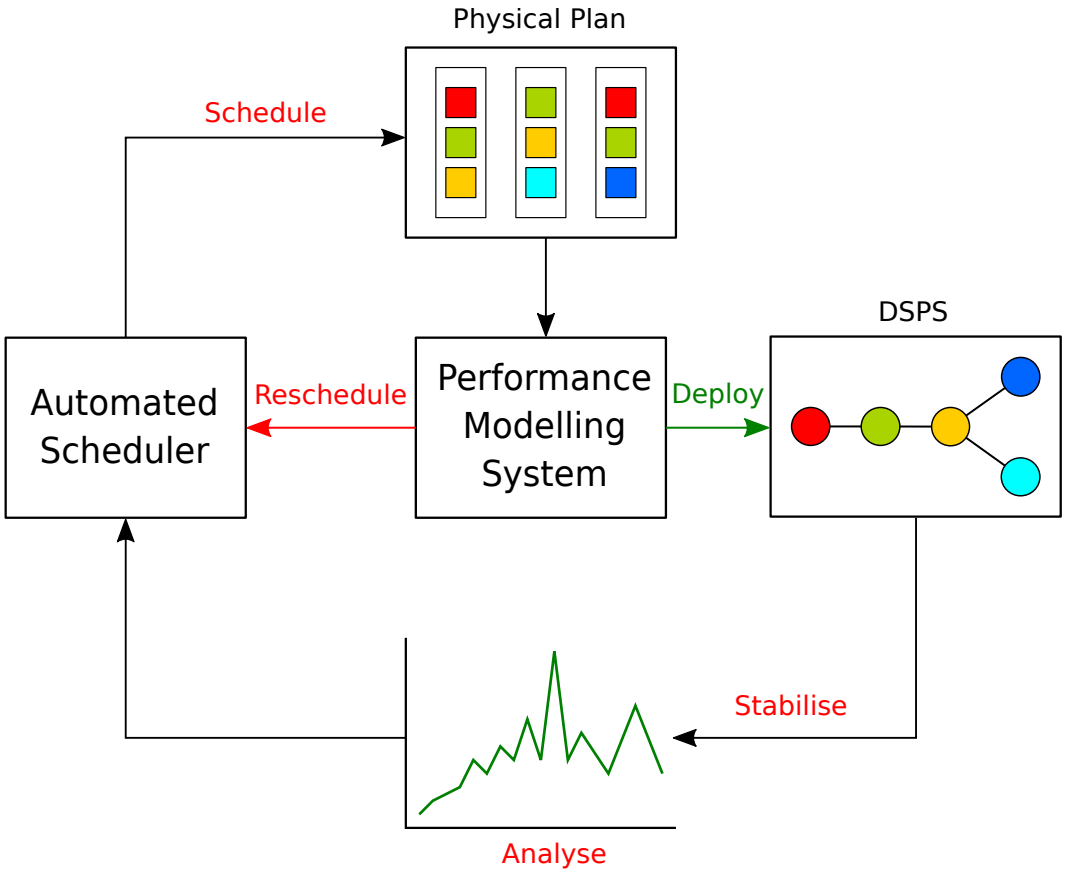

Figure 1.5: An auto-scaling system paired with a performance modelling system.

Figure 1.5 shows the effect of adding a performance modelling system into the scaling decision loop. The deployment section of the loop is short-circuited through the modelling system. Assuming an efficient modelling process, the *schedule* → *model* → *deploy* loop should be significantly quicker at finding a physical plan to meet a given performance target than the *schedule* → *deploy* → *stabilise* → *analyse* loops shown in figure 1.3 and figure 1.4.

1.3.2 Feedback for scheduling decisions

An additional advantage that a performance modelling system would provide is detailed feedback on a proposed physical plan's performance. The modelling system could return detailed breakdowns of aspects of the plan which did not meet given performance criteria. Bottlenecks within the topology's data flow, overloading of individual worker nodes and many other issues could be identified for a proposed physical plan. The scheduler could then use this information to make more informed scaling decisions, potentially reducing the total number of iterations required to converge on a valid physical plan.

1.3.3 Pre-emptive scaling

In order to predict the performance of a proposed physical plan, the modelling system also needs to know the expected incoming workload (Υ) into the topology. The system could simply assume that the workload level will remain unchanged, but the imperative to perform a scaling action on the topology is usually due to a change in workload and so some notion of what that new workload level is likely to be must be provided.

A pessimistic approach to workload prediction could be to take the expected peak load into the topology, discerned from historical data, and find a physical plan that could perform at the target performance level (T_p) under that load. However, this would lead to significant over provisioning and ignores one of the key features of DSPS, namely the ability to scale dynamically as input workload changes.

If incoming workload levels could be forecast some time (τ_f) into the future, then this would allow an auto-scaling system to model the effect of that predicted workload level ($\hat{\Upsilon}$) on the currently running physical plan. If the predicted performance level ($T_p^{\hat{\Upsilon}}$) of the proposed physical plan does not meet the required performance level (T_r) for the predicted workload level, then the system can *pre-emptively* begin the scaling operations. Ideally τ_f would be longer than the time taken to perform the modelling process (τ_{model}) and for the topology scaling operations to complete (τ_{scale}). If $\tau_f > \tau_{\text{model}} + \tau_{\text{scale}}$ then the scaling operation could be completed before the new workload level was expected to arrive and the performance target could be maintained. This processes is illustrated in figure 1.6.

Forecasting incoming workload levels is an entire field of study in itself. However, several *off-the-shelf* approaches exist for forecasting of time series data and could be used in conjunction with the performance modelling system. There are also issues around when to perform the forecasting and how to take forecasting and performance modelling error into account in decision of when and how to begin scaling. However, it is clear that a performance modelling system is a key requirement in any pre-emptive scaling system.

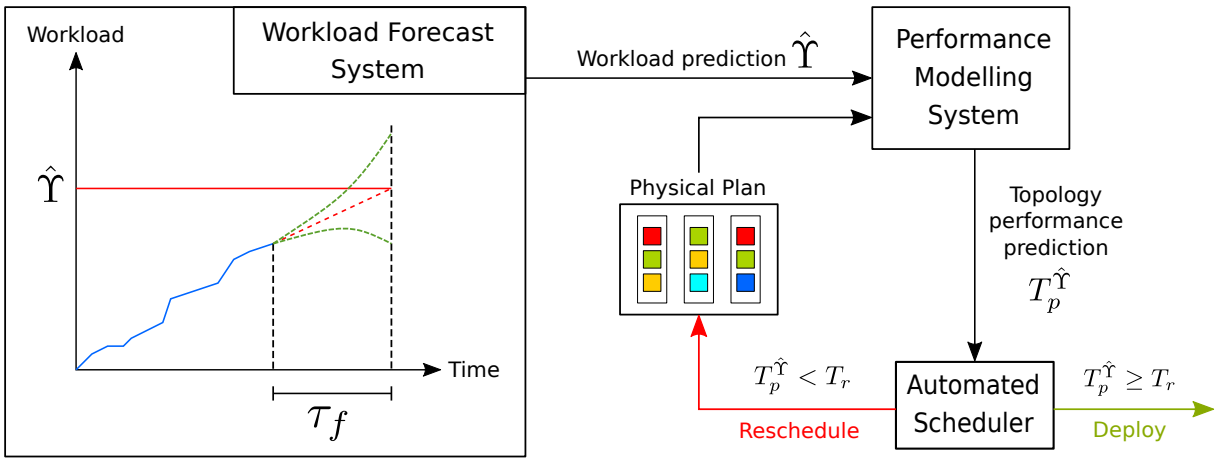

Figure 1.6: The pre-emptive scaling process made possible by combining a performance modelling system with an incoming workload forecasting system.

1.3.4 N-version scaling

Another significant advantage to having a modelling system for proposed physical plans is that it allows the output from different schedulers to be compared. When using the current scaling decision process (see figure 1.3) or the auto-scaling processes proposed in the literature (see figure 1.4), only a single scheduler can be used. The scaling loop occupies a significant amount of time and therefore repeating the loop additional times for different schedulers would be prohibitively expensive, not to mention the fact that currently there is no way for a scheduler to differentiate between different physical plans. However, with a modelling system, multiple physical plans can be produced and compared in parallel.

As scheduling of DSPS topologies is an NP-Complete problem there are diverse approaches for producing physical plans for a given topology. Each has been designed with a specific use case and set of underlying assumptions in mind. Depending on the application that the topology is designed for, as well as its deployment environment, one or another of these proposed schedulers will be more appropriate. With a performance modelling system the user of a DSPS would not need to determine *a priori* which scheduler was the most appropriate for their use case. They, or an automatic scaling system, could simply employ a range of schedulers and use the modelling system to compare their performance and select the best one.

This approach also has advantages with regard to possible errors in the scheduler implementations. For example, if a given scheduler has a bug, multiple other implementations with the same optimisation goals (computing resource, network traffic, etc.) could be compared in parallel and matching plans from a majority of schedulers chosen to reduce the chances of a 'buggy' plan being chosen. This is a form of N-version programming from fault tolerant software design (Chen & Avizienis, 1995).

1.4 Summary

DSPSs provide the functionality to process high volumes of data at high velocity. However, whilst many of these systems provide the functionality to scale up (and down) to match incoming workloads, all but one (Apache Heron) have no way to do this automatically. Proposed solutions in the literature focus on the problem of creating optimal physical plans and not on the time taken to deploy, stabilise and check that a scaling decision has met a given performance target.

The introduction of a performance modelling system, for proposed physical plans from any of the many schedulers proposed in the literature (see section 3), would allow the expensive scaling decision loop to be shortened. Such a performance modelling system provides several advantages:

- Reduces the time to find a viable physical plan to meet a given performance service level agreement (SLA).
- Can provide detailed feedback on the performance of a proposed plan that can be used to inform the scheduler's future decisions.
- When coupled with a workload prediction system, can facilitate pre-emptive scaling by allowing a physical plan that can satisfy the SLA in the face of the predicted workload to be deployed before that workload arrives.
- Allows multiple physical plans from different scheduler implementations to be compared in parallel. Allowing the user to avoid the task of choosing the most appropriate scheduler for their workload and application domain.

1.4.1 Research aims

The research described in this thesis has several aims:

- Create a performance modelling system for DSPS queries (topologies).
- The system should have the ability to model any proposed physical plan created by a scheduler implementation. This includes various streaming operator types (windowing, joining, splitting, etc.) and connections between them (load balanced or key based routing).
- The system should be able to provide performance estimates using a minimum amount of input data and avoid the need for extensive calibration or training periods. Such periods would remove the advantage of adding a performance modelling system by requiring many *deploy* $\rightarrow$ *stabilise* cycles to be carried out in order to provide the calibration/training data.

The central hypothesis of this thesis is that the above aims can be achieved using an approach based on queuing theory and discrete-event simulation (DES), without the need

to resort to using machine learning based approaches and the onerous training and data requirements they entail (see chapter 3 for more details).

1.5 Thesis Structure

Chapter 2 describes the process of choosing the example DSPS, Apache Storm, and goes on to describe in detail the relevant aspects of Storm's operation. Chapter 3 details the previous work in the field of DSPS automation and performance modelling. Chapter 4 describes the approach taken to modelling the performance of Apache Storm topologies. Chapter 5 analyses the accuracy of the modelling system and evaluates its performance. Finally, chapter 6 discusses the results of the doctoral research and looks at future areas of study.

In addition to the main chapters, there are several appendices with supporting information: appendix C details the implementation of the performance modelling system and its supporting infrastructure; whilst appendix D details the outcome of applying the performance modelling approach developed for Apache Storm to Twitter's Heron DSPS as part of a four month internship with the company.

1.6 Related Publications

During the course of my PhD research I have contributed to the following peer-reviewed publications:

- Cooper, T. (2016) 'Proactive scaling of distributed stream processing work flows using workload modelling: Doctoral symposium', in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems - DEBS '16*. pp. 410–413.

  This paper, produced in the first year of my PhD, introduces my research area, defines the problem I was addressing and outlines my proposed solution. The initial focus of my research, as described in this paper, was on pre-emptive scaling (see section 1.3.3), however this subsequently changed to focus solely on the accuracy of the topology performance model.

- Kalim, F., Cooper, T., et al. (2019) 'Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems', in *Proceedings of the 35th IEEE International Conference on Data Engineering*. pp. 1886–1897.

  During my internship at Twitter, detailed in appendix D, I applied the modelling techniques I developed for Apache Storm (see chapter 4) to Twitter's Heron DSPS. As well as a topology performance modelling system, I developed a workload forecasting system for Heron topologies. Following my internship, another PhD student continued

to develop the modelling system (called Caladrius) as part of a subsequent internship and this paper details our combined work in collaboration with Twitter's Real Time Compute team.

- Cooper, T., Ezhilchelvan, P. & Mitrani, I. (2019) 'A queuing model of a stream-processing server', in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. pp. 27–35

  As part of the development of the model used for the predicting the latency of the Storm executors (see section 4.2), various standard queuing models were considered. However, none were deemed to meet the unique characteristics of Storm's queue implementation (see section 2.7). In the end a discrete event simulator was used to model the Storm executors (see appendix B). Towards, the end of my PhD I collaborated with Isi Mitrani and Paul Ezhilchelvan to develop new analytical solutions for modeling these unique queueing systems. This paper details both exact and approximate solutions for modelling the stream processing queues, as well as looking at possible optimisations of the queue parameters.

Chapter 2

Distributed Stream Processing Architecture

2.1 Choosing an Example System

Distributed stream processing systems (DSPSs) share many high-level characteristics. Most represent their streaming queries as directed graphs of operators, commonly referred to as a topologies. They provide ways to have multiple copies of each operator in those topologies, have a notion of a worker entity that contains these operator copies and provide infrastructure to coordinate all these elements. However, in order to efficiently investigate the challenges and possible approaches to performance modelling of DSPS topologies, a single test bed system was required.

This PhD research began in the summer of 2015 and at that time there were multiple DSPSs available. In deciding which system to select, several criteria were used:

- Candidate systems should be open source projects, so that the entire code base could be inspected, altered and any contributions could be released publicly.
- The chosen DSPS should be under active development. This would ensure that any queries around the operation of the system had a chance of being answered by that system's developers.
- The chosen system should have industry backing and be used in production environments. This would mean that the developed modelling system could be of practical use to industry and be developed and tested on a realistic DSPS workloads and infrastructure.

The criteria above excluded many DSPSs discussed in the literature, such as Borealis (Abadi et al., 2005; Ahmad et al., 2005; Balazinska et al., 2005), S4 (Neumeyer et al., 2010) and STREAM (Arasu et al., 2016), as these systems were no longer under active

development or being used in production environments.

The mainstream, open source DSPSs available in 2015[1] were Apache Storm, Flink (Alexandrov et al., 2014), Samza and Spark Streaming (Zaharia et al., 2013). Of these only Apache Storm satisfied all the criteria listed above.

2.2 Apache Storm Overview

This section provides a brief overview of Apache Storm's architecture and operation. The rest of this chapter gives a more detailed breakdown of the Apache Storm DSPS, its internal structure and how it distributes work across a cluster of machines. This information was sourced from Toshniwal et al. (2014), Kulkarni et al. (2015), the Storm documentation[2] and source code[3]. Please note that the descriptions in this chapter refer specifically to Apache Storm version 1.2.2 (unless otherwise stated).

2.2.1 Background

Apache Storm[4] began life at the market research firm Backtype. It was designed by Nathan Marz (the original proposer of the *Lambda Architecture* (Marz & Warren, 2015)) to perform analysis on the Twitter[5] 'fire hose', the real-time stream of all tweets posted to the social network. To cope with such a large volume of data arriving at high velocity and be able to return analysis results in a timely manner, Storm was designed with massive scale in mind. It provided fault tolerance, a simple application programming interface (API) and was able to dynamically scale its topologies.

In 2011 Twitter acquired Backtype[6], in part to gain access to Storm which they considered better than their current stream processing solutions. Once at Twitter, development of Storm accelerated and it was open sourced that same year, becoming a top level Apache project in 2014.

Storm formed the basis of all of Twitter's critical internal real-time compute functionality (Toshniwal et al., 2014), and because of its performance at Twitter was used by many other companies including Yahoo and Alibaba. At the time of DSPS selection in 2015 Storm was seen as the stream processing system of choice and was commonly referred to as the 'real-time Hadoop' in recognition of its wide adoption and big data processing abilities.

[1] See (Hesse & Lorenz, 2015) for a more detailed, comparison of the systems (as they were in 2015).
[2] https://storm.apache.org/releases/1.2.2/index.html
[3] https://github.com/apache/storm/tree/v1.2.2
[4] http://storm.apache.org/
[5] https://about.twitter.com/
[6] Details of this time and the early versions of Storm can be found on Nathan Marz's blog: http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html

Storm is not without its faults and has since been replaced as Twitter's real-time compute backbone by Heron[7] (Kulkarni et al., 2015). However, it has a stable codebase, is still under active development (its version 2.0 release happened in late 2019) and is still used in production at several large companies.

2.2.2 Overview

Apache Storm allows a developer to define a query (sequence of operations) which can be applied to a continuously arriving stream of data items, which are referred to as *tuples* within Storm. This query can be formed of two or more *components* that contain user defined code which receive tuples and create zero or more tuples in response. The developer defines the type of connection between components (see section 2.5) by specifying the streams that components subscribe to and output tuples onto. Components can subscribe and output to multiple streams and can place different output onto different streams. This allows developers to create components that can split and join streams.

The sequence of connected components forms a directed graph, which is referred to as a *topology* (see section 2.3.2). Much of the technical and academic literature will refer to streaming queries as directed acyclic graphs (DAGs). However, there is no functionality in any of the mainstream DSPSs to stop a developer creating a cycle within a streaming query. In fact, many of the message guarantee implementations in DSPSs form them by default. Therefore, topologies should only be referred to as directed graphs (DiGs).

Figure 1.1 shows an illustration of a topology; for example, this could be extracting tweets from a web API and counting the prevalence of words and hashtags (words preceded by special characters to indicate topics). The first component (S) pulls external messages into the topology and is a special type of Storm component called a spout (see section 2.3.2). This then emits the raw text of the tweet to component A which splits the text into individual strings (words and hashtags) which are sent to component B. This then sends words to component C and hashtags to component D on separate streams. Both these components maintain counts of their respective inputs and periodically update external databases.

The developer writes each of the topology's components as a Java[8] class and specifies how these components are connected via a builder class in the Storm API. Once the topology is defined and any additional configurations are set, the developer compiles the topology into a JAR[9] file which contains all the dependencies needed to run the user defined code. This JAR is then issued to the Storm Cluster via a command line client and the main control

[7] https://heron.incubator.apache.org/

[8] Java is the main language used with Storm, however APIs for Python and Clojure exist as well as the option to run generic binaries on the worker nodes.

[9] JAR files are Java's compressed archive format.

node (Nimbus) will then distribute the code across the cluster and begin processing. At any time the developer can use the command line client (or the web user interface) to stop, kill or change the topology's configuration (see section 2.11).

2.3 Storm Elements

2.3.1 Storm cluster

Several nodes, consisting of real or virtual machines, comprise an Apache Storm cluster.

Master/Nimbus

The Master node is the leader of the Storm cluster and is the control point to which jobs and commands are submitted. It runs a daemon called 'Nimbus' (the master node is often referred to as the Nimbus node) that is responsible for distributing code around the cluster, assigning tasks to machines and monitoring for failures.

Worker node

The machines which perform the actual computation on the streaming data are the worker nodes. These run a daemon called the *supervisor*. The supervisor listens for topology configurations (see section 2.6.2) assigned to its machine and starts and stops worker processes (the elements that run the processing logic for a topology — see section 2.4.3) as necessary. Each worker node has a set number of *slots* available to run worker processes. This number of slots is fixed when the supervisor daemon is started and can only be changed via a restart of the daemon.

Zookeeper

All coordination between Nimbus and the supervisors is done through a Zookeeper[10] cluster. Additionally, the Nimbus daemon and supervisor daemons are fail-fast and stateless; all state for these daemons are kept within the Zookeeper cluster so that nodes can be quickly restarted in the event of failure.

2.3.2 Topology

The DiG of operators that form a query over a set of streaming inputs is referred to as a *topology* in Storm. A topology can be represented in many different ways (section 2.6 describes these in more detail). However, at a high level, each topology is formed of several connected components (spouts and bolts) that contain user defined code which operates

[10] https://zookeeper.apache.org/

on the streams of incoming data. A user will define the processing logic in each component and how those components will route data to the components further along the topology (*downstream*). The principal elements of a Storm topology are described below:

Tuple streams

The core abstraction in Apache Storm is the *stream*. This is an unbounded sequence of data items that pass into and out of the topology's components. In Storm these data items are referred to as *tuples* and consist of key/value pairs (the keys are referred to as *fields* within Storm). Each tuple stream is configured with a *stream grouping* that dictates which downstream element will receive the outgoing tuples. Stream groupings are discussed in more detail in section 2.5.

Spouts

Spouts are the source of tuples for the topology. Spouts connect to data sources outside of the Storm system (element S in figure 1.1 is an example of a spout). Spouts can connect to message brokers such as Apache Kafka or Apache ActiveMQ, Web APIs such as Twitter or many other sources of input events. Spouts are also part of Storm's message guarantee system and are responsible for acknowledging the processing of messages and dealing with messages that have failed to be fully processed (see section 2.9 for more details).

Bolts

Bolts are the main components in a Storm topology. They can subscribe to multiple input streams produced by other bolts and spouts, and can emit tuples onto multiple output streams. When a bolt emits a tuple onto a stream, these are sent to all the bolts that are subscribed to that stream. For example, in figure 1.1, bolts C and D subscribe to bolt B. If C and D were subscribed to the same output stream from bolt B then, when bolt B emits a tuple, a copy is sent to both bolts C and D for them to process. However, if bolt B creates different content for bolts C and D, these separate tuple types can be submitted to different output streams which bolts C and D can subscribe to separately.

2.4 Parallelism in Storm

One of the most powerful features of Apache Storm is its ability to *scale* its topologies. Scaling in Storm refers to the increasing or decreasing of the number of *replicas* of a given *component* (spout or bolt). The number of replicas of a component at any given time is referred to as the *parallelism* of that component.

Storm runs a user-defined number of replicas of the topology's components in parallel

across a number of worker processes (see section 2.4.3), which themselves are running on the various worker nodes of the Storm cluster. Each worker process runs several executors, which are the replicas of each component (see section 2.4.1) and are the main processing units of a topology. Each executor is formed of several threads and queues. The executors each hold a number of tasks (see section 2.4.2) which do the actual tuple processing and aid in state partitioning. Figure 2.1 shows a diagram of the internal structure of a worker process and the elements are described in more detail in the sections below.

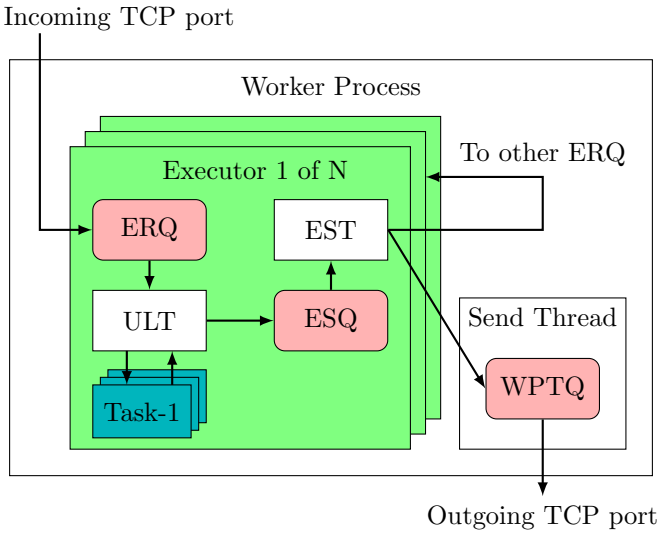

Figure 2.1: The internal structure of a worker process and executors.

2.4.1 Executors

Executors are the principal processing unit of a topology and its primary form of parallelism. They represent the processing elements of the topology's logical plan (see section 2.6.3). Multiple executors can run in each worker process and each executor is formed of two threads, the user logic thread (ULT) and the executor send thread (EST). Along with the processing threads each executor has queues to buffer the incoming tuples. Bolts have two internal queues, the executor receive queue (ERQ) and the executor send queue (ESQ), whilst spouts (which create tuples) have only the ESQ. These internal queues are described in more detail in section 2.7 and the processing of tuples within the executors is covered in more depth in section 2.8.1.

Each executor runs code for a single component. The number of executors for each component is configurable via the *parallelism* that is set when each spout or bolt is created via the topology configuration (see section 2.6.2). The executors are spread across the worker processes in the cluster. The number of executors can be changed while the topology is running to improve performance, see section 2.11 for more details.

2.4.2 Tasks

Each task is an instance of the user defined code for a given component and maintains any state for that code in memory. They are run within the ULT of each executor. An executor can hold multiple tasks but, as the tasks are run in the single ULT, only one task is running at any one time within the executor. Therefore, the number of executors assigned to a component will be less than or equal to the number of tasks assigned to it. The total number of tasks for each component is fixed for the lifetime of the topology and so any changes in the parallelism of a component (the number of executors assigned to a component) will mean that the tasks may be reassigned to different executors.

The motivation for using a separate entity to sub-divide the tuple processing within an executor, instead of running the same code instance on every tuple, is linked to the routing of tuples (see section 2.5) and the management of *state* within the executors.

For example, if we wish to perform a word count, we need to partition the stream of word tuples by a key value (*fields grouping* — see section 2.5), in this case the word itself. This is so that tuples containing the same word are always routed to the same executor, otherwise we would not have a unique, accurate count of each word's frequency. For a topology that never changes, the need for further sub-dividing the processing in each executor is not immediately obvious. We could just assign each executor a unique number and use that to route tuples of the same word to the same executor. However, if we change the parallelism of the topology's components, the state within each of the executors in the old arrangement will need to be redistributed between those in the new arrangement. Also, once the state has been redistributed, tuples need to be routed to the new executor containing the state for their particular key (field) value.

There is no general solution to this state repartitioning issue. The state could be strings (like in the case of a word count), numerical or custom data types. Knowing how to combine or divide this state and still route tuples to the correct executor, after a change in parallelism, is a difficult problem and in order to cover all options a DSPS would need to provide custom state partitioners for every topology.

Storm avoids this situation altogether by fixing the number of tasks for the lifetime of the topology. This approach automatically segregates the state space into a fixed number of partitions. The number of downstream tasks (for a fields grouped connection) does not change, regardless of the parallelism of the downstream component and so the value of the key (field) used for the tuple routing will always correspond to the same task identifier, regardless of which executor that task is hosted in. This situation is illustrated in figure 2.2.

When a component is scaled up or down, the tasks assigned to each of that component's

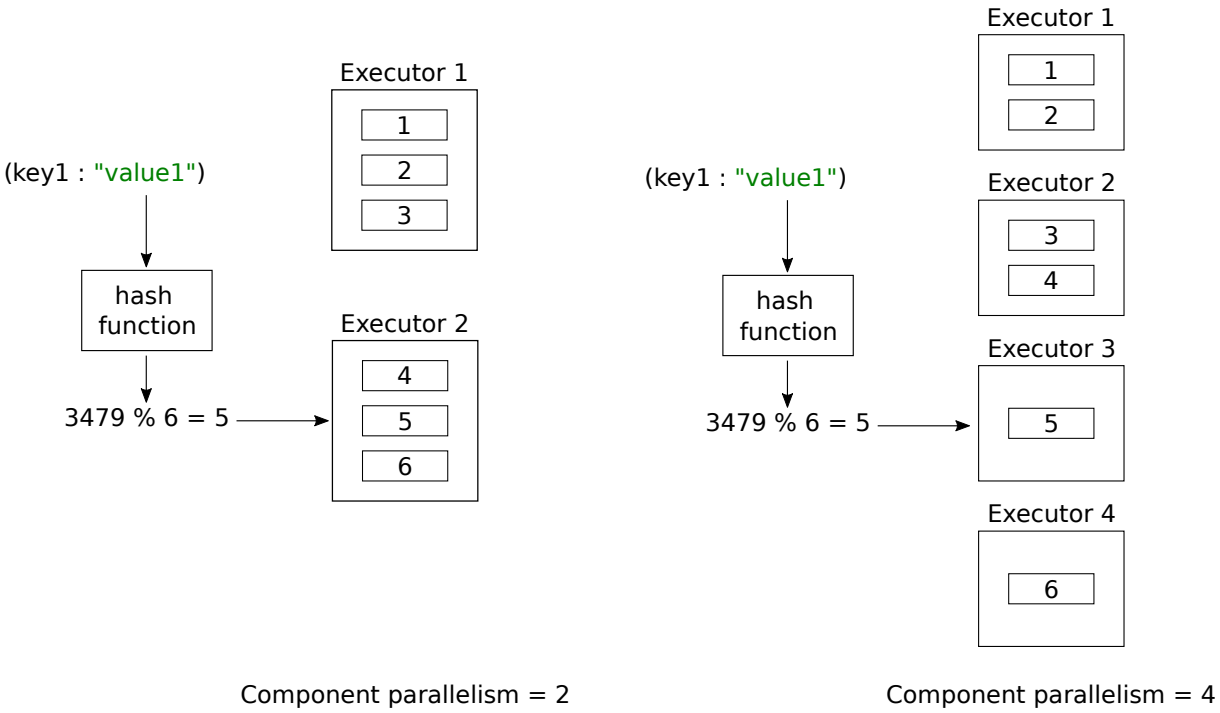

Figure 2.2: Illustration of how the fixed number of component tasks ensures deterministic routing when component parallelism changes.

executors may change. In order to restore their state, the new executors can simply retrieve the state for each of the tasks they have been assigned without needing special logic to recombine that state. This means the user does not need to code state partitioning into their component design.

The default setting is to have one task per executor. However, if you know that in future you may wish to scale up your topology, by having multiple replicas of each component, you can specify more tasks per component. The task per component value represents an upper bound on the scalability of a topology, beyond which you would have to take the topology down and restart it in order to increase the parallelism of a component.

2.4.3 Worker processes

Each worker node (see section 2.3.1) runs a number of worker processes. Each worker process is a separate operating system (OS) level process, in this case a Java Virtual Machine (JVM) instance, that runs operations from a single topology. Multiple worker processes from the same topology may run on the same worker node, as well as worker processes from completely separate topologies.

Each worker process is made up of several threads. The worker process maintains a thread connected to the network which transfers tuples directly from the network to the appropriate executor. There are then a number of executor threads running within the worker process which process the tuples. Finally the worker process send thread (WPST)

takes messages from the worker process transfer queue (WPTQ), which contains tuples emitted by the executors, and sends them on to their required destination. This destination may be a task on a different worker process on the same worker node or a task in a worker process on another worker node in the cluster. Tuples bound for executors in the same worker process are transferred directly to their destination by the executors themselves. This process is covered in more detail in section 2.8.2.

There are a set number of worker processes for each topology. These will be distributed across the available worker nodes within the Storm cluster. The number of worker processes for a topology can be altered while the topology is running to improve performance, see section 2.11 for more details.

2.5 Stream Groupings

As mentioned in section 2.3.2, the core abstraction of Storm is the stream. Topology components can subscribe to any stream emitted by any other component in the topology. The *stream grouping* refers to how the stream routes tuples from the tasks of one component to the tasks of another. There are several groupings available by default and these are summarised below:

Shuffle

When created this grouping will randomise the list of task IDs assigned to the downstream component. Then, every time this grouping is asked for a downstream task, it returns the next ID in this list, wrapping around to the first item when it reaches the end. As the task IDs assigned to a component are shared equally across all executors assigned to that component, this has the effect of load balancing the output tuples across all downstream executors.

Fields

A fields grouping will take one or more field (key) values from an outgoing tuple and use a hash function to get an integer value representing the values of the chosen field(s). The modulo of this hash value with the number of tasks in the downstream component is then taken. This yields an index which is used to extract a task ID from a list of all downstream tasks.

This method ensures that tuples with the same field(s) value(s) will always get routed to the same downstream task. This grouping is used for topologies where aggregation is required (such as a word count).

All grouping

This will send a copy of each emitted tuple to all the tasks of a downstream component. This is usually used for testing and is not recommended for production.

Direct grouping

This is a special kind of grouping which allows the emitting task to dictate which downstream task should receive the tuple. This can only be performed on streams that have specifically declared as direct streams and is most often used for messaging between the Storm system components.

Custom stream grouping

On top of the grouping described above a user can implement their own grouping (routing) strategy by extending the base grouping class.

2.6 Topology Plans

In discussion of Apache Storm topologies there are several distinct ways in which a topology can be represented, depending on the context. These are defined below and illustrated in figure 2.3.

2.6.1 Query plan

The query plan refers to the high level DiG of components (spouts and bolts — see section 2.3.2) that forms the Storm topology. This is the plan that a user would implement by defining the logic that will operate on the stream of tuples and how the components are linked together. The user will also set the number of tasks for each component when designing the query plan. Once set, these values are immutable and are shown in the boxes below the components in the query plan section of figure 2.3.

2.6.2 Configuration

The topology configuration refers to the parallelism (number of replicas — see section 2.4) of each component and the number of worker processes (see section 2.4.3) assigned to the topology. The topology configuration is the minimum information a scheduler requires in order to create a physical plan proposal and is set by the user during topology deployment and rebalancing (see section 2.11).

The topology configuration in figure 2.3 shows the parallelism of each of the components listed in the query plan above it, as well as the parallelism of the *Acker* component.

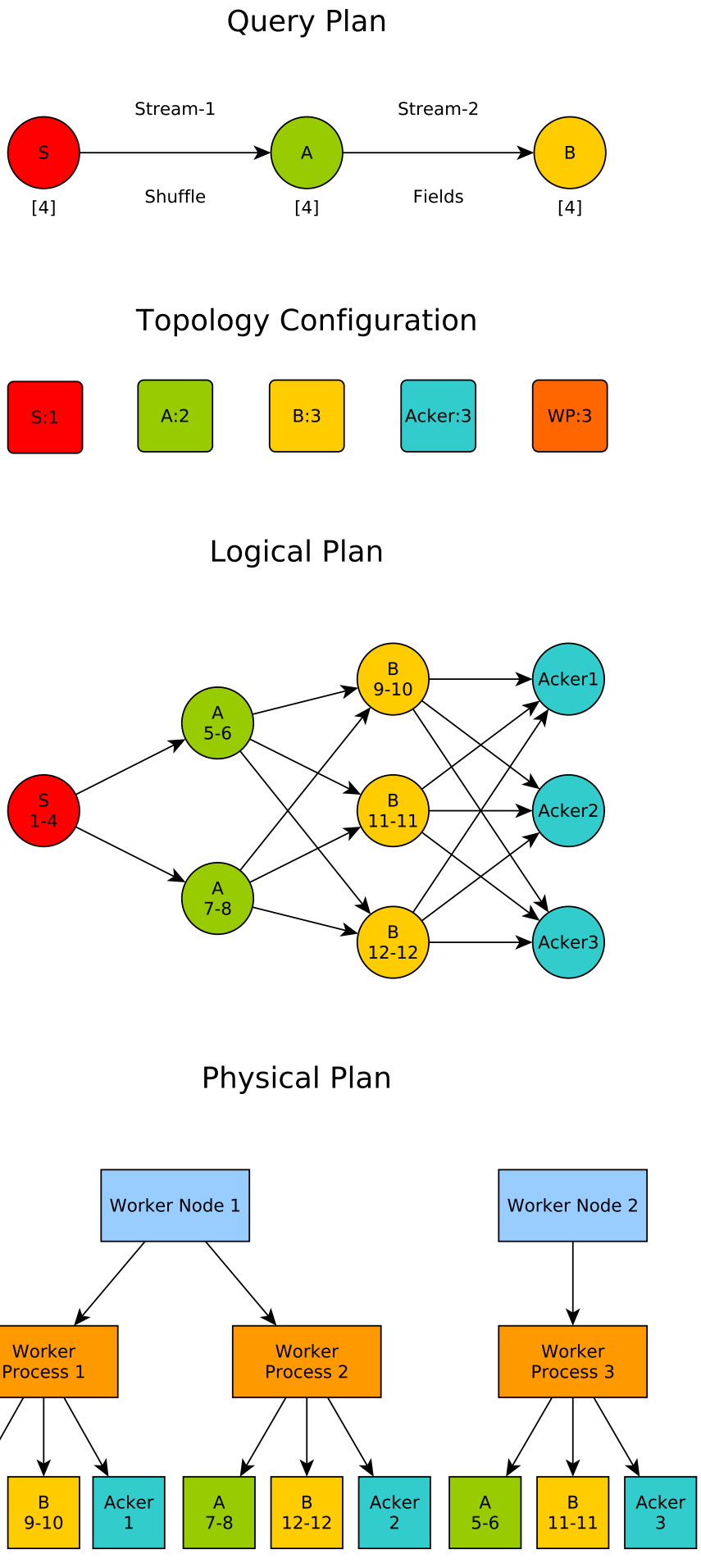

Figure 2.3: Example of the different plan types for a simple linear topology.

The Acker is a system component and is part of Storm's message guarantee system (see section 2.9). It is automatically added to every topology and its parallelism can be altered via the topology configuration like any other component. Finally, the topology configuration sets the number of worker processes assigned to the topology.

2.6.3 Logical plan

The logical plan is the DiG of individual processing units. In the case of Storm this will be the executors (see section 2.4.1) that run the replicas of the user defined tuple processing code. Depending on the topology configuration there could be multiple executor instances for each component. The number of tasks assigned to each component are divided equally between the executors assigned to that component.

The executors in the logical plan shown in figure 2.3 are labeled with their assigned task ranges. The task ID numbers are unique within a topology, tasks in different executors cannot have the same ID and therefore they are sequential across components. The Acker are also assigned task ID numbers, however these are labeled differently for illustration purposes.

2.6.4 Physical plan

The physical plan is the layout of elements of the logical plan on the physical elements of the Storm cluster (see section 2.3.1). Therefore the physical plan shows which tasks (see section 2.4.2) are within each executor, which executors are within which worker process and on which worker nodes those worker processes are located. It is the physical plan which the schedulers, described in section 2.10, produce when given a query plan and topology configuration.

The physical plan shown in figure 2.3 is a representation of the output Storm's default round-robin scheduler would produce if given the query plan and topology configuration shown at the top of the diagram.

2.7 Internal Queues

The queue implementation used in Apache Storm[11] is based on a message passing system developed by the LMAX Financial Exchange[12]. This system, called *Disruptor*, is designed to allow concurrent threads to input into and extract from a shared queue. Quoting from the LMAX white paper (Thompson et al., 2011):

[11]The descriptions in this section relate to Apache Storm version 1.2.2 and earlier. The queue implementation was changed in version 2.0.

[12]https://www.lmax.com/

> "At the heart of the disruptor mechanism sits a pre-allocated bounded data structure in the form of a ring-buffer. Data is added to the ring buffer through one or more producers and processed by one or more consumers."

Storm uses the Disruptor Ring Buffer to allow the executors to be able to extract tuples off the queue, at the same time as the worker processes are adding them without using locks or barriers which may slow processing time or cause race conditions.

In order to use the Disruptor effectively Storm wraps the Ring Buffer in some additional infrastructure. This adds the ability to deal with overflowing queues and uses batch insertion into the Disruptor to ameliorate the overhead of the insertion operation. These Disruptor queues are used for the ERQ and ESQ within the executors (see section 2.4.1) and the WPTQ within each worker process (see section 2.4.3).

2.7.1 Overview

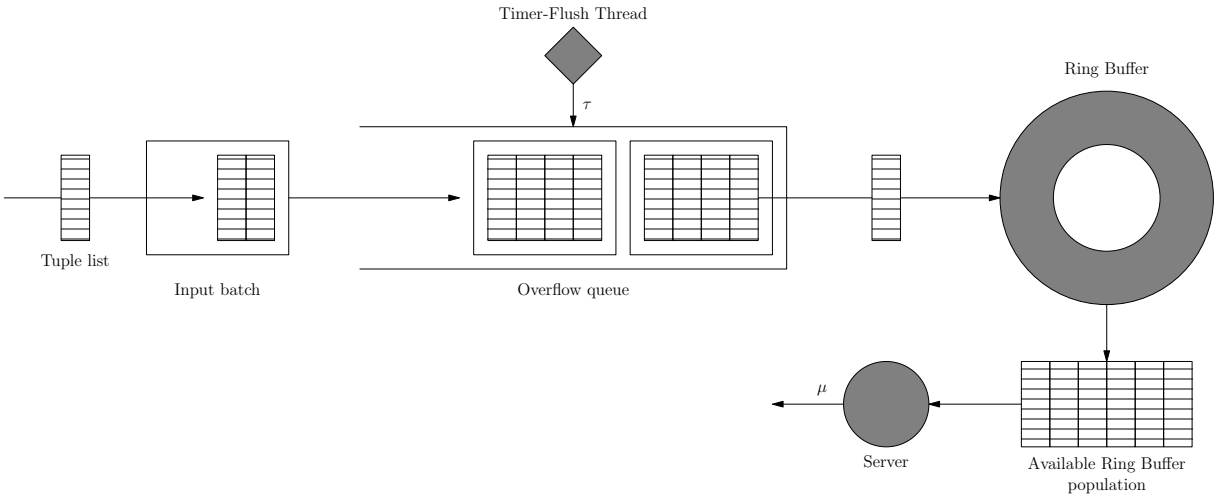

Figure 2.4: The Apache Storm Disruptor queue implementation.

The implementation of the Apache Storm Disruptor queue is illustrated in figure 2.4. It is formed of several elements which wrap the Disruptor Ring Buffer. There is an unbounded *overflow* queue, in front of the Ring Buffer, onto which batches of incoming items are placed. The items in the batches on the overflow queue are moved onto the Ring Buffer, which has a fixed size, when there is space for them.

Periodically, at a constant rate, a *timer-flush* signal is issued to the overflow queue. When a timer-flush is triggered, the current input batch (regardless of its size) is transferred to the overflow queue (or directly to the Ring Buffer if the overflow queue is empty and there is space). Then all batches in the overflow queue are pushed to the Ring Buffer. This timer-flush operation will linger, with subsequent flush signals being ignored, until the overflow queue is empty.

The slots in the Ring Buffer contain individual items from each inserted batch. A consumer

(such as an executor) asks the Ring Buffer for new items when that consumer is idle. The Ring Buffer will then return **all** available items to the consumer for processing.

The sections below go over the operation of the Disruptor queue in more detail and flow charts are provided for easy reference.

2.7.2 Queue input

In the following sections, the items arriving into the Disruptor queue are described as *objects*. This is because, depending on the context the Disruptor queue is used in, the input into the queue can vary. It may be an individual tuple (for the ESQ), a list of tuples (for the ERQ) or a *map* data structure which links identifier keys to a list of tuples (for the WPTQ). Further details on the input to these queues are given in section 2.8 and the second situation, for the ERQ, is illustrated in figure 2.4.

2.7.3 Arrival into the queue

Arriving objects are placed into a *batch* data structure (see *Input Batch* in figure 2.4). If the number of objects now in the input batch reaches a pre-set maximum size then the batch will either be placed directly onto the Ring Buffer or, if there is insufficient space for all objects in the batch, the batch (in its entirety) will be placed onto the overflow queue.

Every time an object is added to a batch and the batch is below the batch size limit, a non-blocking *arrival-flush* operation is called on the overflow queue. This involves looping through the batches in the overflow queue (from oldest to youngest) and attempting to add them to the Ring Buffer. If, at any point in this loop, there is not enough space for all the objects in the current batch in the Ring Buffer then the arrival-flush operation is cancelled. Figure 2.5 shows the arrival process in the form of a flow chart.

2.7.4 Timer flush interval completes

Periodically, at a constant rate, a *timer-flush* signal is sent to the overflow queue. This flush signal first causes the current input batch to be placed on the overflow queue, regardless of how many objects are in the batch. Then the batches in the overflow queue are looped over (from oldest to youngest) and their objects inserted into the Ring Buffer.

The difference between this process and the non-blocking arrival-flush (described in section 2.7.3) is that, if there is no space in the Ring Buffer for all the objects in the oldest batch, the system will wait for there to be space and then insert that batch. If an arrival happens during this wait period, the object will be added to a new input batch and the arrival-flush signal that this triggers will be ignored. If a new input batch fills up during

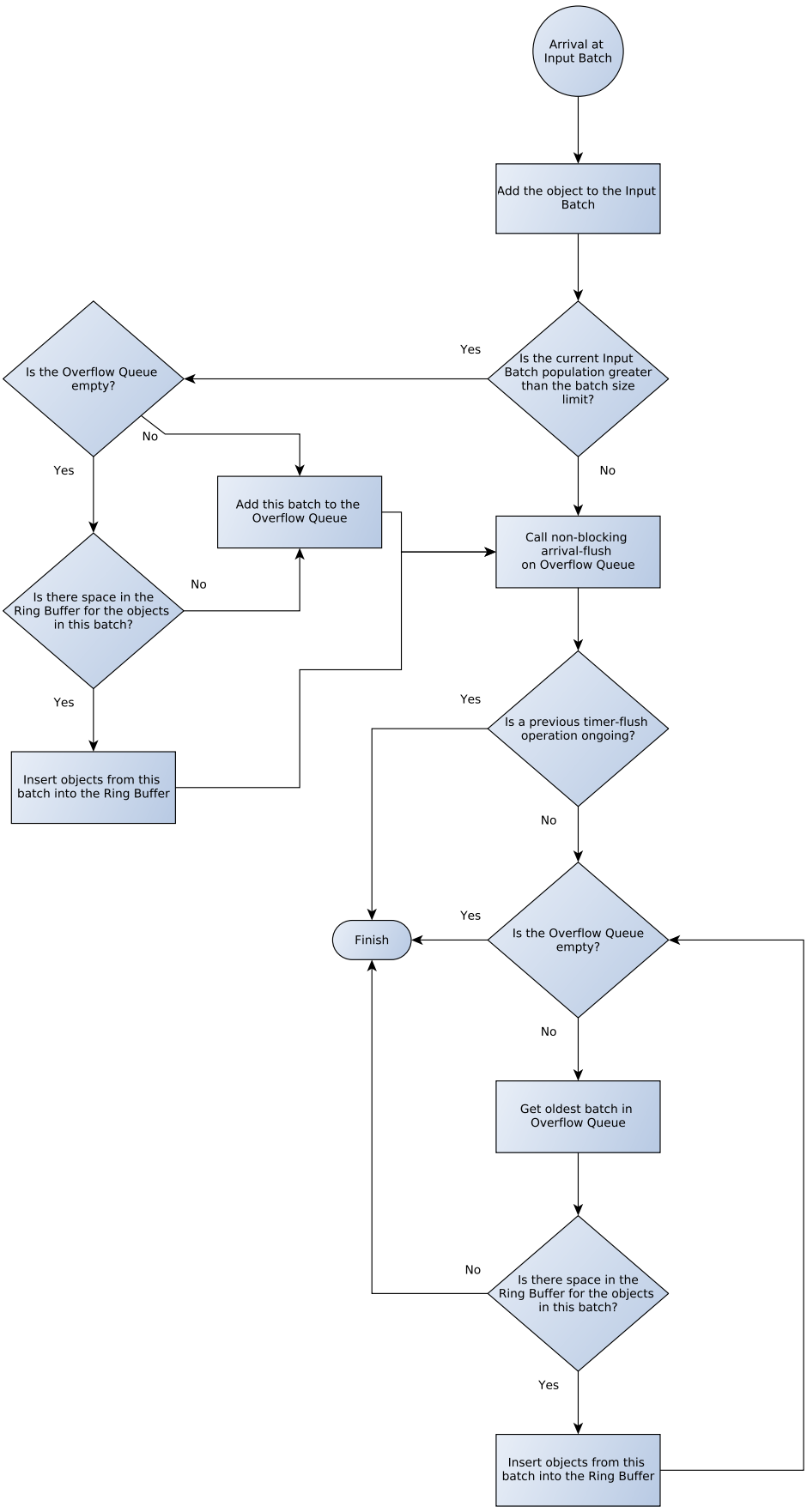

Figure 2.5: Flow chart showing the sequence of operations that occur when a job (either a tuple, list of tuples or map of tuples depending on the context) arrives at the queue.

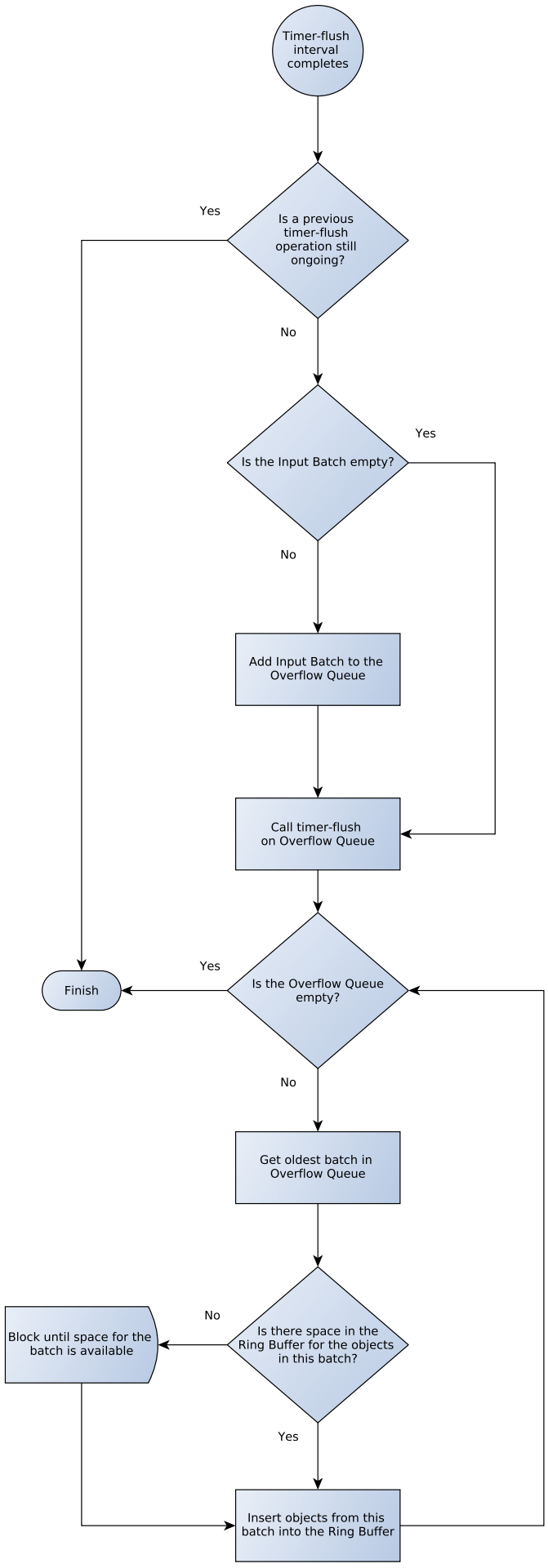

Figure 2.6: Flow chart showing the sequence of operations which occur when a flush interval completes.

the timer-flush operation, then that new batch will be added to the overflow queue and its objects will be added to the Ring Buffer as part of the timer-flush process.

If flushing all the batches in the overflow queue takes more time than the interval between timer-flush signals, then the subsequent timer-flush signal will be ignored. Figure 2.6 shows the timer-flush process in the form of a flow chart.

2.7.5 Service completes

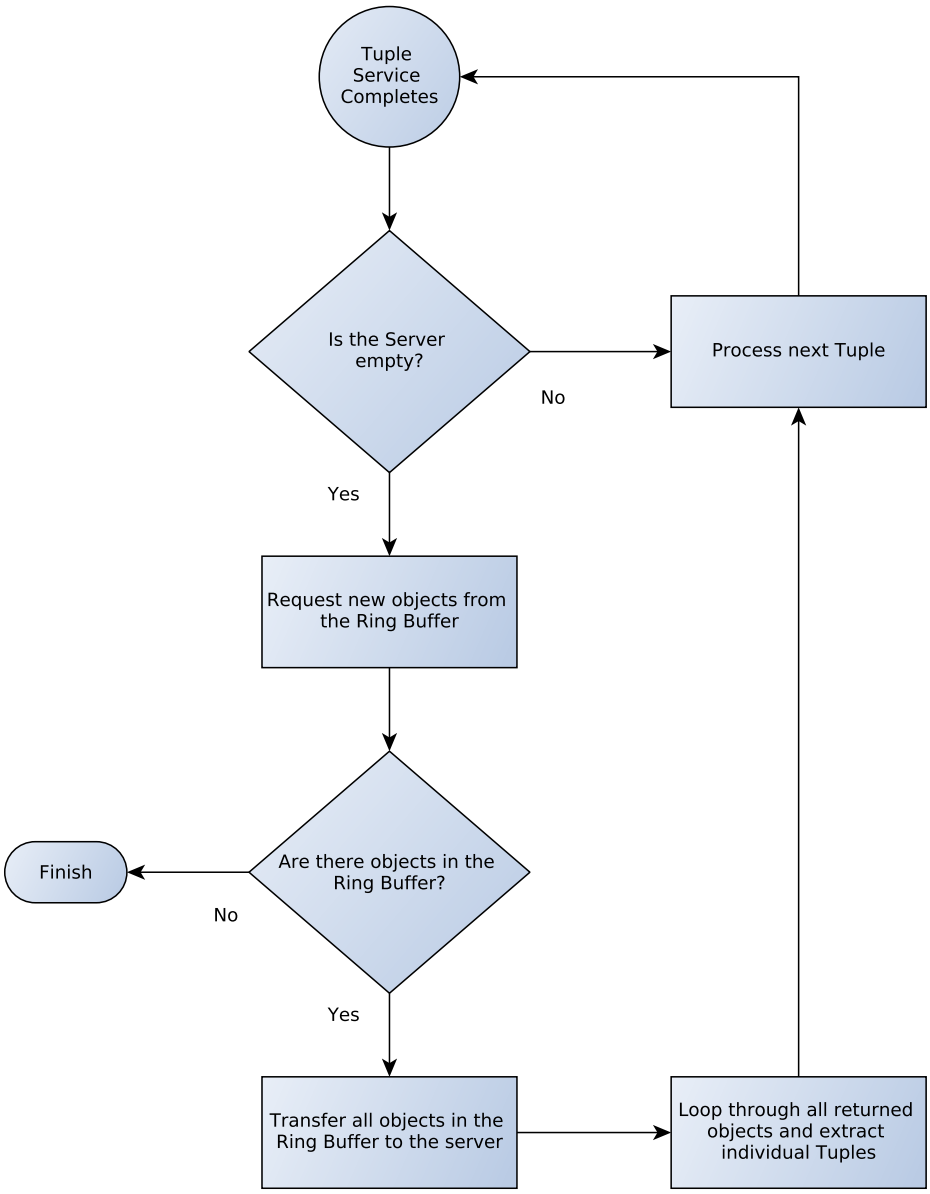

Figure 2.7: Flow chart showing the sequence of operations which occur when a tuple finishes service within the executor.

The servers which consume from the Ring Buffer vary in behaviour depending on the objects that are stored within the Ring Buffer. The server continually asks the Ring Buffer to provide objects. Any time a request is made the Ring Buffer will return **all** the objects

it contains to the server. The server will then process the objects sequentially until all finish service and will then ask the Ring Buffer again. Figure 2.7 shows the process which occurs when a tuple finishes service in an executor in the form of a flow chart.

2.8 Tuple Flow

This section gives a overview of the low-level tuple flow through an Apache Storm topology. Whilst section 2.7 detailed the operation of the queues within the executors and worker processes, this section details how those queues fit into the wider tuple flow.

2.8.1 Executor tuple flow

Figure 2.8 shows a schematic of the tuple flow through the elements of the executor. The inputs into the ERQ are lists of tuples which are addressed for the task instances housed within this particular executor. These are then the *objects* which pass through the Disruptor queue described in section 2.7.

The ULT, which is the thread controlling tuple processing in the executor, continually polls the ERQ Ring Buffer to see if there are tuple lists awaiting processing. Once tuple lists are present, the ULT will extract the entire population of the Ring Buffer into a list, which acts as a buffer for the bolt task (see *internal buffer* in figure 2.8). It will then iterate through this list (of tuple lists) and extract individual tuples which are then given to the task's `execute` method which contains the user defined code for each Bolt.

The bolt task will process the tuples in order and produce zero or more tuples as output. The time taken to process each tuple is recorded and reported to the Storm metrics system (see section 2.13). Once the internal buffer is empty, the ULT will then ask the Ring Buffer again for all available tuple lists, which were added whilst the internal buffer was being processed and will continue this cycle until the executor is terminated.

Tuples emitted by the bolt task are then submitted to the ESQ. For spout executors there is no ERQ and tuples are generated directly in the Spout task and passed to the ESQ. The ESQ is another Disruptor queue (as described in section 2.7), however in this case the *objects* passing through the queue are individual tuples. The EST will poll the ESQ Ring Buffer continually until tuples are available. When tuples are present on the Ring Buffer the entire population of the Ring Buffer will be extracted to a list (see *processing batch* in figure 2.8). This list of tuples is then sorted into tuples bound for tasks on this worker process (Local Dispatch List) and those bound for tasks on separate worker processes (Remote Dispatch Map).

The tuples in the Local Dispatch List are then given to the local transfer function (LTF) which will sort them into lists for each executor and pass those lists to the relevant ERQ.

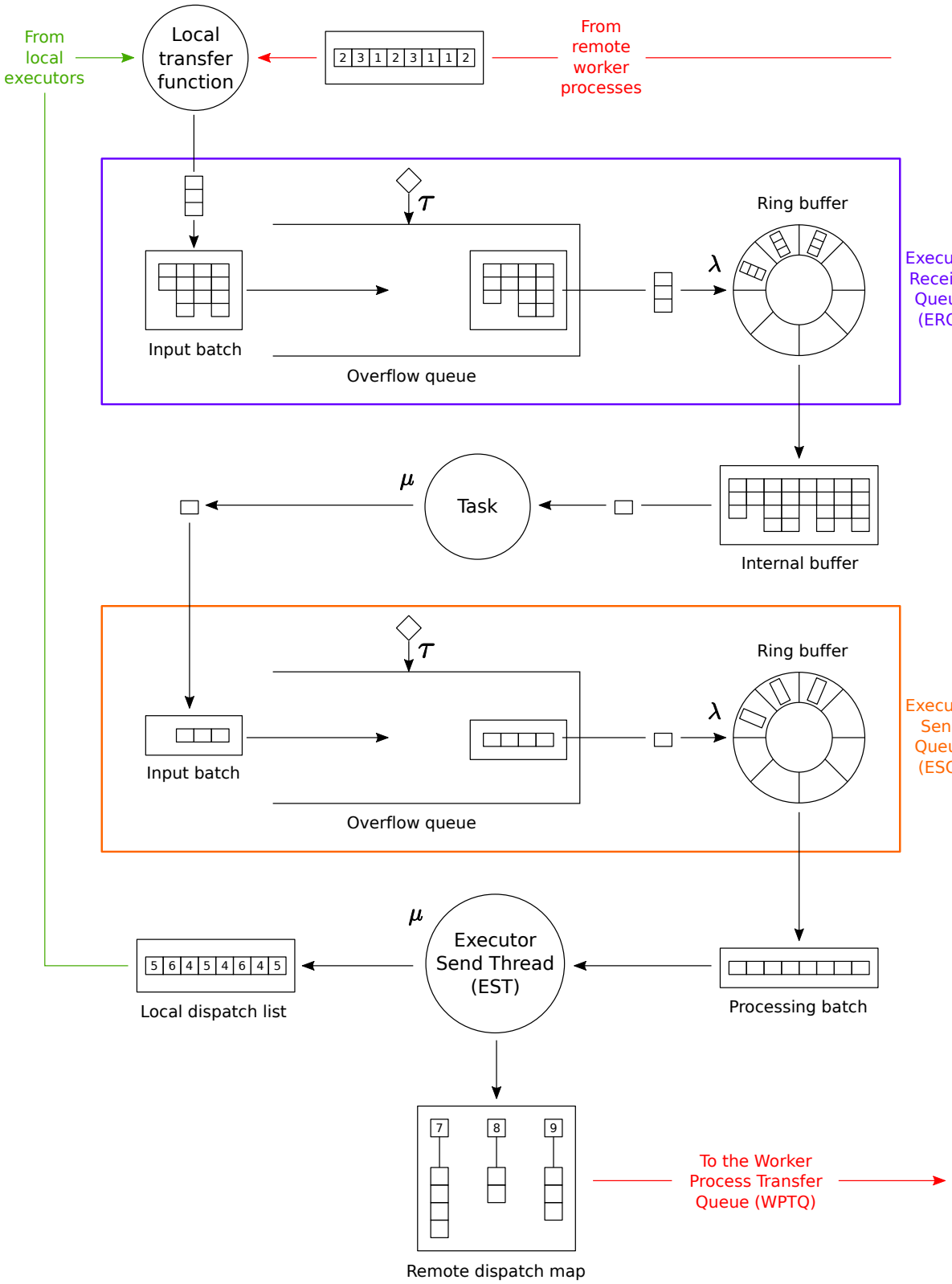

Figure 2.8: The tuple flow through the queues within each executor.

Tuples bound for tasks on separate worker processes are sorted into a map structure where the keys are task ID integers and the values are lists of tuples bound for that task. The Remote Dispatch Map is then submitted to the WPTQ.

2.8.2 Worker process tuple flow

Figure 2.9 shows a schematic of the tuple flow through the elements of the worker process and how tuples are sent across the network to a remote worker process. The primary elements of the worker process tuple flow are the WPTQ and the WPST.

The WPTQ is another Disruptor queue (as described in section 2.7), however in this case the *objects* passing through the queue are the Remote Dispatch Maps, created by the EST (see figure 2.8), which link task IDs to lists of tuples. The WPST will continually poll the WPTQ Ring Buffer until Remote Dispatch Maps are available. Once maps are present, the current population of the Ring Buffer will be transferred to a list (see *processing batch* in figure 2.9). The WPST will then take that list of maps and unify them into a single map so that each remote task ID links to a single list of outgoing tuples. This unified map is then converted into a map from the address of a worker process to a list of tuples bound for tasks on that worker process.

Each worker process list is then *serialised* (converted to a binary representation) and given to the network client for transfer to the remote worker process. The remote worker process receives the messages from other worker processes and *de-serialises* them back into lists of tuples. These lists are then given to the LTF which, just as with the Local Dispatch List, sorts the tuples into lists for each executor on the worker process and then issues these lists to the relevant ERQ.

2.9 Guaranteed Message Processing

There a several forms of message processing guarantee that are typically provided by DSPSs:

At most once This is the lowest form of message guarantee (if it can be called that at all) and means that a message (a tuple in the case of Storm) will be issued only once. The system does not keep track of sent messages and therefore, if the message fails to transfer or the receiver crashes before it is able to process it, the message is lost.

At least once This form of message guarantee means that any message that does not have a confirmed delivery will be resent from the source. This means that node failures and network issues could mean that a particular message is seen multiple times. The user of a system that provides this level of guarantee should implement procedures to handle repeat messages.

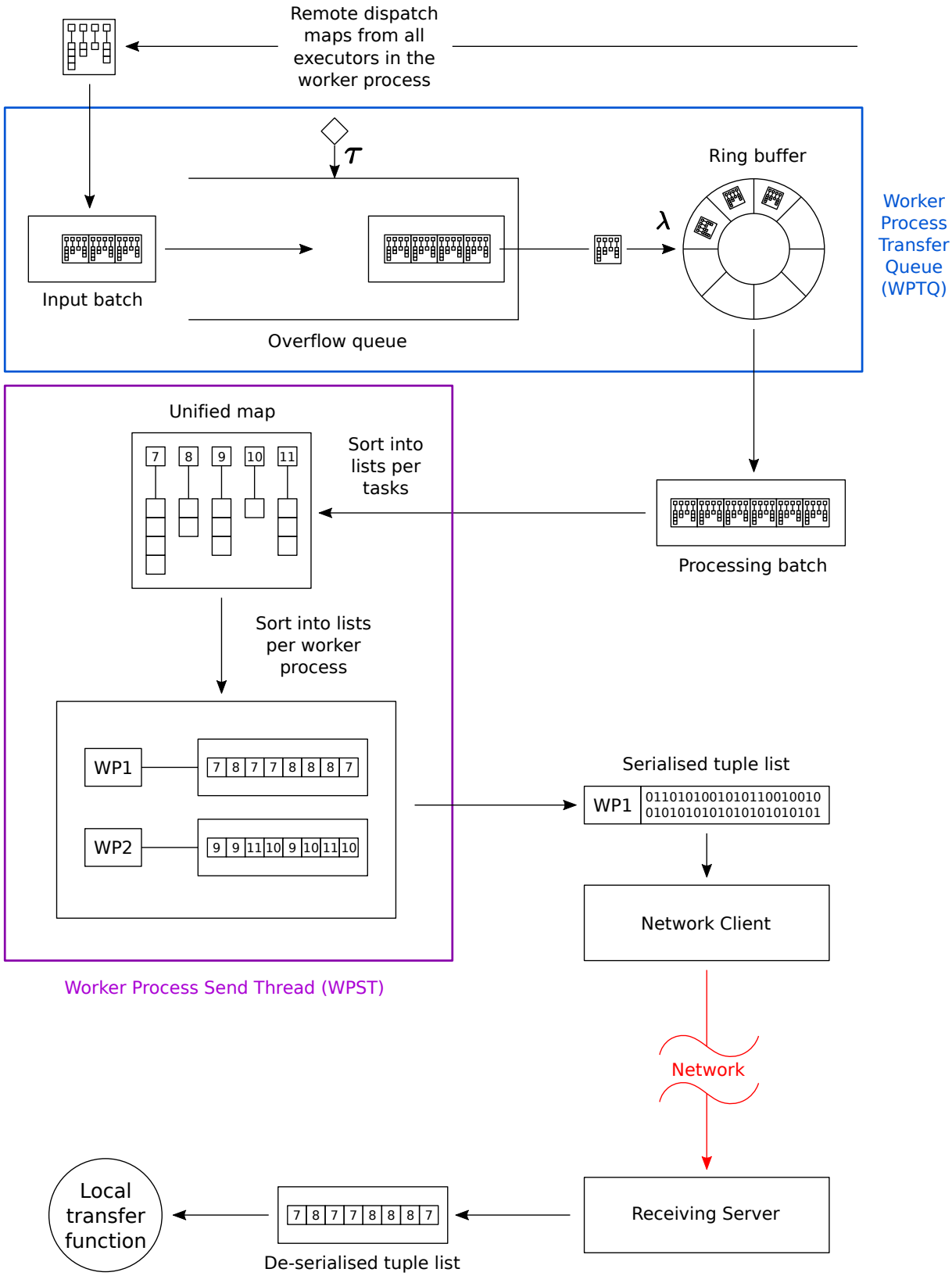

Figure 2.9: The tuple flow through the WPTQ, WPST and across the network to a receiving worker process.

Exactly once This form of message guarantee is the highest available and means that a message will definitely be sent and received once and once only. This means that the user of such a system does not have to create additional logic to handle repeat or missing messages.

Storm, by default, operates an *at most once* message delivery guarantee. However, it also provides the option for a topology designer to enable an *at least once* processing guarantee for every message emitted into the topology. A plugin for Storm called Trident[13] can provide additional message guarantee features including *exactly once* processing if required. The *at least once* processing system can be enabled for the whole topology or only for certain paths through the topology.

2.9.1 Acker

The acknowledger or *Acker* is a key component in Storm's message guarantee system. If guaranteed delivery is enabled for a topology then the Acker component is automatically added to each topology at deployment. The number of Ackers is configurable, but the default is to have one per WP (see section 2.4.3) unless very high levels of parallelism (see section 2.4) are required. A direct stream (see section 2.5) is created between every executor in the topology and the Ackers. Figure 2.10 illustrates how the Acker integrates into a topology by using a sample logical path from the example topology shown in figure 2.3.

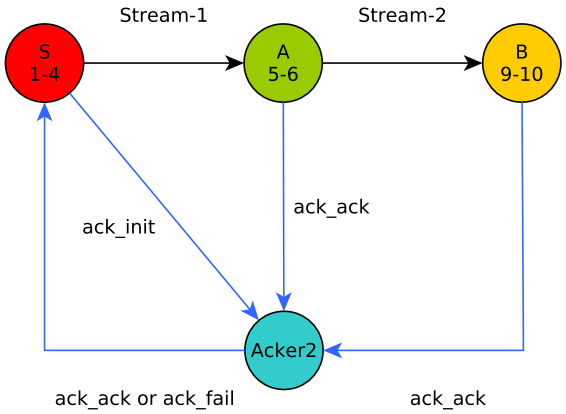

Figure 2.10: A simple linear topology, showing the Acker component and various streams associated with the message guarantee system.

2.9.2 Tuple tree

The message guarantee system works by sending acknowledgement (*ack*) tuples to the Acker to denote when the various stages of message processing have completed.

[13] http://storm.apache.org/releases/current/Trident-API-Overview.html

When a spout emits a tuple into the topology, an *ack_init* message is issued to the Acker via a direct stream. Each source tuple from the spout has a unique message identifier that is attached to the *ack_init* tuple. The Acker creates an entry for each source spout tuple within an internal map. This map links the source spout tuple message identifier to a 64-bit value (*ack_val*) which is initialized to zero and also to the identifier of the spout task that created the source tuple.

When a bolt issues a new tuple, it *anchors* this new tuple to the identifier of the tuple that created it (the input tuple provided to the bolt task's `execute` method). This transfers a list of all the source spout tuple identifiers, that the input tuple resulted from, into the new tuple and sends an *ack_ack* tuple from the bolt instance to the Acker. This *ack_ack* tuple contains the original source spout tuple identifiers and the 64-bit tuple identifier for the new tuple. The Acker looks up the relevant entries for the source spout tuple identifiers and XORs (adds) the new tuple's identifier with their stored *ack_vals*. This effectively adds the newly emitted tuple to the tuple trees which are rooted at the source spout tuples.

Once a bolt instance has finished processing an incoming tuple, i.e. all resulting child tuples of that input tuple have been emitted, the bolt instance will *ack* the original input tuple. This sends an *ack_ack* tuple (containing the source spout tuple identifier for the input tuple and the 64-bit identifier for that tuple) to the Acker. The Acker then looks up the source spout tuple identifiers and XORs (removes) the acknowledged input tuple's identifier from the stored *ack_vals* of those source spout tuples. This removes the acknowledged tuple from the tuple tree rooted at the source spout tuples.

Once the final component of a topology is reached (or further processing does not need to be guaranteed), the input tuple will be acknowledged but no new tuples will be created and therefore no new anchoring (tuple identifiers added to the *ack_val*) will occur. Once all the tuples that were produced from the original source spout tuple have reached an end component (remembering that multiple tuples could be produced from a single input tuple at each component in the topology), then the value stored for that source spout tuple's message identifier in the Acker should be zero (all the anchored tuple identifiers should have been XOR'd with a corresponding acknowledgement identifier). Once the Acker sees a source spout tuple's value has reached zero it sends an *ack_ack* tuple to the spout task instance that produced the message. The spout will then use the source tuple's message identifier in its `ack` method to perform any clean up operations with the external system.

If a source tuple's Acker entry does not reach zero after a certain, configurable, time (measured from the last received *ack_ack* for that source spout tuple message identifier) then the Acker will send an *ack_fail* tuple to the spout task instance that produced the original source tuple. The assumption is that if a tuple was emitted (and anchored) but not

acknowledged, that it either did not arrive at the downstream instance or the downstream instance has failed. At this point the spout calls its `fail` method and will replay the original source spout tuple and/or perform any other appropriate error recovery steps.

Storing a single value for each emitted source tuple is an effective way to limit the overhead of tracking large volumes of tuples through a topology and allows Storm to scale to hundreds of thousands of messages a second whilst still providing message delivery guarantees.

2.10 Topology Scheduling

Figure 2.3 shows an example Storm topology with three components, one spout (S) and two bolts (A and B). The topology configuration (see section 2.6.2) below the query plan shows the parallelism hint (the number of executors) set for each component. The topology configuration also shows the number of worker processes assigned to the topology. If we assume that the Storm cluster has two worker nodes then a possible physical plan produced from Storm's default scheduler (*EvenScheduler*) is shown at the bottom of figure 2.3. The *EvenScheduler* uses a round-robin system to assign the executors to the worker processes, and worker processes to worker nodes. This scheduler is not resource or workload aware and makes no assessment of where best to place a particular executor.

There have been several attempts to create more intelligent scheduling systems for Storm. Xu et al. (2014) created a scheduler based on reducing network traffic across the cluster of machines. Eskandari et al. (2016) use a graph partitioning approach to solve the same problem. Peng (2015) created a scheduler that took network and resource usage (CPU, Memory) into account when placing executors onto cluster machines (see chapter 3 for more examples). However, their system requires the user to provide all the expected performance statistics for each of the components *a priori*.

The resource aware scheduler[14], created by Peng (2015), is the only scheduler from the literature currently available in the core Apache Storm distribution (version 1.0 and above).

2.11 Rebalancing

In order to give topologies the ability to react to changing workloads, Storm provides a *rebalance* function. This function takes a topology identifier and then allows the topology configuration (see section 2.6.2) to be altered. Storm uses a *stop the world* rebalancing approach, where all processing is stopped whilst a new physical plan is calculated by the scheduler. Then new instances of all the executors for a topology are created according to

[14] https://storm.apache.org/releases/current/Resource_Aware_Scheduler_overview.html

the new physical plan. Any state stored by the tasks of a component can be recovered from external storage systems using the task ID as an index.

2.11.1 Worker processes

At any time while the topology is running, additional worker processes can be added via the rebalance command. These will be distributed across the cluster and the current executors redistributed among them. Obviously, if the number of worker nodes has not increased then this will result in more worker processes competing for the same cluster resources.

2.11.2 Executors

The number of executors for each component can be increased via the rebalance command and these will be redistributed among the available worker processes or across the new number of worker processes if that is specified in the call to the rebalance command. However, as mentioned above, if the cluster worker nodes are already heavily loaded then adding additional executors (without adding additional worker processes and worker nodes) may not result in significant performance improvements. This action may even reduce overall performance due to the resulting resource contention.

2.11.3 Tasks

As mentioned in section 2.4.2, the number of tasks per component is fixed for the lifetime of the topology and cannot be changed without stopping the topology and restarting it with a new number of tasks assigned to each component.

2.12 Windowing

A common operation in stream processing is windowing. Often processing every single input tuple individually is undesirable and/or inefficient. For example, computing a complex operation on every single incoming measurement is less efficient than computing it every minute with the average of all measurements that arrived in the last minute (presuming a one minute latency is acceptable). Window operations can take several forms, however they primarily fall into two categories:

2.12.1 Tumbling windows

Tumbling windows are defined as distinct, sequential blocks of time or data items. Once a given duration or count has been reached the current window is completed and the next

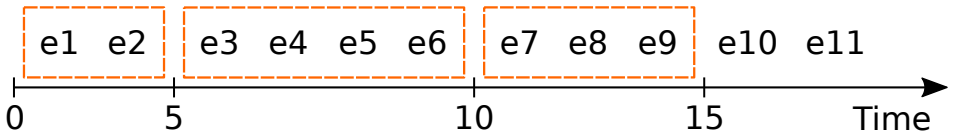

Figure 2.11: Example of a tumbling window.

window block is started. For example, a tumbling window with a period of 5 seconds is shown in figure 2.11.

2.12.2 Sliding windows

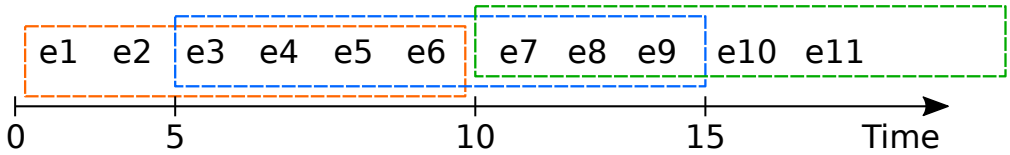

Figure 2.12: Example of a sliding window.

Sliding windows are of a defined length, however the bounds of that fixed length move forward by a fixed slide interval or count. For example, a sliding window with a length of 10 seconds and a slide interval of 5 seconds is shown in figure 2.12. A tumbling window can be thought of as a sliding window where the slide interval is equal to the window length. The main differences between sliding and tumbling windows is the higher completion frequency and that events can appear in multiple windows.

2.12.3 Windowing in Apache Storm

Storm provides native support[15] for windowing operations via an interface that is added to the bolt implementations. This alters the bolt's `execute` method to receive a collection of tuples representing a window. Storm provides functionality for both time and count based window length and slide interval; it also allows mixing the two, i.e. having a count based window length with a time based slide interval (which is useful if a low arrival rate means the count length is never reached) and visa versa.

Internally a special type of executor implementation is used for windowed bolts that uses an additional internal queue to store incoming tuples once they are processed off the ERQ. This windowed executor is provided with a trigger policy which dictates when a window is complete according to the window length and slide interval parameters supplied by the user. Once a window is triggered, the corresponding tuples are extracted from the window queue and given to the windowed bolt `execute` method. There is a corresponding expiration policy that will trigger when a tuple is not a member of any pending window (in the sliding case) and will then remove these expired tuples from the window queue.

[15] http://storm.apache.org/releases/1.2.2/Windowing.html

Time based windowing in Storm can use either *processing time*, which is the time the tuple was received into the executor (using the clock on the worker node), or *event time*, which is a time stamp provided by the source of the original message entering the topology. For example, this could be the time a measurement was taken at a remote sensor or the time stamp on a log entry. The user can supply a field name to the windowed bolt to designate the appropriate event time tuple field. Using event time instead of processing time introduces several challenges around dealing with tuples that arrive out of order or late, which in networked environments can happen often. To deal with these situations Storm employs watermarks and other approaches common to DSPSs like Apache Flink and Spark. However, a full description of the operation of these event time methods is not relevant to the performance modelling effort and therefore the reader is referred to the Storm documentation and in-depth texts such as Kleppmann (2017) and Lax et al. (2018) for more details.

The windowing functionality in Storm currently provides an *at least once* delivery guarantee (see section 2.9). The values emitted from the windowed bolt's `execute` method are automatically anchored to all the tuples in the input window collection. The downstream bolts are expected to acknowledge the received tuple (i.e the tuple emitted from the windowed bolt) to complete the tuple tree. If not, the tuples will be replayed and the windowing computation will be re-evaluated. The tuples in the window are automatically acknowledged when they fall out of the effective window period, i.e. after window length + sliding interval has passed.

2.13 Storm Metrics

Storm provides a wide variety of metrics on the performance of Topology elements.

2.13.1 Accessing metrics

There are two ways to access Storm's topology metrics. The Nimbus node's API provides summary metrics with sliding windows of 1, 3 and 10 hours as well as an *All Time* metric aggregated over the lifetime of the topology. These *Nimbus metrics* are easy to access via the API (REST[16] or Thrift[17] based), however they are summarised on a per executor basis and only over the predefined windows described above.

The second method of obtaining topology metrics is to create a custom *Metrics Consumer* instance. This class is then registered as a consumer with Storm when the topology is created, and the metrics consumer instance is added as a bolt into the topology with a direct stream (see section 2.5) from every other component.

[16] https://storm.apache.org/releases/1.2.2/STORM-UI-REST-API.html
[17] http://thrift.apache.org/

The advantage of creating a custom metrics consumer is that metrics can be accessed with a more fine grained aggregation period. In Storm, metrics are aggregated over a configurable number of seconds (referred to as the *metric bucket period*), count metrics are summed and temporal metrics (such as latency measures) are averaged over this period and the resulting value reported to the registered metrics consumers and the Nimbus Node. For the Nimbus Metrics these values are summed/averaged again within the sliding windows. However, custom metrics consumers have access to the individual metric buckets. A further advantage to using a custom consumer is that many of the metrics are reported at the task level, compared to the Nimbus Metrics which are summarised at the executor level (averaged over all tasks within each executor).

2.13.2 Component metrics

The default Storm performance metrics, provided for the component (spout and bolt) executors in the topology, are described below. Some of these metrics only apply to specific component types, which preface the metric name.

Bolts: Process latency

This metric is defined as the time (in milliseconds) between the task starting the function that processes an incoming tuple (the `execute` method) and the calling of the *ack* (acknowledge) or *fail* functions to indicate that the tuple processing is complete or has failed, respectively.

Bolts: Execute latency

This is similar to process latency and is defined as the total time that the task's `execute` method runs for. This includes any additional processing that is performed after a tuple is emitted and acknowledged (or failed) by the task, such as database connection clean up.

Spouts: Complete latency

This is Storm's measure of the end-to-end latency of a topology and is part of its guaranteed message processing system (see section 2.9). Once all tuples that result from a source tuple have been processed, the Acker will register the completion and issue an *ack_ack* tuple to the spout task that produced the source tuple.

Previous to Storm version 1.0.3, the spout would receive the *ack_ack*, record a local time stamp, compare this to the local time stamp it recorded when the source spout tuple was originally emitted, and report the difference as the complete latency. This system ensured that timestamps would always be compared on the same worker node they were created on, thus avoiding clock synchronisation issues. However, this also meant that the

complete latency included transfer times from the final bolt component to the Acker, the processing time at the Acker, transfer time from the Acker to the spout and also queueing and processing time at the spout. This final element proves to be an issue as, due to the design of the spout components in Storm, incoming *ack_ack* messages and outgoing tuples cannot be processed concurrently. Consequently, for highly active spouts which are busy producing large volumes of source tuples, *ack_ack* tuples were queued for significant amounts of time. As the complete latency clock was not stopped until these tuples were processed by the spout, this led to artificially long complete latency values.

To address this issue, Storm versions 1.0.3 and above start and stop the complete latency clock in the Acker. When a source tuple's *ack_init* message, sent by the spout that created it, is first received by the Acker it adds the current time to its internal map along with the source tuple's message identifier. Therefore, when a source tuple completion is detected the current time can be compared against the stored start time and the delta between them reported as the complete latency. This avoids the need to consider clock synchronisation as the time comparison will always occur on the same Acker and therefore the same worker node.

This approach means that queues at the spout are avoided and some of the travel times (from Acker to spout) are excluded. However, this change of clock start/stop point has consequences for the interpretation of what the complete latency represents. As the clock is not started until the Acker receives the *ack_init* message, the measured complete latency will be shortened by the time it takes the message to travel from the source spout to the Acker, queue at the Acker and then be served. Conversely, as the clock is not stopped until the *ack_ack* from the final child tuple is received at the Acker, the complete latency will be lengthened by the time taken to travel from the final bolt executor to the Acker, queue at the Acker and then be served. Figure 2.13 illustrates the difference between the true topology end-to-end latency (top line) and the measured Storm complete latency (bottom line). If the *init delay* and the *ack delay* were of equivalent length then the complete latency would match the topology end-to-end latency. However, depending on the physical plan for the topology, there could be a large difference between the two delays. For example, the spout and Acker could be in the same worker process and the final bolt could be on another worker node entirely. This would lead to very short transfer times for the *ack_init* and relatively longer transfer times for the final *ack_ack*. This situation would result in the measured complete latency for that source tuple being longer than its true end-to-end latency.

It is also worth noting that the complete latency is not a measure of the end-to-end latency of each tuple, produced from a source tuple, it is actually a measure of the **worst case** end-to-end latency for a source tuple produced from a given spout instance. As the time delta is only calculated when the *ack_val* of a source tuple reaches zero, the complete

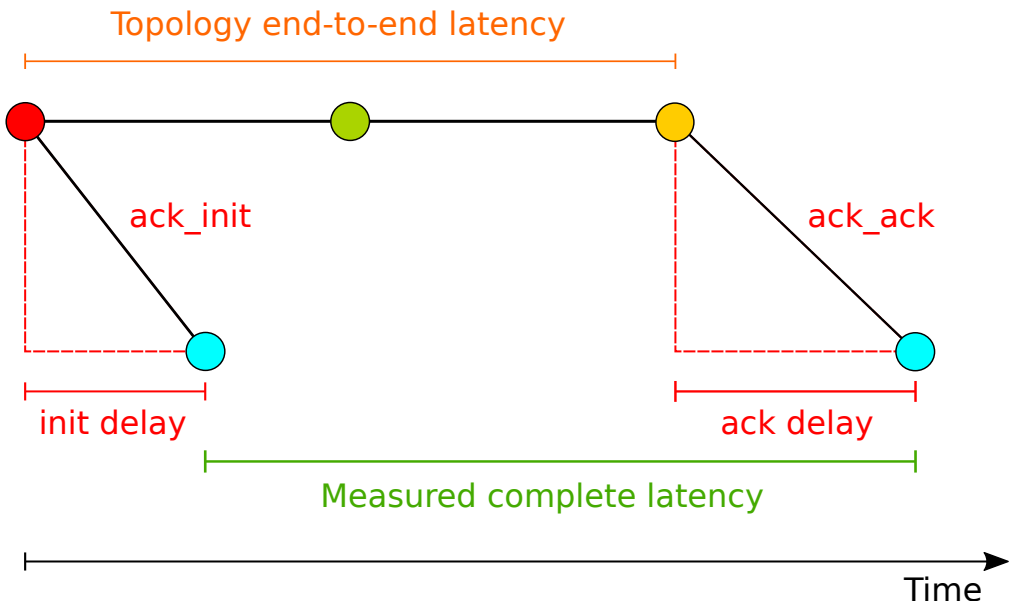

Figure 2.13: The topology end-to-end latency shown against the measured complete latency in Storm (version 1.0.3 and above).

latency is (taking into account the discussion above, illustrated in figure 2.13) the time between source tuple emission and the completion of the last child tuple from all tuples descending from that source tuple. Therefore, if a particular child tuple is routed across a costly remote connection to a final bolt instance and/or has to wait for a significant amount of time at that instance, and all other child tuples from that source take a much shorter route, then the complete latency will be significantly extended compared to the prevailing end-to-end latency of the other tuples from that source. Figure 2.14 illustrates the effect of delayed child tuples on the measured complete latency and shows how the complete latency measure is susceptible to stragglers and represents the *longest path* through the topology.

It is also important to bear in mind that the complete latency measure is averaged across the metric bucket period and also across all possible paths through the topology logical plan from a given spout instance. Therefore, if there are significant differences in the processing time between different paths, the complete latency will average across these. This makes the complete latency susceptible to positive skew as outliers can "pollute" the average over the metric bucket period.

Spouts and bolts: Throughput

The executor throughput metrics come in three forms:

Emitted The count of the number of times the emit method (of the object which transfers tuple out of executors) is called by the task. This will include tuples not emitted to another component (i.e. those leaving the topology), as well as those that fail to transfer

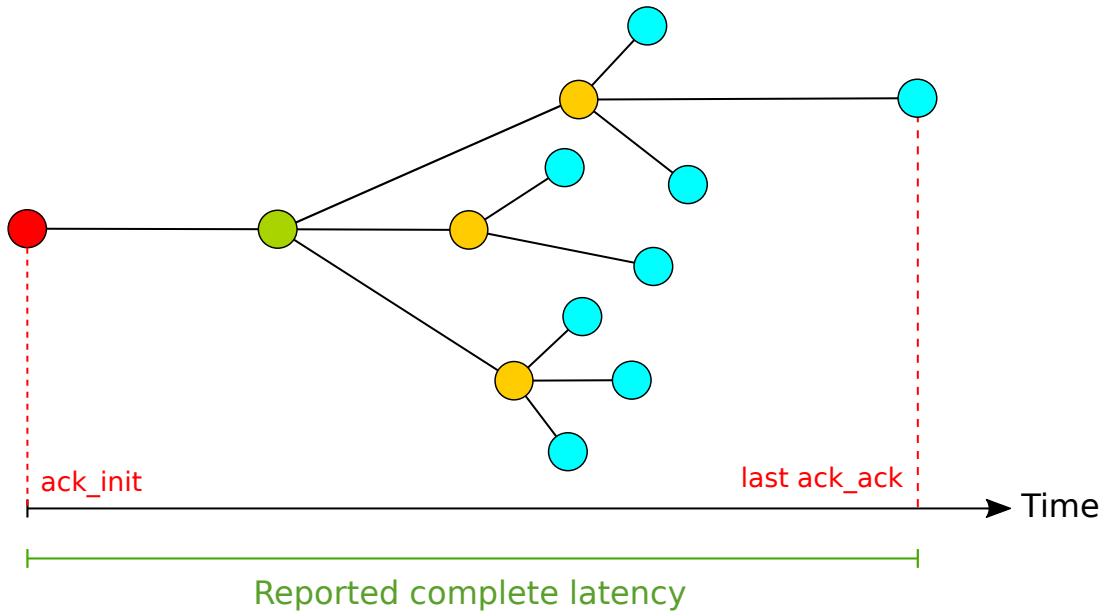

Figure 2.14: The tree produced as tuples are anchored and acknowleged, showing the effect of delayed child tuples on the complete latency.

correctly to other components.

Executed This is a count of the number of times a task's execute method is called. This count will include all attempts to process a tuple, including those that fail to emit and those that fail to transfer successfully to other components.

Transferred The count of the number of tuples that successfully move from this task to another within the topology (are acknowledged).

2.13.3 Queue metrics

Metrics for the ERQ and ESQ of each executor and the WPTQ of each worker process (see section 2.7) are recorded by default. Unlike the metrics in section 2.13.2 above, these are not recorded for each individual task but are linked to each executor. Metrics are reported for each queue every metric bucket period.

Arrival rate

This is the average number of individual tuples arriving into each of the queues per second. It is important to understand where this measure is recorded. The counts of tuple arrivals are not made at the input batch (see figure 2.4), they are made as objects are placed into the slots of the Disruptor Ring Buffer. Storm will inspect each object placed onto the Ring Buffer — which could be an individual tuple (for the ESQ), a list of tuples (for the ERQ) or a map from task identifier to a list of tuples (for the WPTQ) — and ascertains

how many individual tuples that object contains. This is added to a total for the queue and then an average calculated for the configured metric bucket period.

Prior to Apache Storm version 1.1.1, the system contained a bug[18] where it would count one object, being placed into a Ring Buffer slot, as one tuple. Under high traffic loads this meant that the arrival rate for a given queue would be significantly under reported by Storm, as the input tuple lists into the ERQ would contain many individual tuples which were not logged in the arrival rate metrics. The author is very grateful to Tang Kailin and the other Storm developers for tracking down and solving this particular issue.

Population

The Disruptor queue reports several population values:

- Tuple population: This is the number of individual tuples waiting within the Ring Buffer.
- Object population: This is the number of occupied slots within the Ring Buffer.
- Overflow population: This is the number of objects currently within the batches in the overflow queue, this includes the input batch (see figure 2.4).

These values represent a snapshot of the various populations: they are sampled once each metric bucket period and are not averages over that timespan.

2.13.4 Custom metrics

As well as the metrics defined above, Storm allows users to define their own custom metrics which can be reported to the main Storm system for display in the UI and captured by a custom metrics consumer in the same way as the default metrics.

Once a custom metric is created it is up to the user to write logic to update this metric in the topology components as Storm does not offer a way to do this automatically.

2.13.5 Metrics sample rates

In order to reduce the processing load of calculating the system metrics Storm samples the trigger events for the metrics at a user definable percentage. By default only 5% of tuple are used to calculate the metrics. For example, if the default 5% sampling rate is used, the system will randomly select 1 tuple out of the next 20 to increase the metrics by 20. The implication of this is that reported metrics may not be a true representation of the system state. The sample tuple could be logged and then processing stopped in the executor, yet an additional 19 tuples were logged with the metrics system.

[18] see https://issues.apache.org/jira/browse/STORM-2557

To gain an accurate picture of the topology's performance, the sampling rate can be set to 100%. However, this means that every tuple will trigger the metrics update logic and will increase the load on the executors.

2.14 Summary

This chapter has covered the internal operations of the Apache Storm DSPS. The key elements of which are:

- How the topology components are parallelised (section 2.4) and how the component's tasks (section 2.4.2) facilitate the partitioning of state.
- How the types of the connections between components can effect tuple flow (section 2.5).
- The queue implementation that Storm uses (section 2.7) and how, with the Disruptor queue's batching and complex queue flushing behaviour, this is far from a simple queueing system.
- How the flow of tuples, as they pass through the executors (section 2.8.1) and worker processes (section 2.8.2) and the additional routing and batching behaviour can effect tuple flow.
- Storm's message guarantee system (section 2.9) and how this effects the interpretation of Storm's measure of end-to-end latency: complete latency (section 2.13.2).
- How batching tuples into windows within components can takes several different forms (section 2.12).
- The various metrics for service time, arrival rate and throughput that Storm provides (section 2.13).

These key behaviours and their associated complexities are worth bearing in mind during the discussion of previous performance modelling approaches for DSPSs given in the following chapter.

Chapter 3

Related Work

This chapter seeks to provide context for the contributions presented in this thesis by examining both the relevant background literature and more up-to-date related works. It starts with an analysis of the literature on auto-scaling of distributed stream processing systems (DSPSs), beginning with systems based on reactive thresholds and then moving on to those utilising application performance models including queueing theory and machine learning based approaches.

To the best of our knowledge the mainstream[1] DSPSs have limited or no implemented mechanisms to automatically scale their operations. However, there has been significant research into the problem of optimal scheduling for DSPSs.

3.1 Threshold Based Auto-scaling

Heinze et al. have done significant research in the area of elastic scaling techniques for stream processing systems. Using their prototype complex event processing (CEP) engine, FUGU (Heinze et al., 2013), they have developed placement algorithms for the streaming operators (the processing units of the DSPS, like the executors in Apache Storm). They also developed a scaling decision system that can optimise their engine with regard to a service level agreement (SLA) based on sojourn time (Heinze et al., 2015). They have incorporated auto-scaling systems based on thresholds and have added the ability to tune the parameters of these thresholds based on previous scaling decisions, a form of reinforcement-learning discussed later in this chapter (Heinze et al., 2015). The FUGU system, while it does take into account performance SLAs and provides elastic scaling features, does not focus on anticipating increases in incoming workload or attempt to produce a performance model of the streaming system to aid in scaling decisions (Heinze, Pappalardo, et al., 2014). Their system simply changes the topology configuration until

[1] Apache Storm, Spark and Flink

the SLA is met.

With respect to the mainstream DSPSs, Apache Storm (Toshniwal et al., 2014) is well represented in the literature. Aniello et al. (2013) proposed a two-stage optimised version of Storm's default round-robin scheduler. Their first stage works off-line at the time of initial deployment, dividing the graph of streaming operations up into stages (groups) of operations that are strongly linked. These stages are then deployed on the same, or proximate, worker nodes to minimise network latency. It should be noted that the staging system by Aniello et al. is only applicable to acyclic work flows (this would exclude many machine learning algorithms or systems that incorporate feedback) and has no prior knowledge of the actual network performance of the operators. The second stage of their optimiser works on-line as Storm is running, monitoring central processing unit (CPU) load on the worker nodes and network traffic between operators (requiring the use of specifically modified classes within the streaming work flow). Their system then periodically runs an optimisation algorithm, based on the latest metrics, to organise the operators.

Xu et al. (2014) also developed an optimised scheduler. Their approach minimises network distance between connected operators which exchange high traffic or large payloads. Their system is also aware of worker node resource usage and employs an algorithm designed to minimise network distance while preventing worker node overload. Their monitoring system again requires modification of the basic Storm code. This includes sending a signal to their control system every time an operator emits a message, representing a significant communication overhead. Their system, similar to that of Aniello et al. (2013), periodically reads the latest network latency and resource usages and runs their scheduling algorithm to create a new physical plan. Later, a different approach to the same problem was investigated by Eskandari et al. (2016), who created a scheduler that would analyse historical metrics data and use graph partitioning algorithms to minimise remote edges between high-traffic nodes in the topology's physical plan. They also attempt to estimate the required number of worker nodes to prevent the proposed physical plan overloading the available worker nodes.

There are several more examples of iterative systems that use simplistic measures to optimise Apache Storm topologies. Van Der Veen et al. (2015) use a *monitor, analyse, plan, execute (MAPE)* loop based on simple boundary checks (CPU usage, throughput, etc.) to decide when and how to scale the topology. Masotto et al. (2015) use a similar MAPE approach where they continually rebalance the Storm topology and attempt to create a physical plan which minimises tuple waiting times at each executor receive queue (ERQ) and executor send queue (ESQ). However, they treat each executor as an independent entity and do not attempt to optimise the complete end to end latency of a topology.

Peng et al. (2015) have done significant work in the area of resource-aware scheduling in

Storm. They have designed a system, R-Storm, that can take any topology, including cyclical ones, and allocate executors to worker nodes based on the current resource availability on each node. Their system can be tuned to focus on network or processor optimisation. However, it requires the developer to tell the scheduler what the resource and network usage of each component will be. It further assumes that this resource usage will not change over the lifetime of the topology regardless of incoming workload. Liu & Buyya (2017) note this issue with the R-Storm system of Peng et al. and propose a system that continually samples the resource usage of the Storm elements in order to dynamically make scaling decisions. Their system, D-storm, is shown to outperform R-Storm by reducing overall resource usage, including reducing network traffic (by over 16%).

Floratou et al. (2017) introduced an auto-scaling system, Dhalion, for Apache Heron which is the successor to Apache Storm at Twitter and shares many characteristics with it (see appendix D for more details). They began by defining the many benefits of an auto-scaling system, matching some of the points raised in chapter 1, and go on to describe how their system comprises of three distinct components: Symptom detectors, which interface with the metrics from the DSPS and are triggered when certain conditions are met; Diagnosers, which look for certain symptoms being triggered and attempt to identify the cause of those symptoms; and Resolvers, which attempt to fix the cause of the symptoms in question. Whilst this system is general in its design and highly flexible in the way the main components can be applied to a myriad of issues, the approach to scaling outlined in this work is rather simplistic. In essence, if the topology is not processing input tuples at a sufficient rate, Dhalion will simply scale up **all** the components of a topology by an amount proportional to the throughput deficit. If this scale up (or down, if the throughput is significantly over target) does not meet the target, then the process will be repeated again. As Kalavri et al. (2018) note, in their later work comparing their own auto-scaling system's performance to Dhalion's, this blanket approach often leads to extended convergence time (of the order of hours) and to significant over provisioning of resources. Whilst Dhalion is a useful tool to implement auto-scaling in DSPSs, it would benefit from a better informed scaling approach.

All of the systems discussed above for scaling DSPSs use, or are designed around, a reactive approach. The state of the system is monitored and, at set intervals, or when certain thresholds are reached, this information is used with various algorithms to create the optimum arrangement of operators on the cluster. These systems do not require knowledge of an output metric, such as sojourn time or throughput, in order to perform their optimisations. For example, the algorithms that attempt to reduce network latency between heavily connected operators have no concept of what effect reducing connection latency for one set of operators, at the cost of extending it for others, will have on the sojourn time of the entire topology.

Whilst these systems do provide a significant increase in performance over the default DSPS scheduler implementations they are not designed to, and therefore do not, provide the ability to optimise DSPS topologies with respect to a quality of service (QoS) guarantee or SLA.

3.2 Performance Model Based Auto-scaling

Many studies have realised the utility of adding a performance model to the DSPS auto-scaling approach in order to assess the suitability of the physical plans produced by their respective scheduler implementations. Many of the studies focus on predicting resource (CPU, RAM, etc.) usage, which is an important factor in scaling decisions. However, in this review we have focused on those studies that look at predicting application performance metrics such as sojourn time or throughput of the DSPS topologies. These studies can largely be grouped into those based on queuing theory, those based on machine learning and those using other approaches.

3.2.1 Queueing theory

Fu et al. (2015) created a system, Dynamic Resource Scheduling (DRS), on top of the Apache Storm DSPS that attempts to find a topology physical plan that would maintain a response time (end-to-end latency) target for the topology. To achieve this, they modelled the topology's response time using a Jackson network (Jackson, 1957), specifically a $M/M/c$ network where each node of the network can have multiple (c) servers. In this case the nodes are the topology components and the servers are the component's executors. They assume that all the servers of each node in the queueing network are load balanced, which means this model cannot be applied to topologies with fields groupings (see section 2.5). This is because these types of connections will distribute load based on the key (field within the tuple) distribution within the input stream, which may be significantly biased towards specific executors. They use their response time model, in combination with a greedy bin packing algorithm, to select the lowest value of c for each component in the topology that allows the response time target to be met. Whilst their system does select a better performing physical plan than Storm's default round-robin scheduler, the accuracy of their response time predictions is not good. They do not report their accuracy results directly, but their predictions appear to be between 30 and 90% under estimations of the final measured complete latency, depending on the topology and physical plan used. The authors put this significant under estimation down to the fact that their model does not include network transfer latencies between worker nodes. However, whilst this will account for some of the discrepancy, it is unlikely to explain it all — particularly given the order of magnitude difference between the predicted and measured response times. Given that

they are comparing the **average** sojourn time, produced by the Jackson network model, to the *complete latency* which, as described in section 2.13.2, is a worst-case measure of source tuple sojourn time, it is likely that a large amount of their discrepancy was due to the nature of their validation measure. Also, given the behaviour of the executor and worker process queues and their internal processes (described in sections 2.7, 2.8.1, 2.8.2), it is likely that the $M/M/c$ queueing model is a poor approximation of the behaviour of the queuing network nodes, something the results from this study seem to bear out.

Lohrmann et al. (2015) followed a similar approach to Fu et al. to facilitate model-based auto-scaling for Apache Storm and Nephele[2]. They used a queuing theory model to predict topology response time and a bin packing algorithm to create a physical plan which maintains a response time target whilst minimising the required resources. However, unlike Fu et al., who used a $M/M/c$ (Poisson distributed arrival rates with exponentially distributed service times) Jackson network, Lohrmann et al. use a queuing model with general arrival rate and service time distributions. They assume that each executor is a single sever $G/G/1$ queue and use the topology logical plan as the queueing network. They further assume that the load on each executor of a given topology component will be the same. Similar to Fu et al. this means that their topologies can only use shuffle-grouped connections between components. They also assume that all worker nodes in the cluster are homogeneous such that, regardless of where an executor is located, its service time distribution can be assumed to be constant. They model the response time of the $G/G/1$ executor queues by using the Kingman approximation (Kingman, 1961). This formula allows for the estimation of the mean response time, provided that information on the shape of the arrival rate and service time distributions are available and that the queue is close to saturation (experiencing heavy incoming traffic). Their system monitors the end-to-end latency of the topology and if the latency target is breached will attempt to change the parallelism of the components, running each new topology configuration through their model, until they find a configuration that will meet the target latency. They do not report the accuracy of their latency predictions, only how their scaling system allows latency targets to be maintained. They do state that their system continuously models the response time of each executor and produces a *correction factor* between the predicted and measured latency for that executor. It is not clear how they measure the latency across a Storm executor: as described in section 2.8.1 there are several elements which contribute to the latency experienced by a tuple moving through an executor, not all of which provide latency metrics by default. Assuming they have an accurate measure of this time, it is likely that this *correction factor* is compensating for all the latency sources they are not modelling, such as delays in the worker processes and network transfer latency. It is not clear how this correction factor evolves over time with changing conditions or

[2]Nephele was one of the research projects that formed the basis for the Apache Flink DSPS.

how the correction factor may change as a result of a proposed topology configuration change. It seems that the authors have assumed that the correction factor will be constant, regardless of the proposed topology physical plan, which seems unlikely given the amount of potential variance it contains. Also, despite having both the predicted and measured latency values (required to calculate the correction factor) they do not report a latency prediction accuracy.

De Matteis & Mencagli (2016) use a model predictive control (MPC) control approach (often employed in process control in manufacturing) to DSPS auto-scaling. They use queueing theory models to predict the effects on processing latency of changes in the DSPS topology configuration. They also employ various other techniques to predict resource usage of the proposed changes. They predict the incoming workload into the DSPS using time series forecasting methods. They do not specify which methods they use or provide measures of their accuracy. However, they do mention that it becomes less accurate when the workload is more variable. They use this prediction of incoming workload with their latency performance model to assess if a latency SLA will be breached. The time horizon for these forward predictions is not mentioned but it is implied that it is very short (of the order of several tuple processing times). Their latency model assumes each copy of an operator (the equivalent of the executors of a component in Apache Storm) can be modelled as a $G/G/1$ queue and, like Lohrmann et al. (2015), they use Kingman's approximation to find the sojourn time for the operator replica. Their model only applies to a single component with multiple replicas and not to a complex directed graph (DiG) of components; they assume that all replicas will receive an even share of the incoming load and therefore their approach only applies to shuffle-grouped connections and cannot cope with key based distribution. As with Lohrmann et al. they account for errors in their latency model by computing a correction factor between the predicted and measured latency. As stated above, this is likely to cover a large number of missing sources of latency (some of which are unlikely to be linear) and so encapsulating them in a single factor seems unreasonable. It is also worth noting that, as they do not assess the accuracy of their forward workload prediction, this correction factor also encapsulates the errors in that workload prediction. They do not report accuracy measures for their latency although, as discussed above, they presumably had access to them as part of the correction factor calculations and only report the performance of their auto-scheduling implementation.

With a similar goal to Fu et al. and Lohrmann et al., Vakilinia et al. (2016) explore several queuing models which allow for general arrival rate and service time distributions. They state evidence from previous research that "the task response time of stream process [sic] is exponentially distributed" and therefore that they can use a $G/M/c$ queuing model to represent the components of a topology (where c is the number of executors assigned to each component). For occasions when the service time assumption above breaks down they

also looked at the $G/G/c$ queueing model but note that, as it is analytically intractable, they look only at its upper bound. Again, they do not give any accuracy measures for their topology performance model and instead focus on how the model lets them choose a better physical plan than the default round-robin scheduler. It is likely, as with Fu et al. and their DRS system, that an $X/X/c$ model for topology components is too general and does not account for the variation that can occur when executors (the servers of the queuing system) are co-located on worker processes with diverse combinations of executors from other components.

All the Apache Storm latency models based on queueing theory (described above) assume overly simplistic behaviour for the executors within the topology. The batching behaviour of the Disruptor queue (see section 2.7) has been present in the Storm codebase since it was first open-sourced in 2013, and many other DSPSs and network clients implement some sort of batching to improve throughput. It therefore seems strange that none of the approaches above investigated batch-based queueing models, of which there are many variants (Mitrani, 1998; Gross et al., 2008). It is also disappointing that almost none of the studies report the accuracy of their latency performance models. Although it could be argued that the focus of their research was on optimal scheduling or resource usage it would have been useful, for those of us concerned with modelling application performance, to have an indication of the effectiveness of the applied models. The exception to the rule is Fu et al. (2015), who show that a $M/M/c$ open Jackson network appears to be a poor approximation for an Apache Storm topology. However, Fu et al. — as well as many of the other studies mentioned — appear to use Storm's *complete latency* metric (see section 2.13.2) to assess their predictions. As mentioned previously, this measure is highly susceptible to positive skew due to outlier measurements. This fact may well explain the large under-predictions reported by Fu et al.. It may also explain why, if they were significantly under-estimating the reported complete latency, others did not report the accuracy of their predictions. If they were putting their test topologies under particularly high load, some tuples may have been delayed, skewing the *ground truth* measure. Further discussion of this issue is given in section 5.7.1.

3.2.2 Machine learning

As well as the heuristic-based algorithms and queueing theory based performance models discussed above, there are several examples in the literature of applying a machine learning approach to DSPS auto-scaling.

Lorido-Botran et al. (2014) provided a summary of various auto-scaling strategies for cloud based applications and covered reinforcement-learning in detail. Essentially reinforcement-learning, is a method by which the auto-scaler learns the best course of action for a given

state via a trial-and-error approach. A course of action is chosen and the effect of that action on the current state is recorded and passed through a cost function. In future iterations the auto-scaler uses the cost of past actions to choose the action for the current state. In this way, the best course of action is reinforced over time and many different approaches have been used to choose this correct course (Barrett et al., 2013; Tesauro et al., 2006; Bu et al., 2013).

Specific to DSPSs, T. Li et al. (2016) used a supervised-learning approach involving a support-vector-regression (SVR) model which takes various parameters of a running Apache Storm topology, trains a model and then uses the proposed changes to the topology configuration (from a newly proposed physical plan) to predict the expected end-to-end latency. The use of SVR allows them to create a high-dimensional model with parameters which are not linearly related. They report a prediction accuracy of greater than 83% for the three test topologies in their validation and show how this model allows their greedy bin packing algorithm to iterate to a physical plan with the lowest possible end-to-end latency for a given set of available resources (worker nodes). It is worth noting that their test topologies use only shuffle-grouped connections and they assume that all executors of a component will receive equal loads. Whilst their accuracy results are impressive, beyond saying that random configurations were generated they do not give any details on the length or complexity of their training phase. For example, the number of training configurations or the parameter ranges covered are not reported and they also do not state if the model was tested and trained on the same set of configurations. It would be useful to know how accurate the latency predictions are for topology configurations that were not in the training set and also to have an idea of how long a reasonable training session would be for this system.

Foroni et al. (2018) pursued a similar approach to Li et al., but applied to Apache Flink. They use both a linear-least-squares and support-vector-machine (SVM) based approach to create models mapping input workload to various application-level performance metrics such as topology throughput and sojourn time. They use the models with their monitoring framework, to tweak the various parameters that Apache Flink exposes until the performance targets are reached. As with Li et al., they do not report any accuracy measures for their performance model and only show how their scheduling implementation is better than the default Flink round-robin version.

Jamshidi & Casale (2016) propose a Bayesian optimisation approach, using Gaussian processes (GPs), to choose an Apache Storm topology configuration to meet a given performance target. They highlight how the relationship between the topology configuration and end-to-end latency is non-linear and multi-modal, and how the performance of one element of the topology will influence the others in the topology and therefore that acting on only one parameter at a time may not lead to a global optimum. Their GP models are

trained on many topology parameters and they treat the topology itself as a black box. They show their model's root mean squared error (RMSE) for several of their algorithms as a function of the number of modelling iterations (each 5 minutes long). For example, one of their algorithms, Branin, for a simple word-counting topology, starts with a RMSE of over 100 after 5 iterations (25 mins) and drops to less than 0.1 after 100 (8 hours 20 mins), whilst another, Dixon-Szego, starts at 2.5 after 5 iterations and drops to a consistent 0.7 after 40 (3 hours 20 mins). This is a good level of accuracy, however to reach it a significant period of training time is required. It should be noted that these levels are for a simple word-counting topology with a low level of parallelism. When the complexity of the topology is increased the best achieved RMSE after 80 iterations (6 hours 40 mins) increases from 10^{-3} to 20. As you would expect, other, more complex topologies suffer even larger errors (up to a RMSE of 100) even after hundreds of iterations. However, it is refreshing to have accuracy results to compare and their method does show high accuracy for simpler, linear topologies. Another significant feature of their method, besides using a non-linear modelling approach, is that it provides a measure of uncertainty with each performance estimate. This uncertainty information allows their optimisation system to make an informed decision on whether to act on a given prediction for a physical plan.

Lombardi et al. (2019), building on earlier work (Lombardi et al., 2018), present an auto-scaling system, PASCAL, for Apache Storm (and the Apache Cassandra[3] distributed database) based on artificial neural networks. They train the neural network on past workload traces to create a future workload prediction. They train another neural network on historic performance metrics from a running topology in order to predict the resource usage and performance of a proposed physical plan. As with Li et al., they do not report the accuracy of their performance model, although they do state that their workload predictor (tested against seasonal workloads with a regular period) had a RMSE of 3%. They focus their results on their scheduling implementation, which is able to create a physical plan that maintains a sojourn time target whilst minimising resource (in this case CPU load) usage. Again, as with Li et al., they do not give details on the overall length of the training phase required to create their models or whether test configurations were present in the training data. Their system was tested on a realistic, linear topology and does implement proactive scaling. However, there are no accuracy measures for their topology performance modelling.

Gautam & Basava (2019) use a similar approach to that of J. Li et al. (2016) and Jamshidi & Casale (2016), but applied to Apache Heron (Storm's successor). They use SVR in combination with GP models to predict the resource usage of proposed Heron topology physical plans and also predict the *execute latency* of the topology, which is the time taken for each instance (Heron's equivalent of Storm's executors) in the topology

[3] https://cassandra.apache.org/

to process incoming tuples. They do not appear to predict the overall sojourn time or throughput of a Heron topology. They test against a wide variety of topologies and report prediction accuracies of up to 81% for their resource usage estimations. They do not report accuracy figures for their *execution latency* predictions, but do show charts that show their modelling approach produces good accuracy for simple topologies with some significant over-estimation of the latency for more complex topologies. There is no discussion of the amount of training data required or comparisons of the prediction accuracy for seen and unseen physical plans.

Key to all reinforcement-learning methods is a cost function to allow the system to learn the best strategies. In the systems reviewed by Lorido-Botran et al. this is either based on resource usage or application metrics resulting from the deployed operator plan and/or horizontal scaling of that plan. The issue with this approach, which is identified by Lorido-Botran et al. and is discussed in section 1.2, is that it requires a physical plan to be deployed and run before its effectiveness can be assessed, leading to long training periods and issues when unique states (that are not like any previously run state) are encountered. It is encouraging that several of the studies discussed above do report prediction accuracy for their performance models. However, it is disappointing that none of the studies mentioned above, with the exception of Jamshidi & Casale (2016), give details of the length of the training period required to reach the prediction accuracy they report. This makes it difficult to assess the trade-off between the extended training period and the accuracy of machine-learning based approaches compared to other modelling methods. In the case of Jamshidi et al. they can obtain good prediction accuracy but require between 3 and 9 hours to obtain them, even on simple word-counting topologies.

3.2.3 Other approaches

Farahabady et al. (2016) use a MPC approach to auto-scaling Apache Storm topologies. They use a performance model that predicts the uncontrollable variables, such as the arrival rate into the topology, which allows the estimation of the effect of those variables on the topology's performance to be gauged. It then follows the MPC processes to select the controllable variables, primarily the component parallelism and number of worker nodes, such that a given performance target can be met. They predict incoming workload into the topology using an auto-regressive integrated moving average (ARIMA) time-series model, however they do not give any details of the performance model they use to gauge the effect of this arrival rate, instead referring to general texts on MPC. Consulting these referenced texts seems to suggest that the performance model is linear in nature, relating incoming arrival rate to end-to-end latency. Their results show that the MPC controller performs significantly better than Storm's default round-robin scheduler at maximising resource utilisation whilst minimising end-to-end latency QoS targets. However, they do

not report accuracy measures for their system performance model. The fact that they minimise the breaches of a latency target does seem to suggest that the model has some useful predictive properties, however the fact that they still report breaches of the target, up to 18% of the time, indicates that there must be errors in the predictions. A further point to note with this system is that it is one of the few systems that provide a proactive, rather than reactive, auto-scaling approach. Their use of an ARIMA model to predict incoming workload allows the scaling operations to happen **before** the workload arrives, limiting QoS violations.

Sun et al. (2018) created a DSPS auto-scaling system, E-Stream, using Apache Storm as a test bed. This system uses a topology performance model, developed in their earlier work (Sun & Huang, 2016), and is based on both sojourn time, which they refer to as the *makespan*, of the topology and its resource usage. Their sojourn time model calculates a processing cost (in milliseconds) for each vertex and each edge between vertices executors in the topology's logical plan. It then creates a timeline through the logical plan by identifying the critical path (those nodes which will dominate the sojourn time measure). They calculate the *makespan* of the topology from this critical path. They use this model to compare the expected sojourn time of each of the physical plans produced by their scheduling implementation and select the one with the fastest performance and lowest resource usage.

The key components of the topology performance model, created by Sun et al., are the vertex processing and edge communication cost functions. The vertex cost function is based on the number of instructions (in the case of Apache Storm these are Java byte code instructions) that the vertex contains, scaled by the vertex's processing ability which is the number of instructions per second it can process on a given worker node. They scale the processing ability by a *performance degradation percentage*, which is a parameter that captures how adding more executors to a worker process can result in the executor latency extending due to the multi-threaded nature of the worker processes. Although they never explain how this parameter is calculated, this is nevertheless an important insight and one that is not addressed in other studies featuring Apache Storm. The use of instructions and instruction-processing rate to infer a processing time is an interesting one and could allow the extension of the execute latency because of *thread-pausing* to be excluded. However, it is not clear how they identify the number of instructions for a user defined bolt instance or how they account for execution branches (conditional code blocks such as *if* statements) which will form part of the instruction count but may not be executed. Furthermore, it is not clear why you would want to exclude the thread pausing effect as it will be experienced by tuples moving through the topology and therefore will contribute to the measured sojourn time (in this case, complete latency) reported by Storm. Indeed, the *performance degradation percentage* seems to add the thread pausing

effect back into the processing cost measure. However, as they do not explain how they calculate the degradation percentage it is difficult to discern if this is actually the case. Similarly, their edge communication cost function depends on this degradation percentage and, whilst it is important to account for the fact that remote transfers between worker processes could have a significant effect on the sojourn time, without an explanation of how they calculate this it is hard to gauge its effectiveness. Whilst their model includes some innovative features, they do not report any accuracy measures for their sojourn time predictions. They only report the improvement over Storm's default round-robin scheduler that their performance-model based scheduler implementation achieves.

Kalavri et al. (2018) proposed an auto-scaling system (DS2) for Apache Flink and their own Timely DataFlow DSPS, which is reactive and will trigger when the throughput of a system drops below a given threshold. DS2 creates a model of the streaming topology based on the *useful time* each operator (the Flink equivalent of Storm's executors) in the topology spends processing tuples. They define this useful time as the fastest a topology operator could process a tuple and this includes the de-serialisation of the input tuple, user-defined logic processing and serialisation of the output tuple(s). From this useful time they calculate the true processing (input) rate of an operator and its true output rate. When their system detects that throughput targets are not being met, they calculate the current true output rate of each operator of each component in the topology. The true output rate is calculated from historical metrics data from the last time step: using this and the true processing rate of each downstream operator they calculate the number of operators needed to match the corresponding upstream output rate. This assumes, which they freely admit, that each operator will receive the same load and that the aggregate processing rate for a component is a linear combination of its operator's processing rates. This assumption means that their approach could not be used for topologies with key-based connection routing (referred to as fields grouping in Apache Storm). Their approach shows significant improvements in the time taken to find a physical plan to reach throughput targets compared to current auto-scaling systems like Dhalion (Floratou et al., 2017) (1 min for DS2 vs 33 min 20 sec) and avoids over provisioning resources to meet those targets (30 operator replicas for DS2 vs 52 for Dhalion). Whilst this method has its limitations — for example, it is reactive and is not able to simulate the effect of higher input traffic on the throughput of the topology — it is strong evidence for the significant speed advantage that is gained by adding performance modelling (of any kind) to the scaling decision loop.

3.3 Summary

The studies reviewed above show how much of the early work on auto-scaling for DSPSs was focused on threshold-based reactive systems that would respond to overload within

the topology by adding resource and finding one of many optimal physical plans. However, as noted by Kalavri et al. (2018) this results in a system "which is unable to consider the structure of the dataflow graph or computational dependencies among operators". This succinctly describes the issue faced by auto-scaling systems that do not utilise some form of performance model: they are effectively wandering in the dark trying to find a solution that meets the required performance target. As Kalavri et al. highlighted in their comparison with Dhalion, this can lead to slow converging and inefficient physical plans.

The advantages of providing a performance model as part of the auto-scaling system are well established in the studies described in section 3.2. Almost all demonstrate the ability to select a better-performing plan than the default scheduler and in some cases do this whilst maintaining a performance target, be it latency/throughput, resource use or both. However, there is a general lack of focus on the accuracy of their performance predictions, particularly the application-level measures such as sojourn time and throughput. For those studies that do not provide accuracy measures for their performance predictions, this can partially be attributed to their focus on their scheduling implementations. If a performance model produces significant errors then it is likely to produce these for every physical plan. Therefore, provided that the error is the same for each proposed physical plan, it is largely irrelevant if selecting the best plan is the priority. However, this de facto assumption of consistent error, regardless of the physical plan, is a strong one. Many aspects of a physical plan can have significant performance effects, and assuming a linear relationship between a given model's error and all the possible parameters of a given physical plan seems unreasonable. Furthermore, maintaining a given performance target is clearly more difficult if you do not know the accuracy of the performance model.

This seems to be more of a problem for those studies using a classical queueing theory approach. The few studies that did report accuracy measures for their predictions showed significant under-predictions of the sojourn time. As discussed above this could be the result of several factors including ignoring the batch processing that takes place at a low level within many DSPSs, particularly in Storm executors, and using the complete latency as a validation measure, which is particularly susceptible to positive skew. It is also important to note that none of the queueing theory based models took into account the DSPS system components that could add additional latency. For example, in Apache Storm, the executor send thread (EST) (see section 2.8.1) and worker process send thread (WPST) (see section 2.8.2) both contain queues and processes that will add additional delays. None of the studies that used Storm as a test-bed included these elements. It seems likely that, before this approach can begin to deliver more accurate results, more-advanced queueing models and queueing network representations would need to be investigated. However, the advantage of using a queueing theory modelling approach is that it allows the encoding of many of the internal characteristics directly into the model. This means

that, theoretically, these models could function on a relatively small amount of input data.

The machine-learning based models report higher accuracy levels than those based on queueing theory. Most models appear to use either a SVR or GP based approach, which allows them to account for non-linear relationships between the performance (and resource usage) of the topology and its attributes. Whilst these systems do report better accuracy than any of the other methods reviewed, they suffer from the need for an extended training period where sufficiently diverse topology configurations and workload conditions are seen. Almost none of the studies describe the duration or complexity of their training periods. However, based on the reviewed papers, it seems that a training period could be between 3 and 9 hours even for simple linear topologies. Without a more thorough investigation of the training period requirements it is hard to reason about the trade-offs of using a machine-learning based approach in order to gain better accuracy.

There is a common characteristic of almost all the studies mentioned above: most assume that the processing elements of a DSPS topology receive equal load and therefore only support the modelling of topologies with components linked with shuffle-grouped connections. This simplifies the modelling assumptions but rules out a very large class of topologies that route tuples by key, the simplest example being the word count topology. There is clearly a need for a performance-modelling system that can produce accurate results for topologies with key-based (fields grouped) connections. If this model could also produce accurate results without a significant training phase then this would go a long way towards addressing some of the issues with the studies discussed in this review. Such a modelling approach is the goal of this thesis.

Chapter 4

Topology Performance Modelling

This chapter describes the methodology used to model the performance of a proposed Apache Storm topology physical plan. Specifically, we focus on the *end-to-end latency*, which we define as the time from a tuple being emitted into the topology (from the spouts) to completion of processing in the final sink component (the Acker bolts). As described in chapter 1, distributed stream processing systems (DSPSs) are employed primarily, in configurations like the Lambda Architecture, to provide **rapid** responses to queries. The end-to-end latency of a topology is therefore of paramount importance to its users and a common basis for performance orientated service level agreements (SLAs).

Section 4.1 gives an overview of the modelling procedure. Section 4.2 to section 4.13 cover the modelling methods for the various elements outlined in the modelling procedure. Finally, section 4.14 summaries the modelling process.

4.1 Performance Modelling Procedure

The aim of the performance modelling is to predict the average end-to-end latency of a proposed topology physical plan in order to aid in selecting the most appropriate physical plan to meet a latency SLA. In order to avoid the extended training phases linked to machine learning techniques (see chapter 3), among other reasons, a queueing theory based approach was chosen. Section 4.1.1 describes the various approaches from queueing theory that were investigated in order to model the topology's end-to-end latency. A primer on queueing theory notation and terminology is given in appendix A.

4.1.1 Modelling the topology

The end-to-end latency of a topology is a function of the delays experienced by tuples passing though the topology's elements, executors and worker processes. At the most basic level we could consider, as a basis for the end-to-end latency modelling, the topology's query

plan (see section 2.6.1) with each component being a multi-server queue. Alternatively, we could use the topology's logical plan (see section 2.6.3) and treat each executor as a single sever queue. Regardless of which topology plan we use, they both look very similar to a queueing network. Mitrani (1998) defines such a network as:

> "... a connected directed graph whose nodes represent service centres. The arcs between those nodes indicate one-step moves that jobs may make from service centre to service centre... Each node has its own queue, served according to some scheduling strategy. Jobs may be of different types and may follow different routes through the network... A network is said to be 'open' if there is at least one arc along which jobs enter it and one arc along which jobs leave it, and if from every node it is possible to follow a path leading eventually out of the network."

Open queueing networks

Open queueing networks have analytical solutions that allow us to predict various performance characteristics. For example, the total average network end-to-end latency W can be found, assuming all nodes are M/M/1 queues, using:

$$W = \frac{1}{\gamma} \sum_{i=1}^{N} \frac{\rho_i}{1 - \rho_i} \tag{4.1}$$

Where γ is the sum of all external arrival rates into the network nodes and ρ_i is the offered load of the i^{th} node out of N total nodes in the network. This equation comes from Jackson's theorem (Jackson, 1957) (for open queueing networks) which often leads to open queueing networks being referred to as *Jackson Networks*.

Equation 4.1 shows how an estimate for the end-to-end latency of a proposed physical plan could be calculated. However, in order to be able to use this approach, several conditions have to be met. Mitrani (1998) lists these as:

1. Every node contains a single server, with an unbounded queue which is served using the FIFO discipline.
2. The service times at node i are distributed exponentially with mean b_i (the service rates follow a Poisson distribution with mean μ_i).
3. Jobs arrive into node i in an independent Poisson process with rate λ_i.
4. A job arriving at node i goes to node j with probability q_{ij}, regardless of its history.

The above list is the set of requirements for the simplest situation that has an analytical solution. Scenarios involving queues other than the M/M/1 system have been investigated

in the literature (Mitrani, 1998; Gross et al., 2008). However, these solutions share similar requirements to those listed above.

Generally, point 1 stands for topologies logical plans as the executor receive queues (ERQs) contain an unbounded overflow queue (see section 2.7) and use a FIFO discipline. The service times however may not be exponentially distributed (see section 4.9 for more details), which may suggest the use of a general service time solution to the open queueing network (Meyn & Down, 1994). Point 3 requires jobs (tuples) to enter the system according to a Poisson process which, as described in section 4.3, may not be the case for all topologies. Point 4 is true for executor to executor communications, however these can be different depending on the input and output streams and so the routing probabilities may not be memoryless. Also, Jackson networks deal with jobs passing one to one from server to server. But in Storm topologies, tuples are often aggregated (windowed) or multiplied within the executors, which complicates things further. Add in the batching behaviour of the Disruptor queues, as well as the batch processing of the user logic thread (ULT) (see section 2.7 and 2.8.1), and it becomes clear that, in order to account for a topology's performance, a batch arrival with bulk service open generalised Jackson network ($G^X/G^Y/1$) would be required, a fact supported by the low accuracy reported by Fu et al. (2015) and their $M/M/c$ Jackson network (see section 3.2.1). To our knowledge no such treatment is present in the literature.

It may be that an existing Jackson network formulation could be used as an approximation for a Storm topology. However, they require specific criteria to be present in order to use them and some of these criteria may be present for some topologies and not for others. Assessing this *a priori* would be difficult and finding a queue network model to fit all criteria may not be possible. Therefore, a more general approach to topology performance modelling was pursued.

Closed queueing networks

It is also worth mentioning that as well as open queueing networks, like those discussed above, there are networks which are 'closed' in nature. These networks have a fixed population of jobs which move around the network indefinitely. This scenario has some useful properties which allow performance measures to be derived easily (Gordon & Newell, 1967). Mean value analysis (MVA) (Reiser & Lavenberg, 1980) is one such closed queuing network approach and has been used in DSPS performance modelling previously (Wang et al., 2006). However, in order to use closed networks to simulate open networks, such as Storm topologies, we have to assume that every job that leaves the network will be instantaneously replaced by a job entering the network. This assumption may be valid for topologies that are experiencing very high incoming workload and so are almost at saturation. However, whilst this is a desirable state for a topology to be in from an

efficiency point of view (as it indicates that the topology is correctly configured for the input workload) it is very rarely the case in practice. As with the Jackson network approach discussed above, the limited number of scenarios that it applies to means that MVA is not a good fit for providing a generalised performance modelling approach for Storm topologies.

Individual queues

Given that treating a topology as a single queueing network, open or closed, seems infeasible the next option was to look at a topology as a collection of individual queueing systems. These queueing systems are connected in a network and tuples pass along paths in this network, experiencing delays at each node. Each path will have a total end-to-end latency value and the various paths in the topology can be analysed to produce summary statistics on the performance of that topology as a whole. However, to facilitate this kind of analysis and incorporate all the main sources of latency a tuple would encounter, a new way of representing the topologies was required.

4.1.2 Tuple flow plan

Section 2.8 describes the various elements tuples encounter when passing through the executors and worker processes assigned to a topology. From the descriptions in that section, it is clear that the actual tuple flow is more complicated than that depicted by the topology's logical plan (see section 2.6.3). To discern all the elements a tuple will encounter, a more comprehensive representation of a topology is required, one which takes into account the worker processes and network transfers as well as the executors.

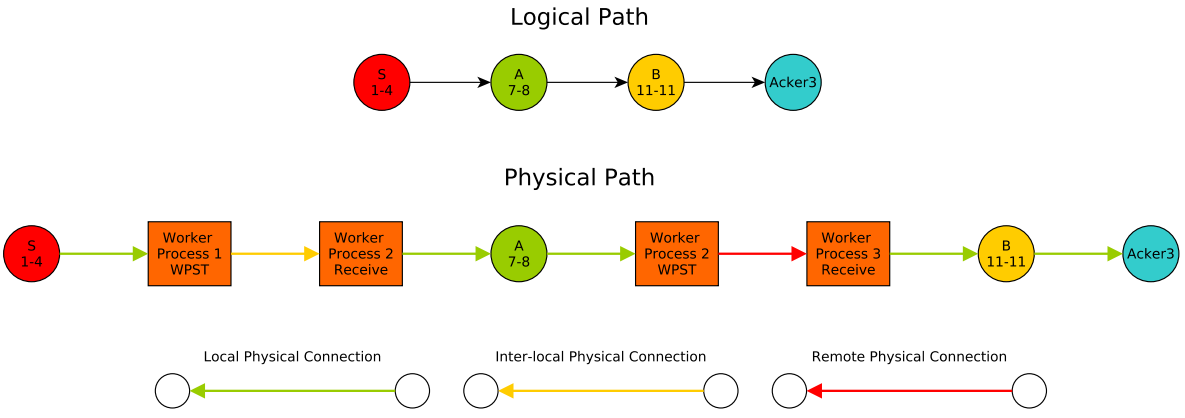

Figure 4.1: An example logical and physical path through the topology shown in figure 2.3.

Using the topology logical plan shown in figure 2.3 as an example, figure 4.1 shows a particular path a tuple could take through the logical plan (we call this a *logical path*) from $S_{1-4} \rightarrow A_{7-8} \rightarrow B_{11-11} \rightarrow \text{Acker}_3$. This logical path includes both the delay at the ERQs,

service delay in the ULT and the delay through the executor send queue (ESQ). However, it does not include the delay encountered whilst passing through the worker processes or across the network. If we include these elements we get a path like the second one displayed in figure 4.1. This *physical path* includes the worker process send thread (WPST) (including the worker process transfer queue (WPTQ)) and the worker process network receiving logic which includes the de-serialisation and sorting functions which route tuple lists to the relevant ERQs (see section 2.8.2). The connections in a physical path are colour coded to show the type of network transfer they represent: green for *local* connections within worker processes; yellow for *inter-local* connections between worker processes on the same worker node; and red for *remote* connections between worker processes on separate worker nodes.

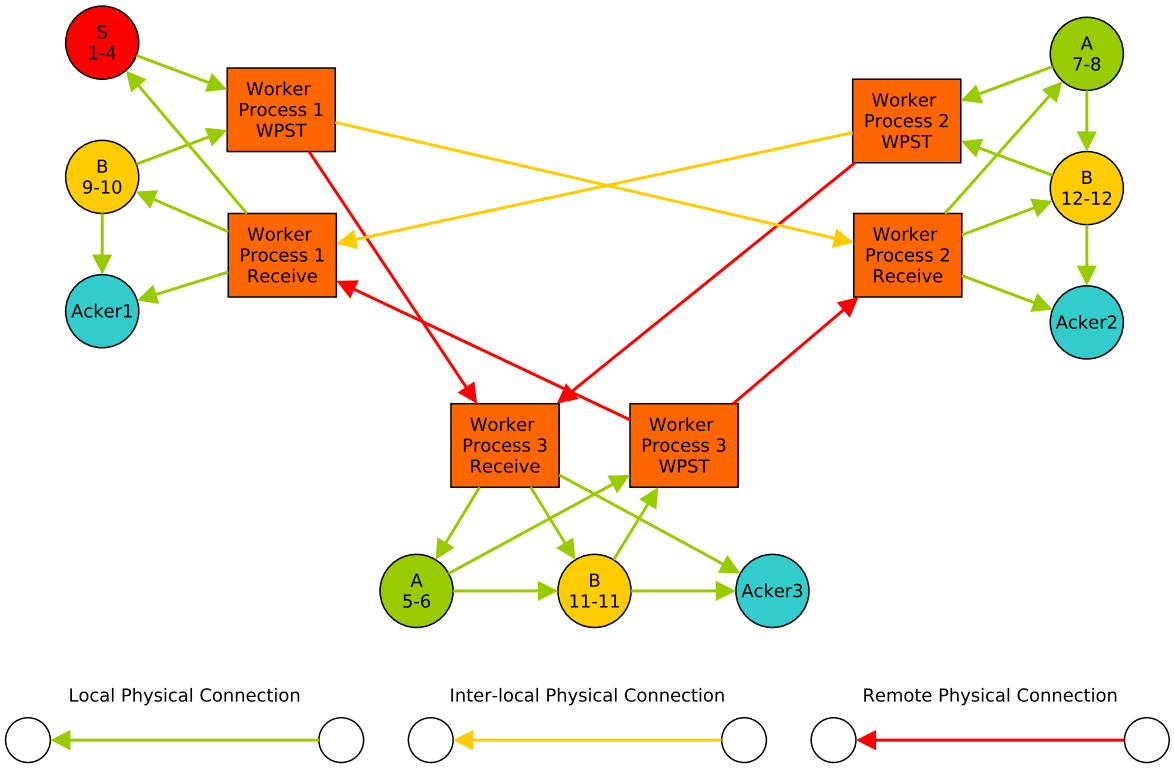

Figure 4.2: The tuple flow plan for the simple linear topology shown in figure 2.3.

If we apply the physical path approach to all paths in the topology logical plan, we obtain a new topology representation called the *tuple flow plan*. Figure 4.2 shows an example of the tuple flow plan for the topology shown in figure 2.3. The topology logical plan allows us to discern the paths tuples will follow between executors and the tuple flow plan allows us to identify what physical elements the tuples on those logical paths will encounter.

4.1.3 Elements to be modelled

Using the tuple flow plan we can identify the elements encountered by tuples along each path in the topology. In order to calculate the expected end-to-end latency for those

paths and the topology as a whole, we need to predict the expected latency for each of the encountered elements, namely the executors, the worker processes and the network transfers.

Executors The delay due to the executor's ULT, which includes the ERQ and tuple processing in the tasks, is covered in section 4.2. The delay due to the executor send thread (EST) is discussed in section 4.13.2.

Worker processes The worker process delay is due to tuple processing in the WPST which consists of the WPTQ and associated operations. The modelling of this delay is discussed in section 4.13.3.

Network transfer times New arrangements of operators on the cluster will affect the expected transfer times of tuples between elements. Introducing more remote network connections will affect the end-to-end latency. The approach to modelling this aspect is detailed in section 4.11.

In addition to the elements above, the parameters listed below will need to be predicted for proposed physical plans:

Arrival rates The expected tuple arrival rate, at the executors and worker processes of a proposed physical plan, depends on many factors. These include the routing probabilities of outputs sent to downstream destinations in the tuple flow (connections between logically connected elements of the topology) (analysed in section 4.4) and the input to output coefficients of each topology element (studied in section 4.6). The overall approach to predicting the arrival rates of each topology element for a given incoming workload is detailed in section 4.8.

Incoming workload The expected incoming workload into the topology (the expected emission rate from the spouts) is a key variable in the prediction of a proposed physical plan's performance. The incoming workload is also the initial state for the arrival rate calculations detailed in section 4.8 and discussion of the prediction of this parameter is given in section 4.3.

Service times The expected service times for the executors in the proposed physical plan can be affected by changes in the number and type of executors that are co-located on each of the topology's worker processes. The approach to predicting this parameter is given in section 4.9.

Input tuple list size This is the number of tuples that are expected in each of the input lists that arrive at the ERQ (see section 2.8). This value is important for the prediction of the executor latency detailed in section 4.2. The estimation of the input tuple list size is discussed in section 4.12.

Once estimates of the delay due to the elements of the tuple flow plan have been found for a given proposed physical plan, it is then possible to look at modelling end-to-end latency of each path through topology tuple flow plan. The approach for this is detailed in section 4.13.

4.2 Executor Latency Modelling

The complexity of the Disruptor queue (described in section 2.7) means that simple queueing models, such as the $M/M/1$, are unlikely to be appropriate for the executors. This theory is supported by the low accuracy reported by studies using these models, discussed in section 3.2.1. The primary challenges in modelling the expected latency at the executors are:

Service times The multi-threaded nature of the executors means that tuple service can be paused arbitrarily by the Java Virtual Machine (JVM) or operating system (OS), and that these pauses are likely to get longer the higher the number of executors that are co-located in a worker process. This means that exponentially distributed service time cannot be relied upon. Further discussion of these issues is given in section 4.9.

Batch arrivals and internal transfers Recall that section 2.8 had show how tuples arrive into the ERQs and WPTQs in batches. Storm does not provide details of the sizes of these batches, nor does it provide arrival rate information on the batches, but only provides details of the tuples placed onto the Disruptor queue Ring Buffer. These issues mean that batch arrival queueing models, even with general service distributions, are not strictly applicable without modifying Storm to report these metrics or inferring the size of these batches. These issues are addressed in section 4.12.

Bulk fetch and sequential service Tuples are taken off the Disruptor queue Ring Buffer in batches (see *internal buffer* in section 2.8.1). This implies the use of a general bulk service queuing model, many of which exist in the literature (Sasikala & Indhira, 2016). However, whilst the tuples are taken off the queue in bulk they are served sequentially. This forms another queue within the system and this situation is not covered by the bulk service queuing models in the literature.

Timed queue flush The Disruptor queue will move input batches into the overflow queue when either the batch size limit k is reached or the flush interval δ (typically 1 ms) completes. This combination of two distinct clearance mechanisms is intended to prevent low arrival rates causing tuples to wait in the input batch for extended periods. However, it also means that none of the standard queueing models will

accurately cover the Disruptor queue behaviour. This is because batch arrival and bulk service models are based on fixed size limits for the batches and the flush interval means that batch sizes are potentially highly variable.

Also, due to the timed queue flushing, there is now a minimum queue waiting time (l) for all queues whenever the arrival rate (λ) is less than k arrivals within δ time $\left(\lambda \leq \frac{k}{\delta}\right)$. This input batch waiting time will be reduced with higher arrival rates, although waiting time in the overflow queue may increase with higher arrivals, but none of the standard queueing models will account for this behaviour. Therefore, an alternative approach is required to accurately model the executors.

4.2.1 Queue simulation

For the reasons outlined above, a classic queueing model is unlikely to provide accurate predictions of the performance of the Disruptor queue and executor ULT combination. When systems become too complex to model and derive closed form analytical approaches, an option is to simulate the system in order to gauge its likely performance (Gross et al., 2008).

The simulation of queueing systems can take advantage of the fact that the state of these systems is discrete. It only changes by a set amount according to which of several predefined events (job arrival, service completion, etc.) occur. This situation lends itself to a discrete-event simulation (DES) approach.

Discrete-event simulation

A DES models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. There are three typical approaches in DES:

Activity oriented Activity oriented DES divides time into discrete steps and advances through a set time range, altering the state as the time of given events is reached. This form of simulation is very slow as the simulation has to pass through all time steps even if nothing is changing in the system.

Event orientated Event oriented DES uses the assumption that the system's state doesn't change between events and therefore these inter-event periods do not need to be simulated as in the activity oriented case. The simulation maintains a set of pending events and will find the next event in the set, advancing the simulation time to match that event. The simulation has logic for adding events to the set and for how each event type changes the system's state.

Process orientated Similar to the event oriented DES, process oriented DES only deals

with events and not the time between them. However, event oriented DES performs all its operations in a single serial process or thread which may contain logic for several distinct entities. Process oriented DES, in comparison, uses separate processes (threads) for each entity in the simulated system. These separate processes generate and consume events from the pending events set and pass events between each other. The advantage of this approach is that it can be more modular in nature, as the logic for each entity is self-contained, which can ease understandability of the simulator and re-usability of its various elements. However, it does introduce concurrency and its associated complexities into the logic of the simulator implementation.

Although process oriented DES can yield more modular, easily reusable code, creating these types of simulators can be complicated. Therefore, in order to model the response time of an executor, an event orientated DES approach was adopted.

4.2.2 Executor simulator

Section 2.8.1 outlines the various elements of an executor that a tuple will pass through. These can be split into two sections:

- User logic thread (ULT): which contains the ERQ and the tasks containing the user defined tuple processing code.
- Executor send thread (EST): which contains the ESQ and the code which sorts the outbound tuples into the local dispatch list and the remote dispatch map.

We focused our simulation design on the ULT elements of the executor. The ULT is likely to contribute the majority of the executor's response time, also there are several metrics which Storm does not provide for the EST which are vital for accurate simulation. Discussion of how the delay, due to the EST operations, is accounted for is given in section 4.13.2. A detailed discussion of the executor ULT simulator implementation is given in appendix B.

4.3 Incoming Workload

As described in section 1.3.3, one of the key advantages that a performance modelling system brings to DSPSs is the ability to assess the effect of future workload on the performance of a currently running topology.

In the case of Apache Storm, the incoming workload into the topology is the expected output from each of the spout executors. Predicting this is highly dependent on the topology's application. For example, a topology that is processing tweets will see strong seasonality in the input rates as a results of users sending more messages in the mornings and evenings. Conversely, a topology that is taking input from other machines, such as

parsing server logs or data from a sensor network, may have a steady or highly bursty input profile (Calzarossa et al., 2016).

The performance modelling system detailed in this thesis is designed to accept values for the expected emission rate from each spout in the proposed topology physical plan. Naively, this could be based on the average output of all spouts from the running topology over a given period or it could be a prediction from a more sophisticated method, many of which have been assessed for auto-scaling and autonomous system management (Huifang Feng & Yantai Shu, 2005; Herbst et al., 2017). There are many *off-the-shelf* packages available for time series predictions that could be used with historical emit count data from the topology's spouts to predict future workload. For example, Facebook's open source *Prophet*[1] software is able to perform predictions using a generalised additive model approach to take account of both trends and seasonality in time series data (Taylor & Letham, 2018). More detail of the application of packages such as this, to DSPS workload prediction, are discussed in chapter D.

Whilst the prediction of future workload for an existing topology may be possible via an off-the-shelf method, this would not take into account the effect of changing the parallelism of the spout components themselves. Predicting how a change in the number of spout executors would affect their emission rate would require developing a model of the upstream infrastructure that supplies data to the spouts. This could be highly specific to each application and so it was decided to leave this aspect to future research, see section 6.3.2. For the time being, the performance modelling system has been designed to accept a workload prediction so that a future prediction system could be incorporated easily.

When it comes to validating the end-to-end latency prediction of the performance modelling system, the metrics from a topology running with a *source* physical plan are used to predict the end-to-end latency of the same topology with a proposed physical plan. The validation process, described in chapter 5, records metrics for the topology running under both the source and proposed physical plans, before running the prediction system. In this way the true performance of the proposed physical plan is known. This allows the true incoming workload of the proposed physical plan to be supplied to the modelling system as a workload "prediction" along with the source physical plan metrics. Essentially this represents a perfect workload prediction.

4.4 Routing Probabilities

In order to calculate the tuple arrival rates into the executors of a proposed tuple flow plan, we need to know how the tuples will be routed along the various logical connections between

[1] see: https://facebook.github.io/prophet/

executors in the proposed logical plan. The proportion of tuples that will be routed along each logical connection is dictated by the *routing probability* of that connection. This describes how likely a tuple leaving an executor is to pass along a given connection. There are two forms of routing probability used in the performance modelling calculations: the *stream routing probability (SRP)* which is defined in section 4.4.1 and the *global routing probability (GRP)* defined in section 4.4.2.

4.4.1 Stream routing probability

When designing a Storm topology the user will create the query plan (see section 2.6.1). This dictates which components are connected and what type of connections (shuffle, fields grouped, etc. — see section 2.5) they are. Each connection (referred to as a stream in Storm) between components in the query plan can be given a unique name. This allows multiple outgoing connections to be defined for a given component, which facilitates "splitting" incoming streams and dividing output into distinct streams for downstream components to subscribe to. It is also possible to define multiple streams between two components.

The SRP $R^S_{i,j}$ is the likelihood that a tuple emitted from a source executor i onto stream S will be sent to a downstream destination executor j. Where $0 \leq R^S_{i,j} \leq 1$ and for each output stream S from the source executor i to the set J_i of all destination executors j downstream of i:

$$\sum_{j \in J_i} R^S_{i,j} = 1$$

The prediction of the expected SRPs for logical connections between executors in a proposed logical plan is dependent on the stream grouping (see section 2.5) of those connections:

Shuffle grouping

Shuffle grouped logical connections (see section 2.5) are *load balanced*, meaning that tuples are routed down each downstream connection with equal probability. Using the topology query plan shown in figure 4.3 as an example, *Stream-1* between components S and A has a shuffle grouping. Tuples traveling from any of the executors of component S, along Stream-1, will have an equal probability of being routed to any of the executors of component A. In the case of the logical plan shown in figure 4.3, the two logical connections from executor S_{1-2} ($S_{1-2} \rightarrow A_{5-6}$ and $S_{1-2} \rightarrow A_{7-8}$) will both have an SRP of 0.5, as will the two outgoing connections from the other two executors of component S.

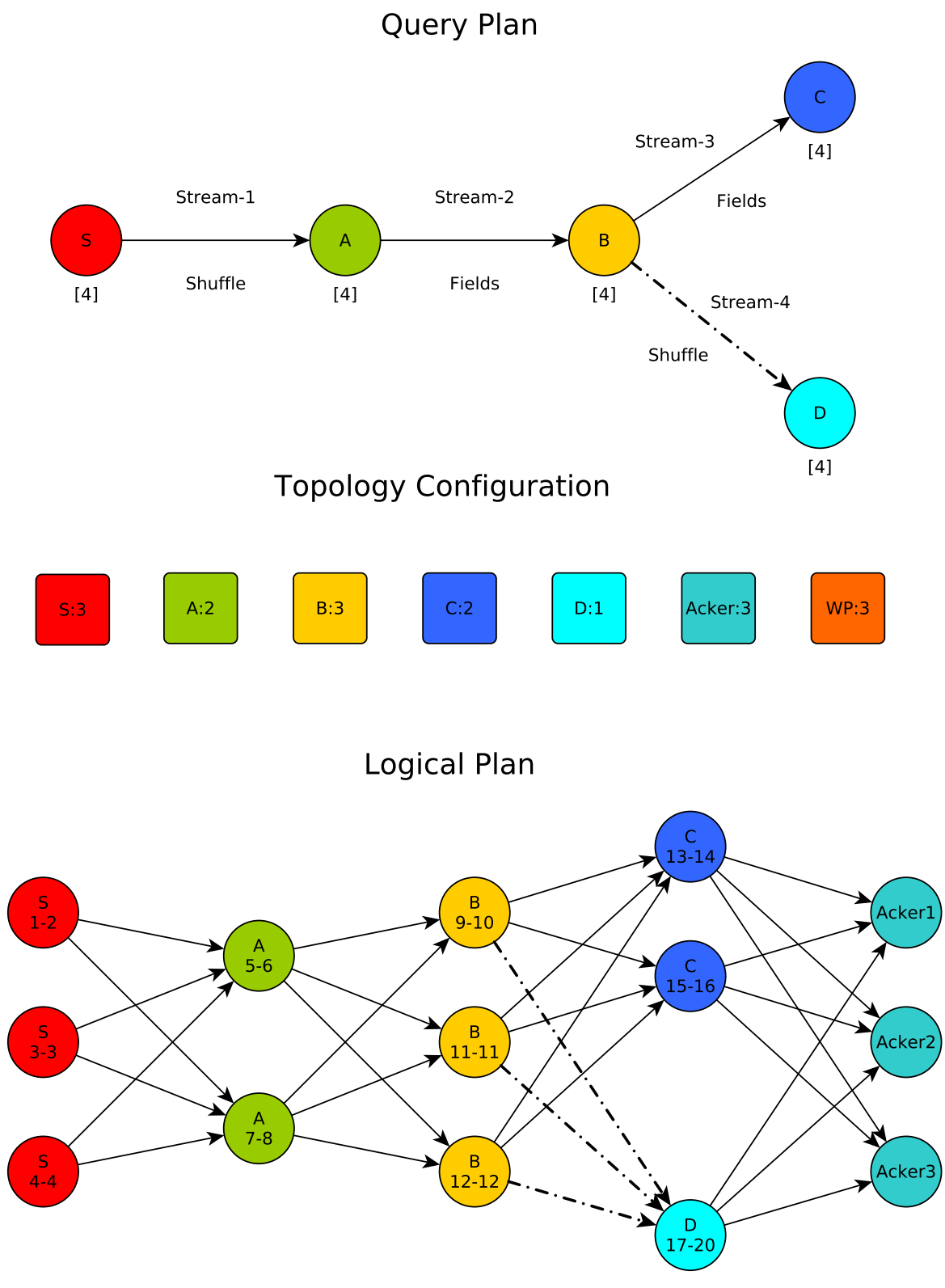

Figure 4.3: Example topology featuring a component with multiple output streams.

Fields grouping

The SRP of fields grouped connections (see section 2.5) are dependent on the partitioning function used to assign tuples to downstream tasks (see section 2.4.2) and the distribution of field (key) values within the tuple stream.

For the lifetime of a topology, a given field value (or combination of field values) will always[2] result in the same task ID being returned by the fields grouping hash function. However, that task may be assigned to a new executor during a topology *rebalance* (see section 2.11). If a proposed physical plan changes the task distribution within the source and/or destination executors of a given fields grouped logical connection, then this will effect the SRP of that connection. This is why it is important to track the proportion of tuples sent from a source task to a destination task (on a per stream basis) whilst the topology is running. To do this we define a task level SRP called the *task to task routing probability (TRP)* ($R^S_{t,u}$), which is defined as the proportion of tuples leaving task t on stream S which will be routed to task u. Where $0 \leq R^S_{t,u} \leq 1$ and for each output stream S from the source task t to the set U_t of all destination tasks downstream of t:

$$\sum_{u \in U_t} R^S_{t,u} = 1$$

In order to calculate the TRPs, the per stream task to task transfer counts $\sigma^S_{t,u}$ are required. The TRP from task t to task u is then given by dividing the transfer count from t to u on stream S by the total output count of task t onto stream S (σ^S_t):

$$R^S_{t,u} = \frac{\sigma^S_{t,u}}{\sigma^S_t} \tag{4.2}$$

The task to task transfer counts that are required to calculate the TRPs are not tracked by default in Storm. The metrics system will record, for each downstream task, the number of tuples it receives. However, it will only log the incoming stream name and source component for those tuples. The reason for this is to reduce the memory overhead of the metrics system. Using the current Storm approach, each task instance only needs to store counts equal to the number of upstream components times the number of streams between those upstream components and the one that task instance is part of. For example, in figure 4.3, task 13 (part of component C) will contain one count for tuples arriving from component B on Stream-3.

If upstream task identifiers were added to this it would increase the key space by a factor proportional to the number of upstream tasks. For task 13, in figure 4.3, recording

[2]Unless a custom fields grouping is used.

transfers at the task level would mean it would go from storing one transfer count to four (component B task 9 Stream-3, component B task 10 Stream-3, etc.). However, if the upstream component had hundreds of tasks, recording metrics at the task level could conceivably increase the memory footprint of the stored count metrics by several orders of magnitude.

Despite the memory implications, the task to task transfer count information is vital to the routing probability predictions, described in section 4.5.1. It would be possible to alter the Storm metrics system to record this information. Even with an order of magnitude increase in the stored metrics values, only integers are being stored and so the actual effect on the memory footprint of the worker process is unlikely to be significant for all but the most large scale topologies. However, instead of altering Storm itself our modelling system provides a custom metric, as part of the metrics gathering system described in section C.2, that can be added to a topology to be able to track the task to task transfer counts.

4.4.2 Global routing probabilities

The GRP ($G_{i,j}$) describes the probability that a tuple leaving executor i will be routed to executor j, from the set J_i of all executors downstream of i, regardless of the stream that the tuple is emitted on.

$$0 \leq G_{i,j} \leq 1$$

$$\sum_{j \in J_i} G_{i,j} = 1$$

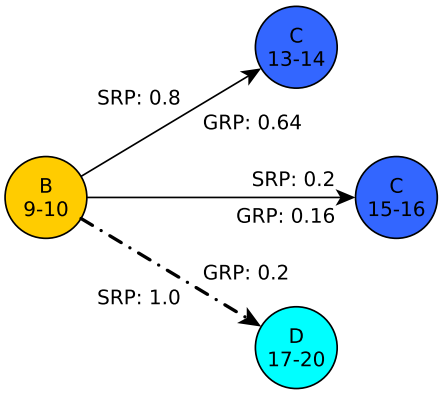

Figure 4.4: Example of the SRP and GRP values for an executor from the topology shown in figure 4.3.

Using the topology shown in figure 4.3 as an example, component B has two outgoing streams (Stream-3 and Stream-4). As there is only one executor for component D the SRP

of logical connections from the executors of component B onto Stream-4 will be 1.0. Stream-3 is fields grouped and so the SRP for logical connections from the executors of component B on Stream-3 will depend on the key distribution in the tuple stream. For example, logical connection $B_{9-10} \rightarrow C_{13-14}$ could have an SRP of 0.8 whilst $B_{9-10} \rightarrow C_{15-16}$ would have an SRP 0.2.

The GRP for the logical connections is dependent on the relative proportion of the total tuple output sent down each stream. The stream output proportion (SOP) (O_i^S) describes what proportion of the total output of executor i is emitted to output stream S. For example, if 80% of tuples from executor B_{9-10} are issued onto Stream-3 ($O_{B_{9-10}}^{\text{Stream-3}} = 0.8$) this 80% would be shared according to the SRP of each logical connection leaving B_{9-10}. Figure 4.4 illustrates the SRP and GRP of each outgoing logical connection for executor B_{9-10} from the logical plan shown in figure 4.3.

4.5 Predicting Routing Probabilities

4.5.1 Predicting stream routing probabilities

When a new physical plan is proposed, the expected SRPs for all the logical connections between executors in the resulting logical plan need to be estimated.

Shuffle grouping

For any shuffle grouped streams, it is assumed that the tuples submitted to that stream will be shared equally between all the destination executors of each component subscribed to that stream. Therefore, the SRPs for the proposed connections between a source executor i and each of its downstream destination executors (j) from the set of all possible downstream executors J_i, on shuffle grouped stream S, is defined as:

$$R_{i,j}^S = \frac{1}{|J_i|} \tag{4.3}$$

Fields grouping

Predicting the SRPs for the logical connections of fields grouped streams is a more complicated process than for shuffle grouped connections. It is likely that this complexity is the reason why the vast majority of the previous studies detailed in chapter 3 ignored topologies with these types of connections.

There are two distinct situations that occur for fields grouped connections: the case where the incoming streams into the source component of a fields grouped connection

are exclusively shuffle grouped; and the case where at least one incoming stream is fields grouped. The reason it is important to differentiate between these cases is because of how they affect the assumptions regarding tuple routing to the tasks of the downstream executors.

As described in section 2.4.2, the tasks act to partition the state space and facilitate tuple routing. The number of tasks assigned to each component is set for the lifetime of the topology and so will be shared evenly between the number of executors assigned to each component in the topology configuration. For fields grouped streams, a number of tuple fields (keys) are used to route tuples with the same fields values to the same downstream task.

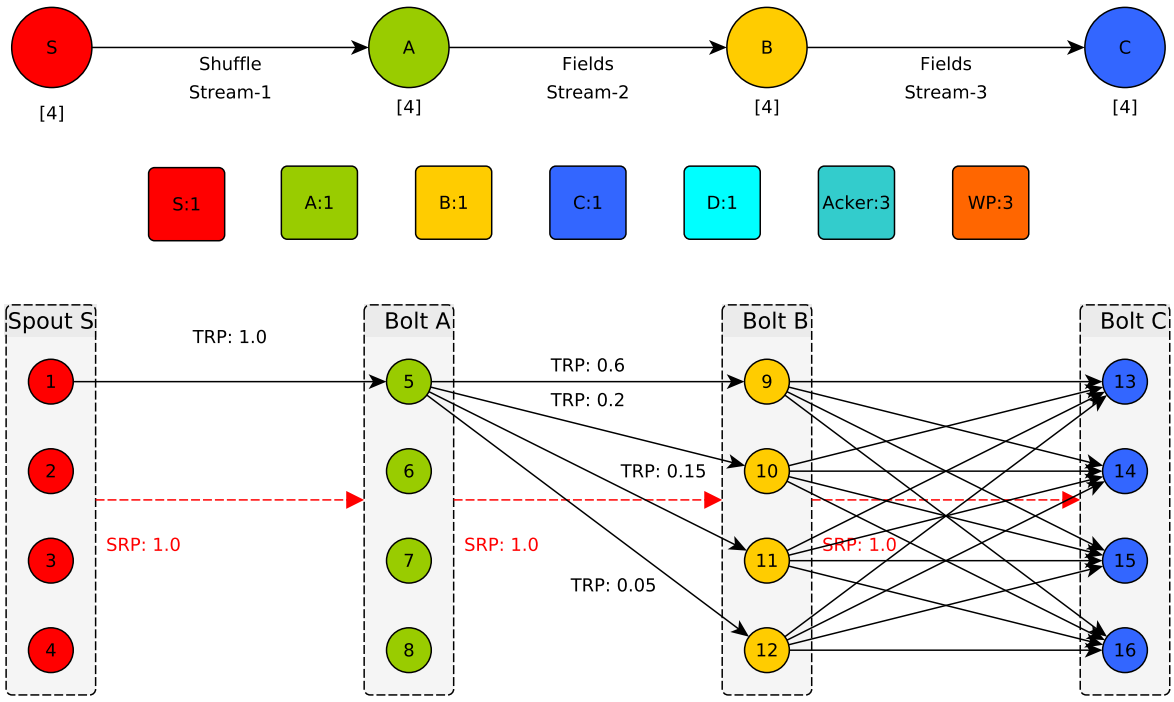

Figure 4.5: Example path from the topology shown in figure 4.3 showing the two distinct fields grouping cases: shuffle only input and fields grouping input.

Figure 4.5 shows one of the two paths through the query plan of the example topology shown in figure 4.3. This particular path ($S \rightarrow A \rightarrow B \rightarrow C$) illustrates both the cases mentioned above with the fields grouped Stream-2 having only shuffle grouped connections into its source and the fields grouped Stream-3 having a fields grouped connections into its source. The numbered circles, in figure 4.5, represent the individual tasks within the executors. The red dotted lines indicate the logical connections and the black indicate the task to task connections. In figure 4.5, the topology configuration sets the number of executors for each component to one and therefore all the tasks for each component are allocated to the single executor for that component.

Figure 4.6 shows the same query path as figure 4.5 after a rebalance, where each component

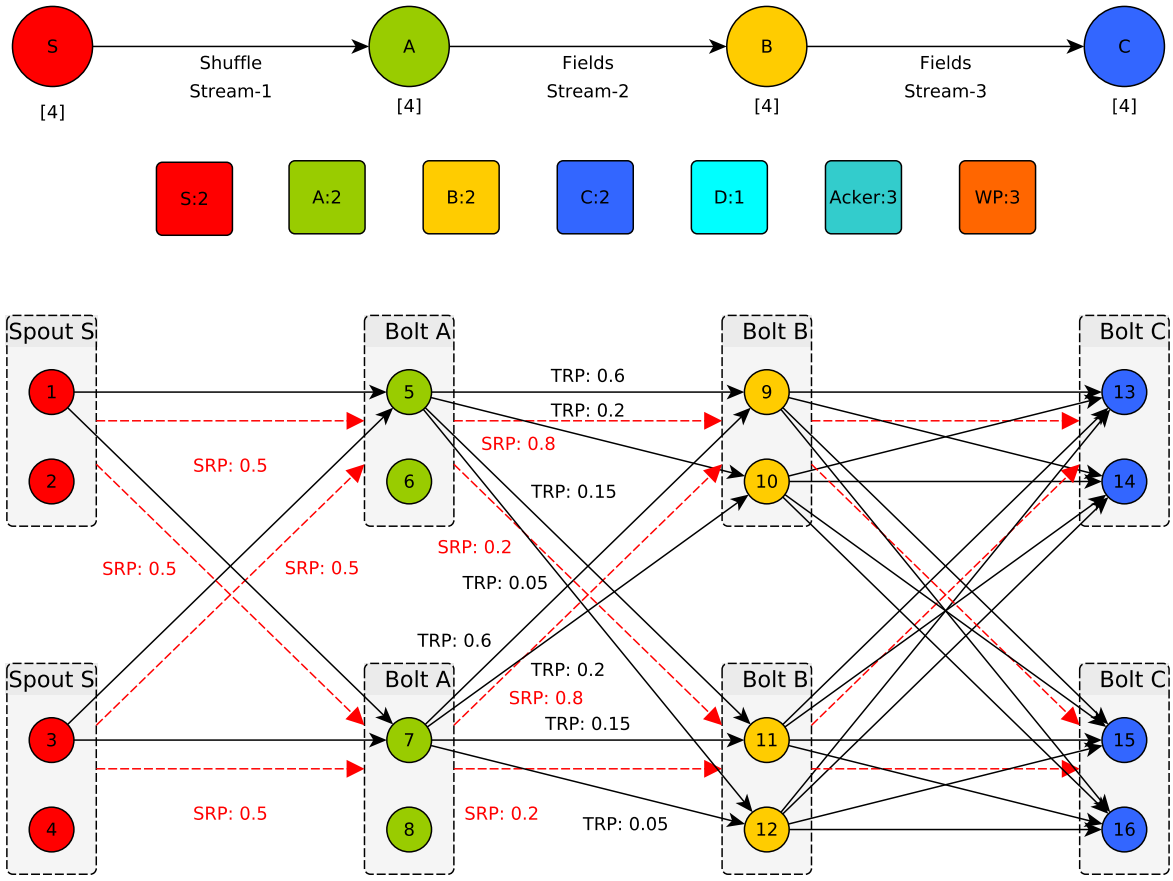

Figure 4.6: Example topology shown in figure 4.5 but with a new topology configuration where each component now has two executors instead of one.

now has double the number of executors. This shows the tasks for each component reassigned to the new executors and the resulting changes in the logical connections and their SRPs.

Both figures show that only one task in executors S_{1-4} and A_{5-6} is connected. This is because the stream connecting these components is shuffle grouped. In this case there is no state to partition as the routing is random and therefore Storm only ever activates one task in the source executors (typically the task with the lowest numbered task identifier).

Source component with shuffle grouped input streams If the source component of a fields grouped stream has only shuffle grouped inbound streams, such as component A in figures 4.5, 4.6, then executors of that source component will all receive a balanced share of the tuples from the upstream executors. This implies, assuming that the shuffling algorithm is sufficiently random, that all executors will receive approximately the same distribution of tuples with given field (key) values. For example, if we assume that the tuples leaving spout S have an *animal* field and that 10% of them have the value *"aardvark"*, then we would expect both executors of component A in figure 4.6 to see 10% of tuples arriving with "aardvark" as the value of their animal field.

As all executors receive the same distribution of field values and the hashing function that maps from field value to task ID is the same for all executors, the pattern of TRPs to all downstream tasks will be the same for all source executors which only receive shuffle grouped streams. This can be seen in figure 4.6, where the TRPs and the logical connection SRP (executor to executor) from the two executors of component A to the executors of component B are the same.

This simplifies the prediction of the SRPs for the logical connections in this case. Take for example the case where we wish to predict the SRPs between the executors of components A and B in figure 4.6 (proposed plan) using data from the topology as configured in figure 4.5 (source plan). To do this we perform the following steps:

1. Calculate the TRPs, for each output stream, using metrics from the source plan.
2. When you have shuffle only input streams, only one task is active in each executor. Therefore, the TRPs to all downstream tasks in the fields grouped stream will be the same for all executors of the source component. This results in a component to task routing probability (only applicable in this specific case). For our example, using the topology configuration shown in figure 4.5, this would result in the values shown in table 4.1.
3. For each of the downstream destination executors in the proposed plan, sum the component to task routing probabilities (shown in table 4.1) for all the tasks in each proposed destination executor, for each output stream. This value is now the

estimated SRP of the logical connections from each executor of the source component to each proposed downstream executor. Table 4.2 shows these calculations for our example connections.

Table 4.1: Measured routing probabilities for connections from executors of component A to downstream tasks of component B on Stream-2.

Connection	Component to task RP
$A \rightarrow B_9$	0.60
$A \rightarrow B_{10}$	0.20
$A \rightarrow B_{11}$	0.15
$A \rightarrow B_{12}$	0.05

Table 4.2: Predicted SRPs for all executors of component A to the proposed downstream executors of component B on Stream-2.

Connection	Predicted SRP
$A \rightarrow B_{9-10}$	0.60 + 0.20 = 0.8
$A \rightarrow B_{11-12}$	0.15 + 0.05 = 0.2

Source component with at least one incoming fields grouped stream The second case, and the more complex of the two, is where the source component for a fields grouped stream has one or more incoming streams that are themselves fields grouped. Referring to the example topology shown in figure 4.5, component B which emits onto Stream-3 is an example of this situation.

If a source component receives a fields grouped stream then the executors of that source component cannot be assumed to receive the same distribution of field values as each other. Looking at the topology in figure 4.5, if tuples on Stream-2 contain a *colour* field and 10% of those tuples have a colour value of "purple", then **all** of those tuples will be routed to one of the executors (and specifically one of the tasks) of component B. As only one task receives "purple" tuples and the others do not, its internal logic may results in tuples with different field values (compared to other tasks) being issued onto the fields grouped Stream-3. As a result of this, each individual task instance (in component B) may have different routing probabilities to all downstream tasks of component C. Therefore, combining tasks in new executors arrangements, for a proposed physical plan, will result in completely different SRPs for the logical connections between the executors. This is in contrast to the case described in the section above, where each source executor had the same predicted SRPs to the set of downstream executors.

The variation in TRPs means there is no uniform component to task routing probability that can be applied across the proposed executors. Instead, the combination of TRPs for the tasks in each set of proposed source and destination executors needs to be used to predict the SRP of the logical connection between them. For example, for the proposed logical connection between executors B_{9-10} and C_{13-14}, shown in figure 4.6, we would need the TRPs (measured from the source plan shown in figure 4.5) for tasks $B_9 \rightarrow (C_{13}, C_{14})$ and $B_{10} \rightarrow (C_{13}, C_{14})$. However, it is not simply a case of taking the average TRP for the proposed logical connection.

For components which receive fields grouped streams, such as B in figure 4.5, each task of that component receives a different level of input depending on the distribution of field values incoming into the executors of the upstream component (e.g. A). Therefore, it is fair to assume that the level of *output* from each of these tasks onto the outgoing fields grouped stream (e.g. Stream-3) will also vary. This means that the relative output of each task, on to a given output stream, should be taken into account when predicting the resulting SRP of the logical connections of that stream. If we did not take the relative output of each task into account, then we could have a situation where a task that almost never emits tuples, but has a TRP of 0.99 to a given destination task, could heavily weight the resulting SRPs of the source executor towards the executor containing that destination task. In which case this would ignore the contribution of other heavily emitting tasks, with lower TRPs, which emit to tasks on other executors.

In order to weight the contribution of the TRP of each task to the predicted SRP, we calculate the *measured task output proportion (MTOP)* (ζ_t^S). This value represents the proportion of the total output (σ_c^S) of a given source component c, onto a given output stream S, that task t emits:

$$\zeta_t^S = \frac{\sigma_t^S}{\sigma_c^S} \tag{4.4}$$

Where σ_t^S is the measured total emission onto stream S of task t in the source component of the fields grouped stream. The MTOP is calculated separately for every output stream of the source component. This is important as a given field value in an input tuple could mean many tuples being emitted onto one output stream but significantly fewer onto another, affecting the TRPs. The source plan, shown on the left hand side of figure 4.7, gives an example of the MTOP values for the tasks of component B, where task 9 emits 70% of all the tuples emitted by the executors of component B onto Stream-3.

Once the MTOP is calculated for the tasks of a source component (for a fields grouped stream), using data from the source plan, the executor task arrangement for the proposed

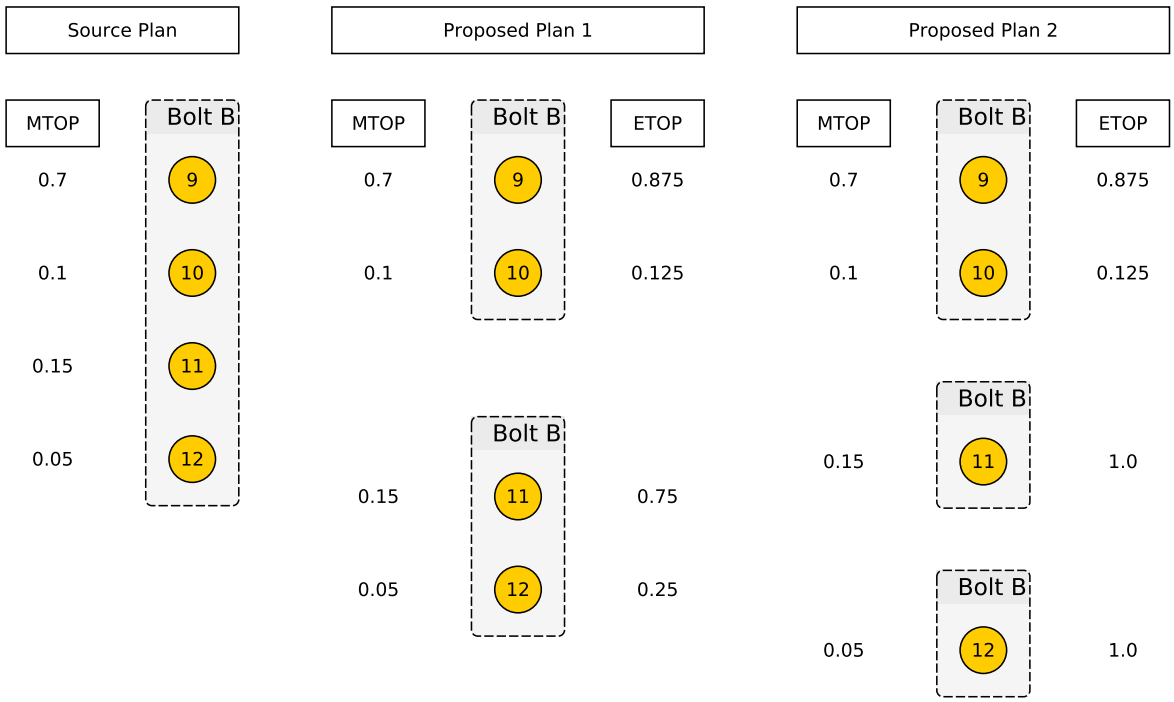

Figure 4.7: The ETOP values for two proposed plans (centre and right) using MTOP values from a single source plan (left).

plan is used to calculate the *estimated task output proportion (ETOP)* for the tasks in each proposed executor. The ETOP is the expected proportion of the total output (on a given output stream) from the executor that each task will emit. ETOP values ($\xi_t^{i,S}$) for each of the tasks t in proposed executor i, emitting on to stream S, are calculated by taking the MTOP value (ζ_t^S) of each task t and dividing it by the sum of the MTOP values over the set E_i of all tasks in the proposed executor i.

$$\xi_t^{i,S} = \frac{\zeta_t^S}{\sum_{u \in E_i} \zeta_u^S} \tag{4.5}$$

An illustration of these values in two different proposed topology configurations, using data from a single source plan, is shown in figure 4.7. Once the ETOP has been calculated for each task in a proposed source executor, we use these values as weights to the respective TRPs.

To calculate the SRP ($R_{i,j}^S$), we take the set of tasks E_i in the source executor i and the set of tasks F_j in the destination executors j and calculate a weighted average of the TRPs between the two:

$$R_{i,j}^S = \sum_{t \in E_i} \xi_t^{i,S} \left(\sum_{u \in F_j} R_{t,u}^S \right) \tag{4.6}$$

By way of an example, the stages of the process described above are carried out below for predicting the SRPs of the logical connections of Stream-3 in the topology configuration shown in figure 4.6 (proposed plan) using the data from the topology configuration shown in figure 4.5 (source plan):

1. Calculate the TRPs (equation 4.2) using measured data from the source plan. These are shown in table 4.3.

2. Calculate the MTOP (equation 4.4) for the tasks in the source plan. These will be as shown in the source plan section (left most) of figure 4.7.

3. Using the proposed executor task arrangement, from the proposed physical plan, calculate the ETOP (equation 4.5) for the tasks in the proposed executors. These will be as shown in Proposed Plan 1 (center) in figure 4.7.

4. Weight the TRP of each task in the proposed (logical connection) source executors by their respective ETOP values. Then sum these weighted TRPs for tasks in the proposed source and destination executors in order to predict the SRP (equation 4.6) for the proposed logical connection between those executors. These calculated values are shown in table 4.4.

Table 4.3: TRPs for the tasks of component B to the downstream tasks of component C on Stream-3.

Task	C_{13}	C_{14}	C_{15}	C_{16}
B_9	0.60	0.15	0.05	0.20
B_{10}	0.25	0.05	0.40	0.30
B_{11}	0.15	0.65	0.05	0.15
B_{12}	0.05	0.35	0.20	0.40

Table 4.4: Predicted SRPs for the logical connections between executors of component B to the downstream executors of component C on Stream-3.

Logical Connection	Predicted SRP
$B_{9-10} \rightarrow C_{13-14}$	$0.875(0.60 + 0.15) + 0.125(0.25 + 0.05) = 0.69$
$B_{9-10} \rightarrow C_{15-16}$	$0.875(0.05 + 0.20) + 0.125(0.40 + 0.30) = 0.31$
$B_{11-12} \rightarrow C_{13-14}$	$0.75(0.15 + 0.65) + 0.25(0.05 + 0.35) = 0.70$
$B_{11-12} \rightarrow C_{15-16}$	$0.75(0.05 + 0.15) + 0.25(0.20 + 0.40) = 0.30$

Field value distribution In the methods described above it is important to note that we assume that the distribution of field values within the tuple stream will be the same for the proposed plan as it was for the source plan. This is not an unreasonable working assumption, however if the field value distribution is time varying or dependent on some other external factors, then the predictions of the proposed plan SRPs will be affected.

To counter this a system could be developed to predict field value distribution changes over time and use this to predict a routing probability distribution for each connection. Such a system would, however, be quite intrusive and would require sampling the field value distribution for each fields grouped connection (if not all connections) in the topology. This could introduce significant processing overhead but may be worth the effort if higher accuracy is required. See section 6.3.3 for further discussion of this issue.

4.5.2 Predicting global routing probabilities

To calculate the GRPs we weight each logical connection's SRP by the proportion that connection's stream contributes to the total output of each source executor.

This process is reasonably straightforward once you have predicted the SRP ($R^S_{i,j}$) for each logical connection between source executor i and destination executor j on stream S. You then need to calculate the SOP (O^S_i) (discussed in section 4.4.2) for each output stream (S) from source executor i, which is simply the fraction of the total tuple output each stream contributes. The GRP between executors i and j is then calculated by multiplying the SRP of the logical connection between i and j by the corresponding SOP for that connection's stream:

$$G^j_i = R^S_{i,j} O^S_i \tag{4.7}$$

4.6 Input to Output Ratios

Another property that affects the flow of tuples through a topology is the input to output (I/O) ratio (ϕ) of the executors in the logical plan.

For simple components which have a single input and output stream, such as those shown in figure 4.5, the calculation of ϕ is a straightforward ratio of the arrival rate λ and the output rate σ.

$$\phi = \frac{\sigma}{\lambda}$$

The user defined code within a component may generate multiple output tuples for each input tuple, for example splitting a sentence into individual words. In this case $\phi > 1$. Conversely, a component may aggregate incoming tuples, only producing an output once a sufficient number have arrived, such as producing an average of the last ten temperature measurements. This leads to $\phi < 1$.

The expected output rate from a given executor i onto output stream S_{out} is then given by:

$$\sigma_i^{S_{\text{out}}} = \phi_i^{S_{\text{out}}} \lambda_i^{S_{\text{in}}} \tag{4.8}$$

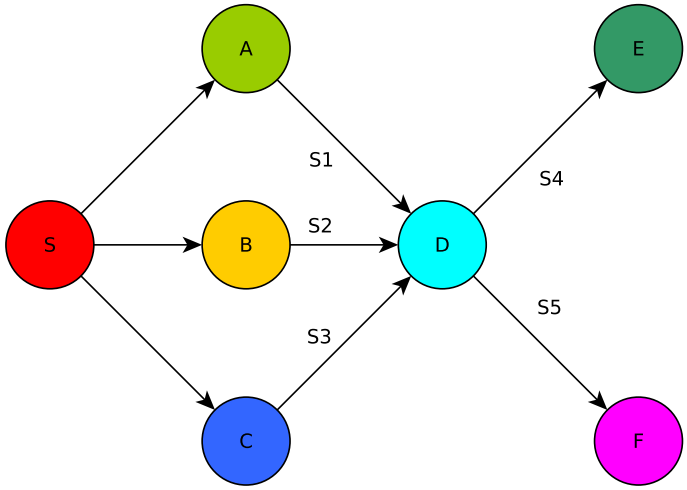

Figure 4.8: An example topology query plan showing component (D) with multiple input and output streams.

However, it is possible for components to have multiple input and output streams. Component D in the query plan shown in figure 4.8 is an example of such a situation. In this case the relationship between the output and input rates on each stream is potentially more complicated. To capture this more complex behaviour we treat ϕ not as a scalar, as in equation 4.8, but as a vector of *I/O coefficients* (β^S), one for each arrival rate λ^S of each input stream:

$$\sigma_i^{S_{\text{out}}} = \begin{bmatrix} \beta_i^1 \\ \beta_i^2 \\ \vdots \\ \beta_i^S \end{bmatrix} \bullet \begin{bmatrix} \lambda_i^1 \\ \lambda_i^2 \\ \vdots \\ \lambda_i^S \end{bmatrix} \tag{4.9}$$

As an example, taking an executor i of component D in figure 4.8, with example values

$\boldsymbol{\phi} = [1, 2, 3]$ and $\boldsymbol{\lambda} = [12, 24, 35]$, the output rate σ_i^{S4} on stream $S4$ can be calculated as below:

$$\sigma_i^{S4} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \bullet \begin{bmatrix} 12 \\ 26 \\ 35 \end{bmatrix} = 169$$

4.6.1 Calculating input to output coefficients for source physical plans

We calculate the vector of I/O coefficients ($\boldsymbol{\phi}$), for each output stream of each task in a running source topology, by using a linear regression approach. We calculate $\boldsymbol{\phi}$ at the task level, instead of the executor level, as this is needed for the prediction process described in section 4.7.

First, a time series is created from the metrics of the running source topology. This consists of output stream tuple output rates (σ_t^S) and input stream tuple arrival rates (λ_t^S) for each task t windowed into equal length time buckets. This forms a set of linear equations which are then solved using a least squares approach[3] to the resulting linear matrix equation.

This approach works well for simple topologies where the relationship between inputs and outputs is relatively straightforward.

4.7 Predicting Input to Output Ratios

In a proposed physical plan, changes in the parallelism of the components can lead to new task assignments within the executors of the resulting logical plan. As in the discussion of predicting routing probabilities (see section 4.5), the overall I/O behaviour of an executor is a function of the I/O behaviour of the tasks it contains. It is possible that the user defined code in the tasks may produce different output rates when receiving different combinations of arrival rates on the component's input streams. Therefore, the stream grouping of the input streams into a component affects how the I/O behaviour of each task contributes to the behaviour of a proposed executor as a whole.

4.7.1 Input streams containing only shuffle groupings

For a component which has only shuffle grouped incoming streams, the prediction of the I/O coefficients for the proposed executors of that component is relatively straightforward.

[3]Currently this uses the linear algebra least squares method of the numpy Python library.

The shuffle grouping of the incoming streams means that all proposed executors of the component will receive the same field value distributions as each other. Therefore, it is reasonable to take an average of the measured I/O coefficients across all tasks in the component from the source physical plan, and apply this averaged I/O coefficient vector to the proposed executors.

For example, assume that component D in figure 4.8 has only shuffle grouped inputs (streams S1, S2 and S3) and that in the source physical plan it has a parallelism of two. Now assume that a proposed physical plan increases the parallelism to three. We calculate the I/O coefficient for each task in the two executors of the source physical plan and then sum the β^S of each input stream and divide by two. This gives an average I/O ratio vector which would be used for each of the three proposed executors.

4.7.2 Input streams containing at least one fields grouping

For cases where a component has at least one incoming stream which is fields grouped, we can no longer assume that all executors of a given component will share the same I/O behaviour. It may be that tasks have logic that will emit different output rates if they receive certain fields values. This means that we need to combine the I/O behaviour of the tasks in the proposed executor.

As with the routing probability calculations for these types of components (see section 4.5.1), the I/O coefficients need to be weighted by the proportion of *activity* each task will be responsible for within the proposed executor.

To do this we make use of a similar approach to that used for weighting the TRPs. Those calculations used the ETOP, which is based on the output of each task, to weight the contribution of each task to the overall SRP of the proposed executor. However, in the case of the I/O ratios, the contributions of tasks to the I/O behaviour of their host executor is proportional to the relative level of *arrivals* they receive.

This focus on arrivals means that a new measure of task activity is needed. The measured task input proportion (MTIP) (κ_t^S) is similar to the MTOP defined in equation 4.4. However, instead of the emission rate (σ_t^S) of task t onto output stream S, the MTIP uses the task arrival rate λ_t^S on input stream S and the total arrival rate λ_c^S into component C:

$$\kappa_t^S = \frac{\lambda_t^S}{\lambda_c^S} \tag{4.10}$$

The MTIP is calculated for every input stream into every task of each component in the source physical plan. We then calculate the estimated task input proportion (ETIP) ($\eta_t^{i,S}$)

of each input stream S into task t in the proposed executors i of the proposed physical plan. This is similar to the ETOP value calculated in equation 4.5 in section 4.5.1. However, it is based on the MTIP value (κ_t^S) of each input stream S and task t divided by the sum of the MTIP values for input stream S over all tasks in the set E_i in the proposed executor i.

$$\eta_t^{i,S} = \frac{\kappa_t^S}{\sum_{u \in E_i} \kappa_u^S} \tag{4.11}$$

Once we have an ETIP value for each input stream into each task, in each proposed executor, we multiply it by the I/O coefficient β_t^S for the corresponding task and stream. We then sum these values for each stream, over all E_i tasks in the proposed executor i, to gain a set of estimated I/O coefficients for the executor.

$$\beta_i^S = \sum_{t \in E_i} \eta_t^{i,S} \beta_t^S$$

4.8 Arrival Rates

4.8.1 Predicting executor arrival rates

In order to predict the arrival rate at each executor of a proposed physical plan, the flow of tuples through that plan needs to be calculated. Once we have predicted values for the SRPs and I/O coefficients of each proposed executor, we can make a prediction of the emission rate from the topology's spouts and propagate this through the proposed plan.

Starting with the set of all spout executors, we perform a breadth-first traversal through the logical connections of the proposed tuple flow plan, taking the emission rate from each spout executor on each output stream and propagating this to each downstream executor using the routing probabilities to calculate the portion of the emission rate assigned to each connection.

Once all executors downstream of the spouts are processed, we will have the arrival rate at each of those executors. At this point we calculate the emission rate on each output stream of the downstream executors using the predicted I/O coefficients. These steps are then repeated for the next set of downstream executors (in the breadth-first traversal) until all arrival rates for the executors in the tuple flow plan have been calculated.

The above calculation will result in a per input stream (S) arrival rate (λ_j^S) at each proposed destination executor j. A check must be performed at this point to ensure that the combined arrival rate of all input streams at each proposed executor does not exceed the service rate of that executor (see section 4.9). If it does, then this proposed plan

should be rejected and the executors where the overload occurs highlighted.

4.8.2 Predicting worker process arrival rates

Once the executor arrival rates have been predicted, the proposed tuple flow plan (see section 4.1.2) is used to predict the combined arrival rate at the WPTQs. The arrival rate is found by combining the rates from all logical connections that pass through a worker process, namely those marked as inter-local or remote. These values are used with the input batch size estimation methods for the worker processes which are detailed in section 4.12.2.

4.9 Service Times

As described in section 2.13.2, Storm provides two different measures of the time taken to process a tuple within a bolt task: *process latency* (see section 2.13.2) and *execute latency* (see section 2.13.2). The difference between the two is that the process latency ends when a bolt acknowledges the received tuple (after it has finished processing it) and the execute latency ends when all code (including any code that runs after the tuple has been emitted) in the bolt's `execute` method has completed.

In practice, for most topologies the difference between the execute and process latency will be minimal (most bolts will not place code after the emission/acknowledgement call) (Allen et al., 2015). However, it is possible that there could be code after a tuple is emitted that updates an external database or a similar action that involves external system calls that could introduce additional latency. In this case there may be a significant difference between the two measures of service time.

The choice of service time is difficult; process latency is a more accurate measure of the delay a tuple sees as it moves through the topology, so may be more appropriate for analysis of individual paths through the topology. However, the execute latency captures the true time before an executor is ready to serve the next tuple, so is a truer representation of the executor's service rate. For this reason we use the execute latency in our modelling.

It is important to note that the execute and process latencies metrics are not point measurements. The latencies are measured at a rate set by the metric sample rate (see section 2.13.5). These measurements are then averaged (for each task) over the metric bucket period and this averaged value is reported as the latency for that task. Therefore, if the tuple randomly chosen by the metrics sampling takes a significantly long time to process, it will skew the average for that metric bucket period.

4.9.1 Executor co-location effects on service time

As described in section 2.4, the executors that run the spout and bolt code are actually two separate threads (the ULT and EST) which are hosted by a worker process which itself consists of a network thread, several other executors and the WPST, all within a JVM instance (see section 2.4.3). Because of this high number of threads co-located on a single worker process, the actual value of the process/execute latency can be misleading. It is possible that one executor could be *paused* by the JVM whilst another is running. Because the latencies reported by Storm are wall-clock times (the current time is compared to a start time[4] at the point the metric is recorded) these include any periods where an executor's ULT is paused. This means that executors, on worker processes which host a large number of co-located executors, could report *longer* process/execute latencies compared to the same executors running the same code on a more sparsely populated worker process. This has implications for predicting the effect on the expected service time of co-locating executors in a proposed topology physical plan.

4.10 Predicting Service Times

Predicting the effects of executor co-location for a proposed physical plan is a difficult proposition. The multi-threading behaviour (when and which threads are paused) can vary depending on the operating system and JVM implementation used to run Storm, as well as the interplay between them.

There are many possible modelling approaches that could be applied to the service time prediction. However, we have opted for a straightforward approach based on weighted averages of the measured service time from a running source physical plan. Discussion of possible future modelling approaches is given in section 6.3.4.

4.10.1 Weighted average service time

As with other predictions for properties of a proposed plan, the service time averages are dependent on the incoming stream groupings (see section 2.5).

Input streams containing only shuffle groupings

Components with only incoming shuffle groupings will receive a load balanced input across all of their executors. As a result, it is reasonable to take an average of the measured service times across the executors of each component and use this as the predicted service times for the proposed executors of that component.

[4] Using the Java `System.nanoTime()` function.

Input streams containing at least one fields grouping

For components with one or more fields grouped incoming streams, the executors of a given component may receive an uneven amount of input tuples depending on the tasks they contain. Therefore, an average across all executors of the component will not be appropriate as certain field values may lead to significantly different service times but may only be activated rarely, skewing the average.

To account for this, the average service time for components with incoming fields grouped connections must be weighted by the level of activation of each task that makes up their executors. For this we use the ETIP given by equation 4.11 (see section 4.7.2) to weight the contribution that each task will make to the proposed executor's service time.

4.11 Transfer Latencies

Another source of delay within a topology, which needs to be accounted for in predicting the end-to-end latency of a proposed tuple flow plan, is the time taken for tuples to transfer from one executor to another. There are three types of transfer between executors in the tuple flow plan:

Local Where tuples are transferred between executors on the same worker process. This involves direct transfer from the EST to the downstream ERQ and is assumed to be almost instantaneous.

Inter-local Where tuples are transferred between worker processes on the same worker node. In this case tuples are serialised and placed on the WPTQ where they are issued to a port on the same worker node. This involves transferring through the worker node's local network interface but involves no transfer across the network.

Remote This is the same as Inter-local transfers but for worker processes on separate worker nodes. This involves a transfer across the network between worker nodes.

As discussed in chapter 3, none of the previously published works focusing on model driven auto-scaling have taken transfer latency into account. Ignoring this additional source of latency is a major omission and ours is the first end-to-end modelling work to account for this properly.

The assumptions behind ignoring transfer latencies fall foul of several of the fallacies of distributed computing: namely that latency and transport costs are zero and that bandwidth is infinite. For small topologies of only a couple of connected components, the introduction of network latencies may not be an issue. However, for larger topologies the network latency can quickly become a significant factor.

As an example, take a topology which is formed of 6 components with 5 connections and has an end-to-end latency of 5ms when all those connections are either local or inter-local (i.e. the topology is running in several worker processes on one worker node). If an additional worker process on a separate worker node, which has a transfer latency of 0.1ms, is now added then some routes through the topology could involve all remote connections. This situation would result in a 10% increase in the end-to-end latency of those remote only paths. If those paths happen to have high routing probabilities they could have a significant effect on the average end-to-end latency of the topology. As we are hoping that our modelling system will allow the best proposed physical plan to be selected from many possible options (see section 1.3.4), this 10% increase could be the difference between a valid plan being accepted or rejected. Therefore, modelling the effect of network transfers is important.

4.11.1 Predicting transfer times

As part of the prediction of proposed physical plans the performance modelling system will predict the type of connections between the proposed executors. For local connections we assume that the latency will be zero. Details on the prediction of the latency due to the other two connection types are given below.

Inter-local connections

The main difference between local and inter-local connections is the encoding of tuples into a binary representation (serialisation) and then sending this binary package to another network port (belonging to another worker process) on the same worker node.

The time taken to serialise tuples and the corresponding de-serialisation when they are received, a process commonly referred to as *SerDes*, can be significant. Users can supply their own SerDes implementations for complex custom Java classes, which can often lead to significant delays in tuple transfers[5].

In Storm, serialisation of tuples occurs as they are transferred from the EST processing batch (see figure 2.8 in section 2.8) into the remote dispatch map to be passed to the WPTQ. This operation happens within the EST, which is outside of Storm's metrics system and therefore does not provide any timing information. Modifying the code to provide this information would also be difficult due to the multi-threaded nature of the codebase and the fact that delay due to message serialisation is a key bottleneck in DSPS performance, so adding additional computation at this critical point is not desirable.

For simple tuple payloads such as integers, doubles or short strings, the impact of SerDes on the topology's end-to-end latency is likely to be negligible. Due to this fact and the lack

[5]See: https://storm.apache.org/releases/1.2.2/Serialization.html

of metrics to allow prediction of the SerDes delay, we have decided to exclude modelling this aspect of the transfer latencies. This omission may introduce errors into the end-to-end latency predictions, however as stated above these are expected to be small for common, simple tuple payloads. Discussion of possible future work to predict SerDes delay is given in section 6.3.5.

Remote connections

Remote connections, like inter-local, include SerDes delays but also the time taken for the serialised tuples to pass across the network to a remote worker process. Whilst we are assuming that the SerDes delay will be negligible, we need to account for the network transfer latency. To do this we added a custom metrics gathering system to the Storm worker nodes which measures round trip latencies between each pair of worker nodes in the cluster. The details of the implementation of these metrics are given in section C.2.3.

From the measured round trip latencies, recorded between each pair of worker nodes, we take half the measured value as the network transfer latency between the two worker nodes. We then take the median of all measured network transfer latencies between a pair of worker nodes and use this value as the predicted network transfer latency for any proposed remote connections between executors on those worker nodes.

The median latency is used in order to exclude the effect of latency spikes in the measurements returned by the Internet Control Message Protocol (ICMP) packets. Discussion of more advanced network latency prediction methods is given in section 6.3.6.

4.12 Tuples Per Input List

While the ULT of the executors process individual tuples, the unit of exchange between an EST and a downstream ERQ are lists of tuples (see section 2.8.1). The number of tuples in each list arriving at the ERQs affects the performance of the associated executor (see section 4.2.2). Therefore, it is important to be able to predict the average number of tuples in each tuple list being submitted to the ERQs. This value is not recorded by default in Storm. Whilst it would be possible to alter the underlying Storm code to record this value and use it to make predictions about proposed physical plans, one of the aims of this research is to create a performance modelling system that can work with mainstream DSPSs in their current unmodified state. Therefore, we estimate the number of tuples (I) in the input lists, arriving at the ERQ of any given executor, using relevant metrics and parameters that Storm provides by default.

As described in section 2.8, the input tuple lists into each ERQ are produced via the local transfer function (LTF) either from a list of tuples bound for tasks on the same

worker process (local dispatch list) or from a de-serialised list which was transferred from a remote worker process (these cases are illustrated at the top of figure 2.8). Both of these situations take as input the batches of individual tuples taken off the ESQ's Ring Buffer; these are shown as the *processing batch* of the EST in figure 2.8.

In order to estimate the expected input list size ($E[I]$) for a given downstream ERQ, we need to estimate the expected input list size for all logical connections (both local and remote transfers) arriving at that ERQ. To do this we first need to estimate the processing batch size (Z) coming off the ESQs of the upstream executors.

4.12.1 Processing batch size estimation

Section 2.8 describes how the ESQ receives individual tuples from the executor tasks and how these then pass through the Disruptor queue mechanism as outlined in section 2.7. The EST continually polls the ESQ Ring Buffer and, when tuples are available, will extract the entire Ring Buffer population for processing. The extracted tuples form the processing batch (see figure 2.8) and the expected number of tuples in that batch ($E[Z]$) is a function of the tuple arrival rate (λ) into the ESQ and the *delivery interval* Y, which is the expected time interval between the production of processing batches from the ESQ:

$$E[Z] = \lambda E[Y] \tag{4.12}$$

The delivery interval is dependent on several factors. The main one is the processing speed of the EST. The Ring Buffer will not be polled until the processing batch is served completely. However, Storm provides no metrics on the EST processing and due to the design of this aspect of Storm it would be very difficult to instrument and extract the required processing rate metrics. However, as the EST is performing a simple sorting operation, it is reasonable to assume that this operation will be completed in a negligible time period and therefore that servicing of any given processing batch will occur almost instantaneously (service rate $\mu \approx \infty$). Using this simplifying, yet realistic, assumption the delivery interval will now depend on which event occurs first: the ESQ input batch reaching its maximum value (k) or the flush interval (δ) completing. Both events will trigger a fresh transfer of tuples into the Ring Buffer and thereby trigger the EST to extract the entire available tuple population into the processing batch.

The calculation of the expected delivery interval ($E[Y]$) has to weight the contribution of three main aspects: k arrivals into the input batch, the flush interval completing and the case where no arrivals occur within the flush interval (i.e. a very low arrival rate).

Full input batch

To account for the possibility of k arrivals before the flush interval (δ) completes, we need to calculate the expected time interval (X_k) until the k^{th} arrival.

$$E[X_k] = \sum_{x=0}^{\delta} xP(X_k = x) \tag{4.13}$$

If we assume that arrivals are from a Poisson process, with rate λ, then the probability that $X_k = x$ is given by the probability density function ($f_k(x)$) of the Erlang distribution (which gives the expected waiting time for k occurrences of an event):

$$P(X_k = x) = f_k(x) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!} \tag{4.14}$$

Where $x, \lambda \geq 0$. Substituting equation 4.14 into equation 4.13 (and multiplying it by x) gives:

$$E[X_k] = \int_0^{\delta} x f_k(x)\,\mathrm{d}x = \int_0^{\delta} \frac{\lambda^k x^k e^{-\lambda x}}{(k-1)!}\,\mathrm{d}x \tag{4.15}$$

We can simplify things further by noting that:

$$f_{k+1}(x) = \frac{\lambda^{k+1} x^k e^{-\lambda x}}{k!} = \frac{\lambda x}{k}\left(\frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}\right) = \frac{\lambda x}{k} f_k(x)$$

$$\therefore x f_k(x) = \frac{k}{\lambda} f_{k+1}(x) \tag{4.16}$$

Using equation 4.16, equation 4.15 becomes:

$$E[X_k] = \frac{k}{\lambda} \int_0^{\delta} f_{k+1}(x)\,\mathrm{d}x \tag{4.17}$$

The integral in equation 4.17 is equivalent to $P(X_{k+1} \leq \delta)$, which can be found using the

cumulative distribution function (CDF) of the Erlang distribution:

$$P(X_k \leq x) = 1 - \sum_{j=0}^{k-1} \frac{(\lambda x)^j}{j!} e^{-\lambda x} \tag{4.18}$$

$$\therefore P(X_{k+1} \leq \delta) = 1 - \sum_{j=0}^{k} \frac{(\lambda \delta)^j}{j!} e^{-\lambda \delta} \tag{4.19}$$

Substituting equation 4.19 into 4.17 gives:

$$E[X_k] = \frac{k}{\lambda} \left(1 - \sum_{j=0}^{k} \frac{(\lambda \delta)^j}{j!} e^{-\lambda \delta} \right) \tag{4.20}$$

Flush interval completes

To account for a flush interval (δ) completing before k tuples arrive in the input batch, we need to weight the flush interval by the probability that X_k (the interval until k tuples arrive) is greater than δ:

$$\delta P(X_k > \delta) = \delta(1 - P(X_k \leq \delta))$$

Using the Erlang CDF shown in equation 4.18 we can replace $P(X_k \leq \delta)$ in the equation above, however we need to account for the fact that no flushes can happen without a tuple being present and therefore we sum from $j = 1$ not $j = 0$ as in equation 4.18.

$$\delta[1 - P(X_k \leq \delta)] = \delta \left[1 - \left(1 - \sum_{j=1}^{k-1} \frac{(\lambda \delta)^j}{j!} e^{-\lambda \delta} \right) \right] = \delta \left(\sum_{j=1}^{k-1} \frac{(\lambda \delta)^j}{j!} e^{-\lambda \delta} \right) \tag{4.21}$$

No arrivals in the flush interval

Finally, we have to account for the fact that no arrivals may occur in the flush interval (δ). We therefore need to add δ weighted by the probability of zero arrivals in δ (P_0). However, we also need to include a further delivery interval (Y_k) as the current period has completed with no action and so a further wait period is required:

$$P_0(\delta) + P_0(E[Y]) \tag{4.22}$$

Delivery interval

Putting equations 4.20, 4.21, 4.22 together gives a formula for estimating the expected delivery interval for a given ESQ:

$$E[Y] = \frac{k}{\lambda}\left(1 - \sum_{j=0}^{k}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) + \delta\left(\sum_{j=1}^{k-1}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) + P_0(\delta) + P_0(E[Y])$$

The $P_0(\delta)$ term can be merged into the second term by changing the sum to include $j = 0$. This leaves $P_0(E[Y])$ as the third term, which we subtract from both sides to leave $E[Y] - P_0(E[Y]) = E[Y](1 - P_0)$ on the left hand side. Dividing both sides by $(1 - P_0)$ gives:

$$E[Y] = \frac{1}{(1-P_0)}\left[\frac{k}{\lambda}\left(1 - \sum_{j=0}^{k}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) + \delta\left(\sum_{j=0}^{k-1}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right)\right] \tag{4.23}$$

As we are assuming that arrivals follow a Poisson distribution, we know that the probability of k events in interval x is given by:

$$P(k \text{ in } x) = e^{-\lambda x}\frac{\lambda^k}{k!}$$

$$\therefore P(0 \text{ in } \delta) = e^{-\lambda\delta} = P_0 \tag{4.24}$$

Substituting equations 4.23, 4.24 into equation 4.12 gives us a formula for estimating the processing batch size Z:

$$E[Z] = \frac{\lambda}{(1-e^{-\lambda\delta})}\left[\frac{k}{\lambda}\left(1 - \sum_{j=0}^{k}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) + \delta\left(\sum_{j=0}^{k-1}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right)\right] \tag{4.25}$$

4.12.2 Transfer list size estimation

Now that we have a way to estimate the ESQ processing batch size (Z) for each of the ESTs on a worker process, we can estimate the size of the input lists that are transferred to the downstream ERQs. As shown in figure 2.8, after the ESQ the EST divides the processing batch into a local dispatch list, for local transfers to executors within the same

worker process, and a remote dispatch map, for transfer to executors on different worker processes. Eventually, the tuples in these two data structures will form the input tuple lists that arrive at the ERQs. However, the route these two tuple collections take and the processes they pass through are different. Figure 4.9 illustrates the two types of path to the ERQ of destination executor j. The figure shows a local transfer from executor i on the same worker process (x) as j and a remote transfer from executors k and l which are on a separate worker process (w).

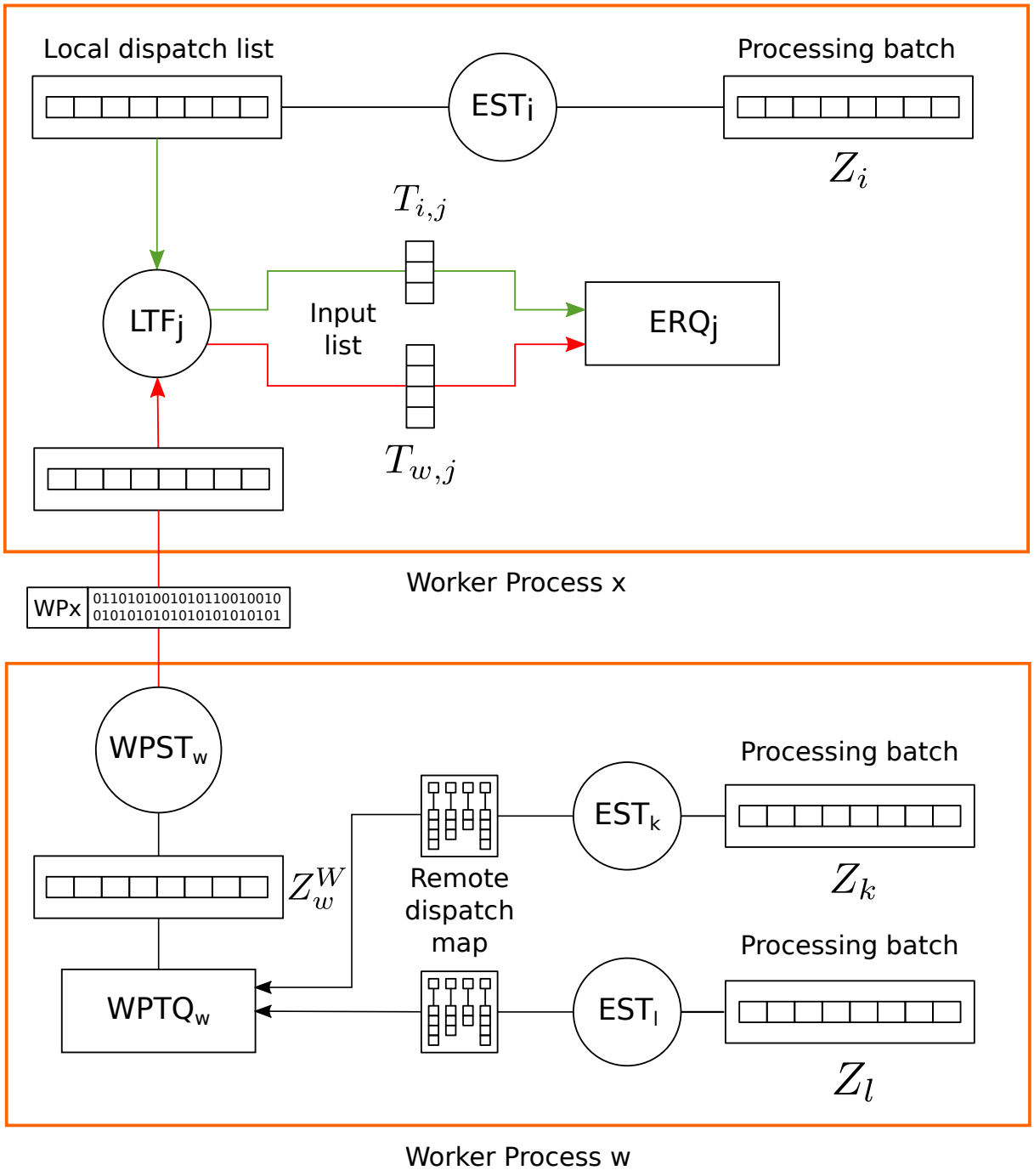

Figure 4.9: Local and remote transfer paths by which tuple lists arrive at the ERQ of executor j.

As a result of these different paths, the process used to estimate the size of the resulting

tuple lists arriving at the downstream ERQs via each path is different, but both follow a similar approach:

1. Calculate the processing batch size (Z) for each EST in the worker process.

2. Calculate the proportion of each processing batch that will be sent to each destination. We refer to these proportions as the *transfer lists*.

3. Weight these transfer lists by the activity of the sending entity (executor or worker process) and calculate a weighted average incoming list size at the receiving entity.

The specific processes used for the local and remote transfers are detailed in the sections below. To aid in the explanation of the two scenarios, a motivating example topology is shown in figure 4.10. Unlike previous example topology logical plans, figure 4.10 shows all executors being logically connected to the Acker executors. This is because all executors will emit *ack* tuples to the Ackers and these will be included in the processing batches of each EST[6]. Figure 4.11 shows a schematic of the internal transfers within worker process 1 of the example topology, along with the associated transfer lists.

Local transfers

The local dispatch list of each executor, consists of tuples bound for tasks on the same worker process (see figure 2.8). The dispatch list is then sorted by the worker process's LTF into transfer lists for each destination executors, which are then moved directly to the relevant ERQs. The number of tuples in these lists is a function of the processing batch size Z_i and the GRP ($G_{i,j}$) from the source executor i to the local downstream destination executor j.

Recall that the GRP, discussed in section 4.4, is the probability that *any* tuple emitted from executor i will be routed to executor j. By multiplying the processing batch size Z_i by each of the GRPs $G_{i,j}$ we can calculate the transfer list size $T_{i,j}$ for each local logical connection between executors.

$$T_{i,j} = Z_{i,j}G_{i,j} \tag{4.26}$$

These types of local transfers can be seen at the top of figure 4.9 (green path) and as the green arrows shown in figure 4.11, for the example topology from figure 4.10.

[6]These connections are excluded in other logical plan visualisations in order to make them easier to understand.

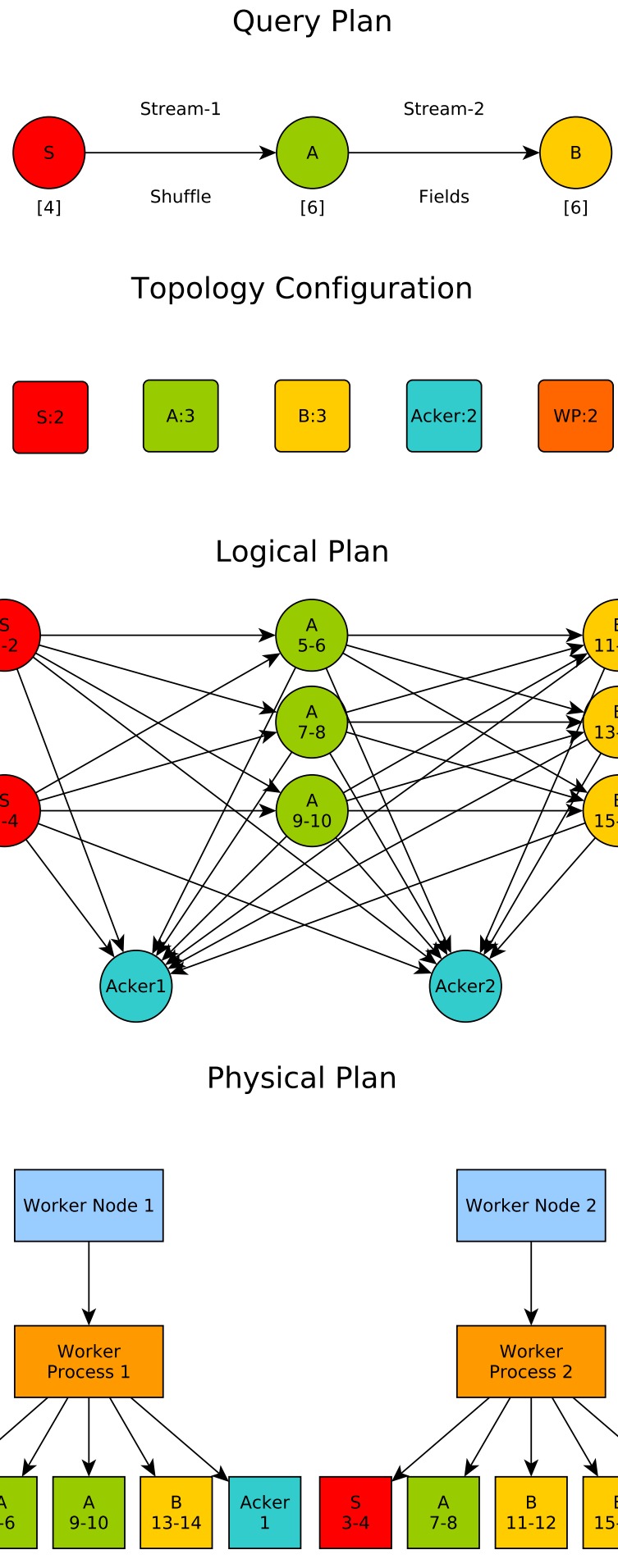

Figure 4.10: Plan types for an example linear topology runing on two worker nodes.

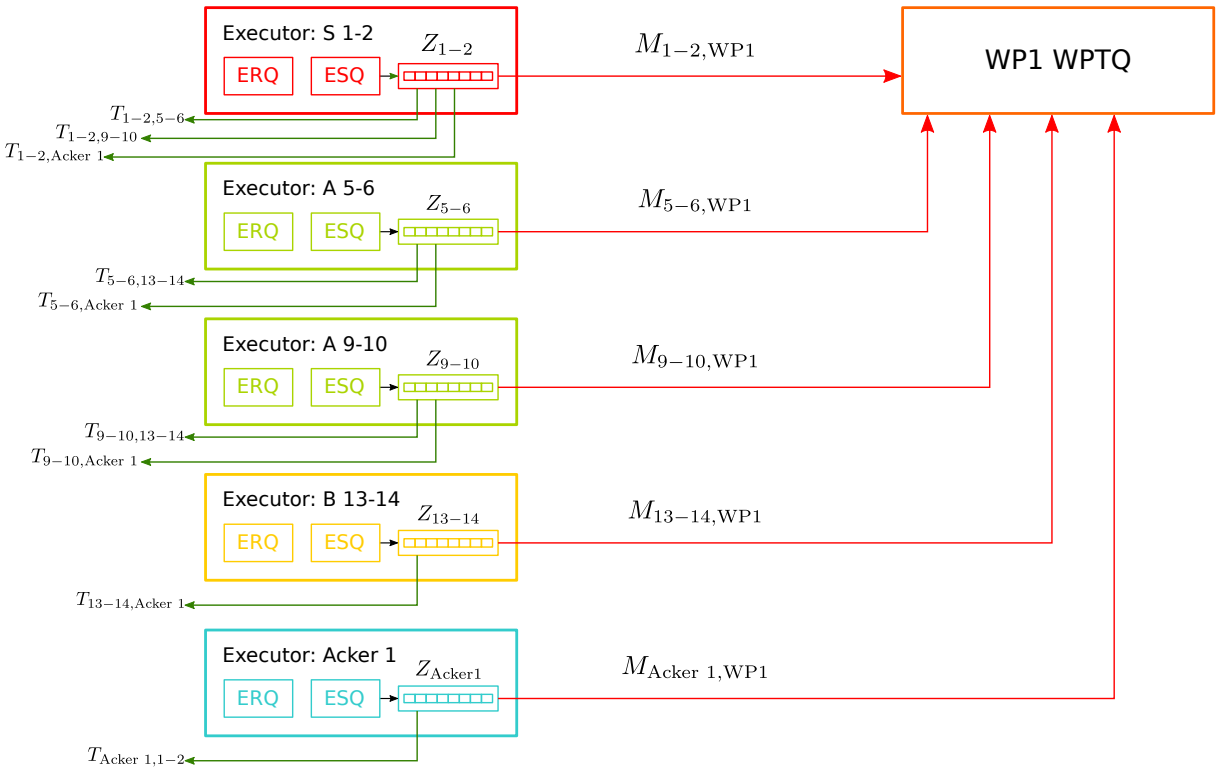

Figure 4.11: Example transfers within worker process 1 from figure 4.10.

Remote transfers

The lower half of figure 4.9 (red path) shows the route taken by tuples arriving at executor j from non-local executors on worker process w. Each remote source executor k will still send $T_{k,j}$ tuples on average, as per equation 4.26, to destination executor j. However, the EST of each source executor will combine all the tuples for remote destination executors into one remote dispatch map. These maps will then be combined as they are taken off the WPTQ, by the WPST, into lists of tuples bound for **all** executors on the receiving worker process. The receiving worker process's LTF will then sort these received lists into lists for each executor. This process means that, for remote transfers, the input tuple lists emitted by the receiving worker process's LTF can potentially contain tuples from multiple source executors. This is in contrast to local transfers which will only contain tuples from a single source executor.

The size $T_{w,j}$ of the input tuple lists arriving at destination executor j from a remote worker process w, are a function of the processing batch sizes and GRPs of all the executors on worker process w sending to j, as well as the processing batch size (Z_w^W) of w's WPST.

Because the WPTQ is a Disruptor queue, just like the ESQ, we can use a similar approach to that described in section 4.12.1, to estimate the WPTQ processing batch size (Z_w^W). However, the delivery interval prediction for the ESQ cannot be directly applied to the WPTQ. In the ESQ the input is individual tuples and in the WPTQ it is a map linking from task identifier to a list of tuples. This affects the values of the arrival rate (λ) and

the input batch size limit (k).

The arrival rate has to be converted into maps per second instead of the tuples per second (ts^{-1}) assumed for the ESQ. This requires an estimate of the average number of tuples (M_w) within a remote dispatch map arriving at the WPTQ of worker process w.

From this we can use the tuple arrival rate (λ), which is reported by the WPTQ (in ts^{-1}) and is measured as maps are placed onto the Ring Buffer, to alter the λ term in equation 4.25 appropriately:

$$\lambda_{\text{map}} = \frac{\lambda}{M_w} \tag{4.27}$$

We can estimate the average number of tuples (M_w) in a remote dispatch map arriving at the WPTQ by using the expected average size $M_{i,w}$ of remote dispatch maps sent from each source executor i on worker process w and weight them by the expected proportion of arriving maps that will have each size. $M_{i,w}$ is calculated by taking the processing batch size Z_i of the source executor i and multiplying it by the sum of GRPs from i to the set J_i^R of all remote and inter-local executors downstream of i:

$$M_{i,w} = Z_i \sum_{j \in J_i^R} G_{i,j} = \sum_{j \in J_i^R} T_{i,j} \tag{4.28}$$

An example of these values for the situation shown in figure 4.11 is given in table 4.5.

Table 4.5: Table showing the amount of tuples, on average, in each remote transfer map sent from an executor to the WPTQ in figure 4.11.

Source	Remote dispatch map size ($M_{i,w}$)
S 1-2	$M_{1-2,\text{WP1}} = Z_{1-2}(G_{1-2,7-8} + G_{1-2,\text{Acker 2}})$
A 5-6	$M_{5-6,\text{WP1}} = Z_{5-6}(G_{5-6,11-12} + G_{5-6,15-16} + G_{5-6,\text{Acker 2}})$
A 9-10	$M_{9-10,\text{WP1}} = Z_{9-10}(G_{9-10,11-12} + G_{9-10,15-16} + G_{9-10,\text{Acker 2}})$
A 13-14	$M_{13-14,\text{WP1}} = Z_{13-14}G_{13-14,\text{Acker 2}}$
Acker 1	$M_{\text{Acker 1,WP1}} = Z_{\text{Acker 1}}G_{\text{Acker 1},3-4}$

In order to estimate the expected average input map size (M_w) arriving at the WPTQ, we need to weight each of the remote dispatch map sizes ($M_{i,w}$) by the proportion of all input maps that have that size. To do this we need to evaluate the relative incoming flow of maps from each executor to the WPTQ.

Let $\sigma_{i,j}$ be the rate at which executor i emits tuples to a non-local executor $j \in J_i^R$ (across

all streams connecting the two executors). Let λ_w be the total arrival rate of tuples at the WPTQ of worker process w.

$$\lambda_w = \sum_{i \in L_w} \left(\sum_{j \in J_i^R} \sigma_{i,j} \right) \tag{4.29}$$

Where L_w is the set of all executors in worker process w. To find M_w we multiply each $M_{i,w}$ by the proportion of λ_w that the source executor is responsible for and sum over all executors in w:

$$M_w = \sum_{i \in L_w} \left(M_{i,w} \left(\frac{\sum_{j \in J_i^R} \sigma_{i,j}}{\lambda_w} \right) \right)$$

Rearranging and substituting equation 4.28 gives:

$$M_w = \frac{1}{\lambda_w} \sum_{i \in L_w} \left(\sum_{j \in J_i^R} T_{i,j} \sum_{j \in J_i^R} \sigma_{i,j} \right) \tag{4.30}$$

Both $\sigma_{i,j}$ and λ_w are calculated as part of the arrival rate prediction methods described in section 4.8. Using equation 4.30 with equation 4.27 and substituting the appropriate terms into the Disruptor queue processing batch size estimation (equation 4.25) allows us to estimate the expected worker process processing batch size (Z_w^W) in terms of maps. We then multiply this by M_w to get Z_w^W in terms of tuples.

Once we have an estimate for Z_w^W, we then need to estimate what fraction of it will be sent from w to each downstream non-local executor j. We have already calculated the transfer batch sizes from each source executor i to each destination executor j ($T_{i,j}$ - see section 4.12.2). From these we can calculate the total average transfer out of w:

$$T_w = \sum_{j \notin L_w} \left(\sum_{i \in L_w} T_{i,j} \right)$$

Using this we can find the fraction going to executor j and multiply this by Z_w^W to estimate the average number of tuples arriving at j from w:

$$T_{w,j} = Z_w^W \left(\frac{\sum_{i \in L_w} T_{i,j}}{T_w} \right) \tag{4.31}$$

In reality, all the tuples bound for the executors on a remote worker process will be grouped

together into one list and sent across the network. However, we don't need to consider this stage as when they reach their destination they will be split back into the per executor lists.

4.12.3 Input list size estimation

Once we have estimates for both the average local ($T_{i,j}$) and remote ($T_{w,j}$) transfer list sizes we can estimate the average input list size (I_j) to each destination ERQ j. However, the expected value of I_j is not just an average of all incoming tuple lists. Due to the variation in GRPs and the relative activity of the upstream executors, it is possible that a destination ERQ will receive more lists of certain sizes than others. Therefore we need to weight the average list size of each tuple list arriving at executor j by the relative proportion of the tuple list arrival rates (λ^T).

Tuple list arrival rate

Calculating the tuple list arrival rate λ^T involves first calculating the tuple arrival rate at each destination executor j of tuples sent from source executor i ($\lambda_{i,j}$). This value is found as part of the arrival rate calculations described in section 4.8. Once we have this per logical connection tuple arrival rate, the next step then depends on whether a given connection is local or remote:

Local incoming connections For these types of connections we divide the tuple arrival rate ($\lambda_{i,j}$) by the estimated local transfer list size ($T_{i,j}$). This gives the local tuple list arrival rate ($\lambda^T_{i,j}$) at each destination executor (j) for each local connection from executor i.

$$\lambda^T_{i,j} = \frac{\lambda_{i,j}}{T_{i,j}} \tag{4.32}$$

Remote incoming connections For these types of connections, the transfer list size is a function of all source executors sending from worker process w to j. Therefore, we need to sum the tuple arrival rates over the set L_w, j of all source executors i in w sending to j.

$$\lambda^T_{w,j} = \frac{\sum_{i \in L_{w,j}} \lambda_{j,i}}{T_{w,j}} \tag{4.33}$$

Expected input tuple list size

Now that we have the arrival rates for the incoming tuple lists at each destination executor j, we can weight the transfer list sizes by their relative arrival rates. The sum of these, over the set of local executors L_w and remote worker processes V, gives the weighted average input tuple list size (I_j) at executor j:

$$\lambda^T_{\text{total}} = \sum \lambda^T_{i,j} + \sum \lambda^T_{w,j}$$

$$I_j = \sum_{i \in L_w} \frac{\lambda^T_{i,j} T_{i,j}}{\lambda^T_{\text{total}}} + \sum_{w \in V} \frac{\lambda^T_{w,j} T_{w,j}}{\lambda^T_{\text{total}}} \tag{4.34}$$

4.13 End-to-end Latency

Now that we have methods to estimate all the variables outlined in section 4.1, we can estimate the end-to-end latency for any physical path through the proposed topology tuple flow plan. Using the topology shown in figure 4.10, a sample path in the topology's tuple flow plan is shown in figure 4.12. This shows all of the stages that contribute to the final end-to-end latency, including delays at the executors, worker processes and transfer latencies across the network.

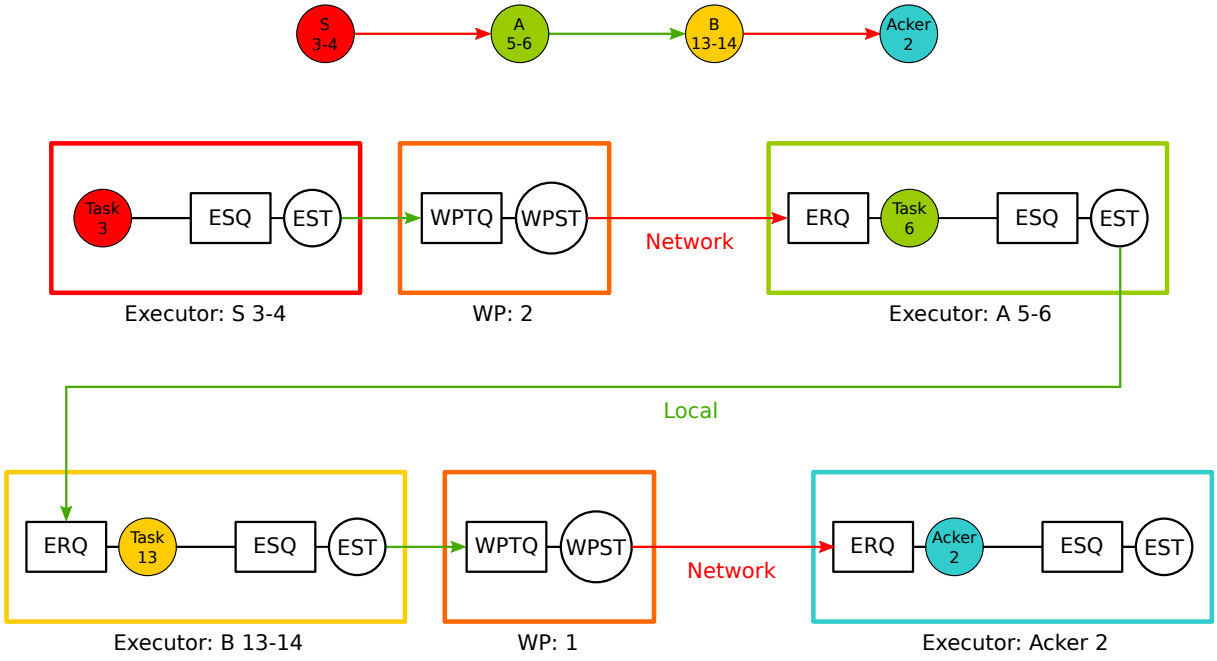

Figure 4.12: An example path through the tuple flow plan of the topology shown in figure 4.10.

Using the arrival rate, service time and input tuple list size prediction methods detailed in the sections above, we can calculate the delay due to the ULT element of the executors in

the tuple path using the method discussed in section 4.2.1. The network transfer latencies can be estimated via the method laid out in section 4.11. This then leaves the delay due to windowing behaviour within the bolt tasks, delays at the EST element of the executors and the delay due to the WPST of the worker processes to be estimated.

4.13.1 Windowing delay

As described in section 2.12, a common operation in stream processing is to window the incoming message stream to allow lower frequency results summaries to be calculated and/or to batch process incoming messages for higher throughput. This aggregation of incoming tuples will affect the I/O ratio of any executors that implement this functionality. This aspect is dealt with in section 4.6.

As well as affecting the I/O ratio of the executors, windowing also affects the latency that an incoming tuple experiences. After it has passed through the ERQ, Storm's windowing application programming interface (API) will place the tuples into an internal buffer where they will wait until their corresponding windows complete. This introduces an additional delay, before processing in the ULT, that needs to be included in the calculation of the total delay across the executor.

To account for this windowing delay, we need to know the time duration τ_W that the window covers. Depending on the configuration of the windows, τ_W may be fixed or variable. If a window's length is based on time (e.g. the window completes every 5 seconds), then τ_W will be a constant value. Alternatively, if the window's length is based on counts (e.g. the window completes when 10 tuples have arrived), then τ_W will vary per window based on the tuple arrival rate (λ).

Storm's API allows us to programmatically access the configuration of each of the topology components and the window configurations are included in this. For window configurations where the length is based on time, we can use the configured window length as τ_W. However, for count based window lengths (C_W), we still need to calculate the corresponding window duration for these values using the arrival rate.

$$\tau_W = \frac{C_W}{\lambda}$$

Tumbling windows

For the simple sequential batches of a tumbling window implementation, once we have the window duration τ_W we can estimate the likely average delay a tuple will experience whilst waiting for the window to complete. We do not know the distribution of arrivals into the window, but they are unlikely to be Poisson distributed after passing through the various

sections of the Disruptor queue (see section 2.7). However, as a first approximation we assume that tuples arrive at a uniform rate over τ_W time. This implies that on average a tuple will be delayed by $\frac{\tau_W}{2}$ time whilst the window completes.

Sliding windows

For the sliding window case the modelling is more complicated. As described in section 2.12, a given tuple can be present in several windows when a sliding window approach is used. This windowing implementation in Storm will automatically acknowledge a tuple once it has passed out of all possible windows. This more complex case is left for future research.

Window service time

In calculating the additional delay due to windowing, the time taken to process the whole window (b_W) also needs to be considered. The execute latency reported by Storm for windowed bolts relates to the time taken to add tuples to the internal window buffer(s). This is fortunate as this per tuple service time allows the ULT simulator to be applied to windowed bolt executors unmodified. The latency of the window execute method is not recorded explicitly, however the process latency reported for the windowing executor will equate to the time taken for all tuples in the input window to be acknowledged. This measure will not include any additional processing such as database connection clean up etc, but can be used as an approximation of the window service time and is used as our measure of b_W.

The prediction of b_W, for the proposed executors of a windowing component, depends on the type of window length:

Count based window lengths For this type of window configuration the window duration (τ_W) will vary depending on the arrival rate (λ). However, the number of tuples in each window will be constant. It is therefore reasonable to assume that the time taken to process the window will also be constant. Using this assumption we can use the approach described in section 4.10, for predicting the service time of the executor ULT, to predict b_W for the proposed executors of windowing components.

It should be noted that the same points raised in the discussion of predicting the non-windowed executor service times also apply to predictions of the window service time (see section 4.10).

Time based window lengths For this type of window configuration τ_W is constant. However, the number of tuples in the window will vary depending on the arrival rate. The variable number of tuples in each window mean that we cannot assume that the window service time is constant for all windows, it will instead be a function of the arrival rate.

This additional dependence on the arrival rate means that the service time prediction method described in section 4.10 will not be applicable.

Instead, a way to predict the window service time for a given time based window length and arrival rate is required. This will require creating a model which can capture the interplay between these variables. This work is left for future research and the current modelling approach will focus on count based tumbling windows only.

4.13.2 Executor send thread

As described in section 2.8, once tuples are created by the executor's tasks they are placed individually onto the ESQ. From here the EST will extract the tuples into the local dispatch lists and remote dispatch maps for routing to their destination executors. As the ESQ is a Disruptor queue, a similar approach to that used for the ERQ and ULT could be applied (see section 4.2.2). However, as discussed in section 4.12.1, by default Apache Storm does not provide any service time information for the EST operations. Adding such a metric would be exceedingly difficult as the operations involved in creating the local dispatch list and remote dispatch map are outside of the scope of Storm's metric system and involve functions from both the executor and worker process code.

As with the input tuple list size calculations described in section 4.12, we can assume that the relatively simple sorting operation that the EST performs is practically instantaneous ($\mu \approx \infty$) and therefore that the delay due to the EST element of the executor will be governed by the time it takes the ESQ input batch to either be full (k tuple arrive) or the flush interval (δ) to complete. This period is equivalent to the delivery interval (Y) given by equation 4.23. However, when predicting tuple delay as opposed to the transfer batch size, we do not need to consider the probability (P_0) of no tuples arriving as, by its very definition, tuple wait time implies that a tuple has already arrived.

Therefore, the predicted delivery interval (Y_{EST}) for the ESQ is given by combining equation 4.20 and equation 4.21:

$$E[Y_{EST}] = \frac{k}{\lambda}\left(1 - \sum_{j=0}^{k}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) + \delta\left(\sum_{j=1}^{k-1}\frac{(\lambda\delta)^j}{j!}e^{-\lambda\delta}\right) \tag{4.35}$$

However, this will not be the average tuple latency (W_{EST}) across the ESQ and EST. We assume that tuples can arrive at any time during the delivery interval. Therefore a tuple could arrive early and wait for almost all the interval or arrive just before the interval ends or at any point in between. For simplicity we assume the distribution of these arrivals within the delivery interval is uniform and therefore that the average time a tuple will

wait within the input batch of the ESQ is half this period:

$$E[W_{EST}] = \frac{E[Y_{EST}]}{2} \tag{4.36}$$

4.13.3 Worker process send thread

Similar to the EST delay described above, the delay due to the WPST also suffers from a lack of service time metrics. However, just like the EST, the WPST is performing a relatively simple sorting and routing operation and therefore it seems reasonable to assume that the processing will be almost instantaneous ($\mu \approx \infty$). This means that the delay at the WPST will be dominated by the waiting time at the WPTQ. As in the case of the EST, we can use the delivery interval Y_{WPST} of the WPTQ to approximate the sojourn time across the WPST.

However, we cannot simply use equation 4.35, as this assumes individual tuples are arriving into the queue. To use this equation we have to alter the λ term to account for the fact that remote dispatch maps are the input to the WPTQ rather than individual tuples. Using the estimated average map size M_w, found in equation 4.30, with equation 4.27 and substituting the resulting remote dispatch map arrival rate λ_{map} for the tuple arrival rate λ in equation 4.35, allows us to estimate the delivery interval Y_{WPST} for the WPTQ.

As with the EST delay discussed above, the delivery interval will not be the average tuple sojourn time (W_{WPST}). We assume that tuples can arrive at any time during the delivery interval and that the distribution of these arrivals is uniform. Therefore, the average time a tuple will wait within the input batch of the WPTQ is half the estimated delivery interval.

4.13.4 Worker process receiving logic

When the lists of tuples from other worker processes arrive at a port on a worker node they are transferred via the network client to the worker process where they are sorted by the LTF into lists for each destination executor. This process will introduce some additional delay for tuples passing through it. However, Storm does not provide any metrics for the performance of the network client or the LTF. As with the sorting functions of the EST and the WPST, we have assumed that the worker process receiving logic is essentially instantaneous and that any latency it adds to a tuple's end-to-end latency is negligible.

4.13.5 Predicting complete latency

Once we have methods for predicting the delays for the various elements of a given physical path through a topology's tuple flow plan, we can estimate the average end-to-end latency of that path. However, the physical path end-to-end latency cannot be validated on its own as Storm does not provide a comparable metric for this measure. Section 2.13.2 describes how the complete latency, Storm's measure of end-to-end latency, is calculated and figure 2.13 illustrates how the measured complete latency differs from the physical path end-to-end latency which our method predicts.

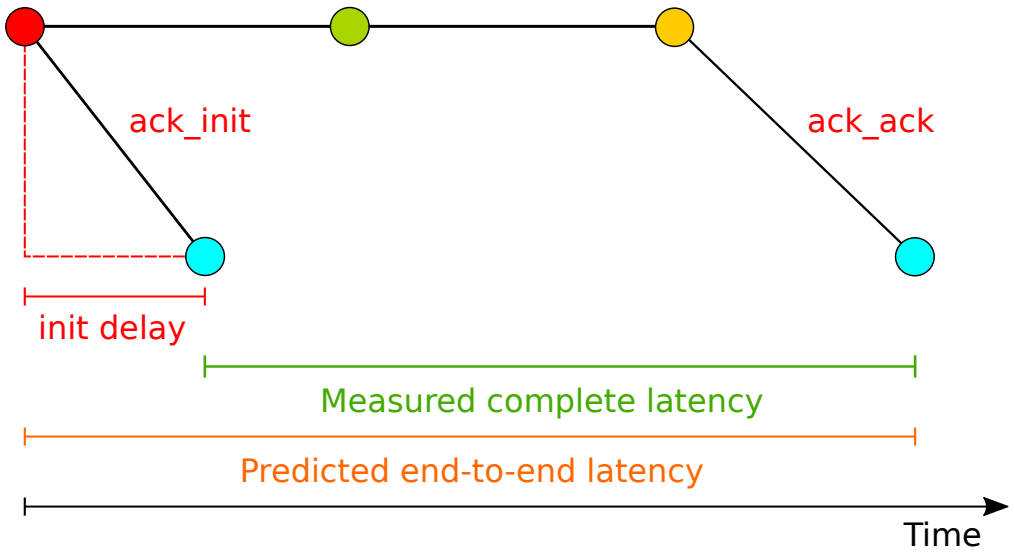

Figure 4.13: Timeline for the tuple path shown in figure 4.12, comparing the predicted end-to-end latency to the measured complete latency.

The path illustrated in figure 4.12 includes the *ack delay*, which the measured complete latency will also include, however it also includes the *init delay* which will not form part of the measured complete latency for that path. This situation is illustrated in figure 4.13. Therefore, in order to estimate the expected complete latency for a given path, the init delay must be subtracted from the predicted end-to-end latency of the physical path. The init delay of a given path can be predicted by estimating the end-to-end latency of the path from the source spout executor to the Acker executor of the tuple physical path. Figure 4.14 illustrates the init delay path for the tuple physical path shown in figure 4.12. This prediction can utilise all the same methods as the main tuple physical path end-to-end latency prediction.

We now have a way of estimating the complete latency of every physical path in tuple flow plan. However, the complete latency reported by Storm is not measured for every path. As shown in figure 2.14 the complete latency is only reported for the last child tuple of the tuple tree rooted at the original source tuple. Therefore, in order to estimate the likely complete latency that Storm would report for a proposed physical plan, we need to select the longest path from each spout on each output stream (which is how Storm

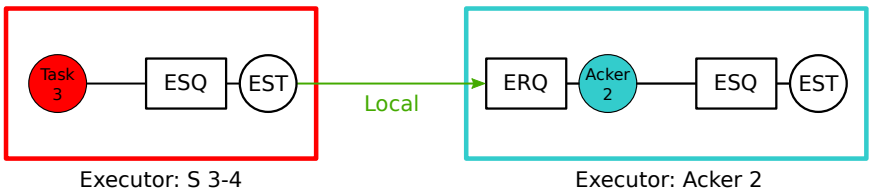

Figure 4.14: An example path that the *ack_init* message from the spout will take to the Acker executor for the path shown in figure 4.12.

breaks down the complete latency metrics). An average of these values will more closely approximate the average complete latency reported by Storm.

4.14 Summary

The many modelling aspects described in the sections above, that form the performance prediction for a given physical plan, are summarised below:

1) Predict the incoming workload level (section 4.3).
2) Predict the routing probabilities (SRPs and GRPs) for the proposed physical plan (section 4.4).
3) Predict the input to output ratios for the proposed physical plan (section 4.6).
4) Use the predicted incoming workload level, routing probabilities and I/O ratios to calculate the expected tuple arrival rate at each element of the proposed tuple flow plan (section 4.8).
5) Predict the service time of the executors (section 4.9).
6) Estimate the tuples per input list/map at each ERQ and WPTQ (section 4.12).
7) Simulate the executor ULT delays (section 4.2.2), EST delays (section 4.13.2) and WPST delays (section 4.13.3).
8) Estimate the network transfer delay between each worker node in the cluster (section 4.11).
9) Generate all physical paths through the tuple flow plan from each spout to each Acker.
10) Calculate the end-to-end latency of each physical path from the latencies calculated above.
11) Estimate the complete latency from the predicted physical path end-to-end latencies (section 4.13.5).

Chapter 5

Evaluation

This chapter details the results of the evaluation of the modelling processes outlined in chapter 4. It evaluates the accuracy of the predictions made by the modelling system implementation described in section 5.1, using data gathered by the system outlined in section 5.2.

5.1 Modelling System Implementation

The modelling system implementation (named Storm-Tracer) consists of three main components: metrics gathering; topology structure storage and analysis; and performance modelling. Figure 5.1 shows how these major components interact with an Apache Storm cluster.

When the Storm scheduler creates a new physical plan, this is issued to the Storm-Tracer system which will store this in the topology structure store. The performance modelling system will then enact the procedures outlined in chapter 4 by querying the metrics system to obtain the various summary statistics needed to perform the modelling. Once an estimate of the end-to-end latency for the proposed physical plan has been calculated, it is returned to the Storm scheduler which can then decide if it wishes to deploy the plan or create a new one.

Further details of the implementation of the modelling system used in this evaluation are given in appendix C.

5.2 Evaluation System

Each of the modelling approaches detailed in chapter 4 requires a set of input metrics from a running *source* topology. Similarly, in order to ensure that the prediction methods are producing accurate results for a *proposed* physical plan, metrics from the topology

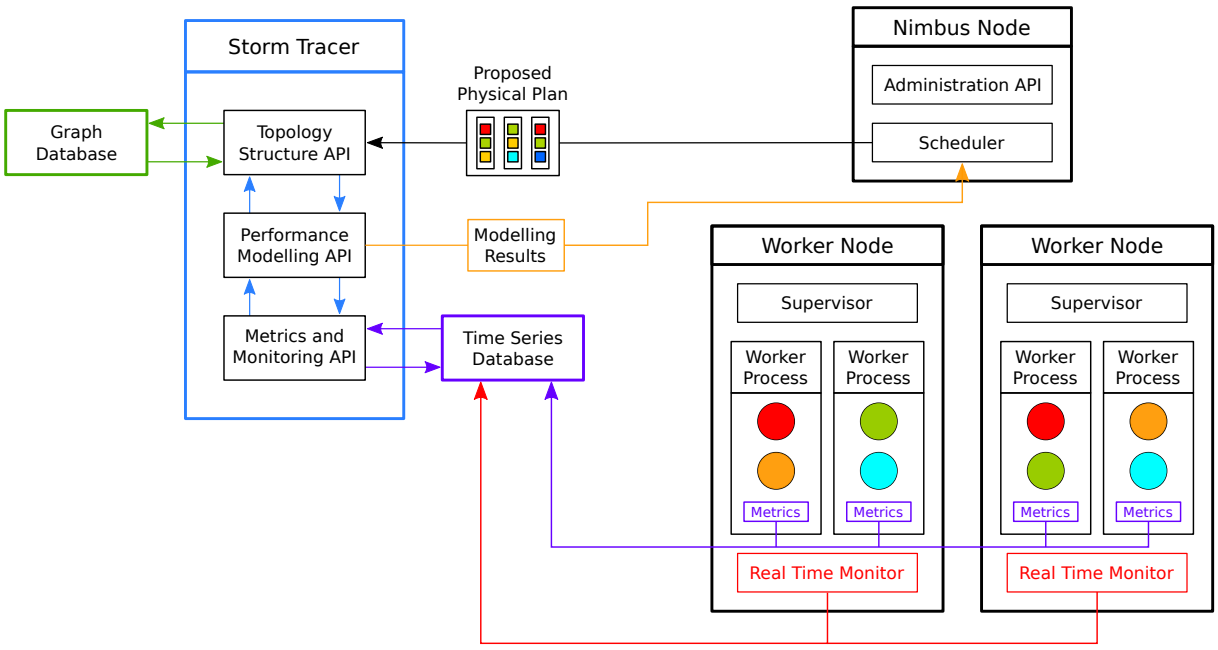

Figure 5.1: The various components of the Storm-Tracer system.

when configured according to that proposed plan also need to be available. In order to provide both the source and actual performance metrics, a data gathering system capable of recording the metrics and physical plans of multiple topology configurations was created.

5.2.1 Data gathering

The data gathering system consists of an experiment control program that processes several pre-defined topology configurations (see section 2.6.2). An experiment configuration file lists a number of *steps*, with each step defining the parallelism of each component in the topology and the number of worker processes assigned to that topology. The experiment configuration also defines the length of the step and a *burn-in* period. For each step, the experiment controller uses the Storm control node (Nimbus) client to issue a *rebalance* command to alter the topology configuration to match the new step. The default Storm scheduler (*EvenScheduler* — which uses a round robin approach) is used to convert the topology configurations into physical plans. At this point the experiment controller logs the start of the experiment step. It then waits for the defined burn-in period so that the topology can stabilise after the rebalance. Once the burn-in period finishes, the experiment controller logs a *start* event and then waits for half the defined step running period. At the halfway point the controller will save the current topology physical plan to a graph database. The controller will then wait for the rest of the running period to complete before logging a *stop* event and moving on to the next experiment step. These stages of the data gathering process are shown in figure 5.2.

At the end of this process there will be a series of logged start and stop event timestamps. These timestamps are logged in the same database that receives the metrics from the

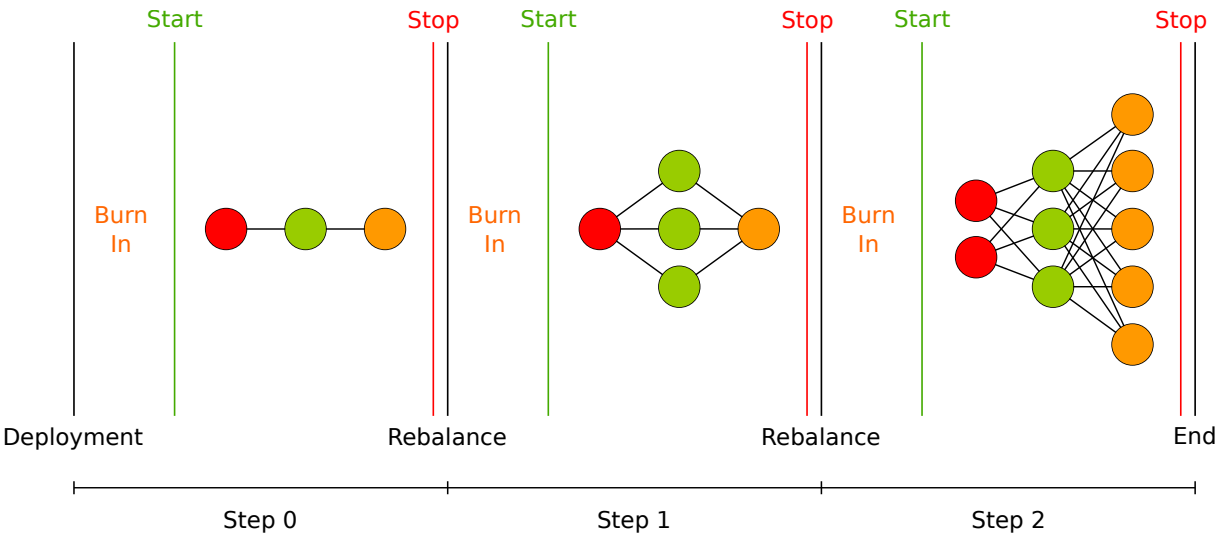

Figure 5.2: Stages of the data gathering process.

topology, therefore the timestamps of these events will be synchronised with those of the metrics. Using these experimental step periods, the modelling system (outlined in section 5.1) can predict the performance of the physical plan from one step using the metrics from another and these predictions can then be compared to the actual performance of that proposed physical plan.

5.3 Example Use Cases

A variety of test topologies were used to evaluate the modelling process. These included variations in the number of components and types of stream groupings between them. These variants are described below and use a common framework to supply input workload. Figure 5.3 shows this messaging framework. A generator creates input messages with unique identifiers (UUIDs) which are issued to an outgoing topic on an Apache Kafka[1] message broker. These messages are then taken in by a spout component in the Storm topology; the spout will then add a timestamp field to the tuples it issues to the downstream components. Each of the components in the topologies will pass this entry timestamp along as a field within their output tuples. Each component also records the component name and task identifier for the current instance processing an input tuple. These values are then appended to a *path* string which is passed downstream with each of their output tuples. At the final component in each topology, the entry timestamp is subtracted from the current timestamp and the difference logged as the end-to-end latency of the particular path taken through the topology (recorded in the path string). Both this end-to-end latency and the path followed are then sent back to the Kafka broker and retrieved by a message receiver program which extracts the values and sends them to the time series database (TSDB) (used for metrics storage) for later analysis. Obviously, the end-to-end

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

latency metric will only be accurate for paths where the spout and sink executors are on the same worker node. For those paths where this is the case, these measurements provide a latency metric that is independent of the complete latency and so avoid certain issues regarding outliers discussed in section 2.13.2 in more detail. This end-to-end latency metric is referred to as the *ground truth latency* and is discussed in more detail in section 5.7.1.

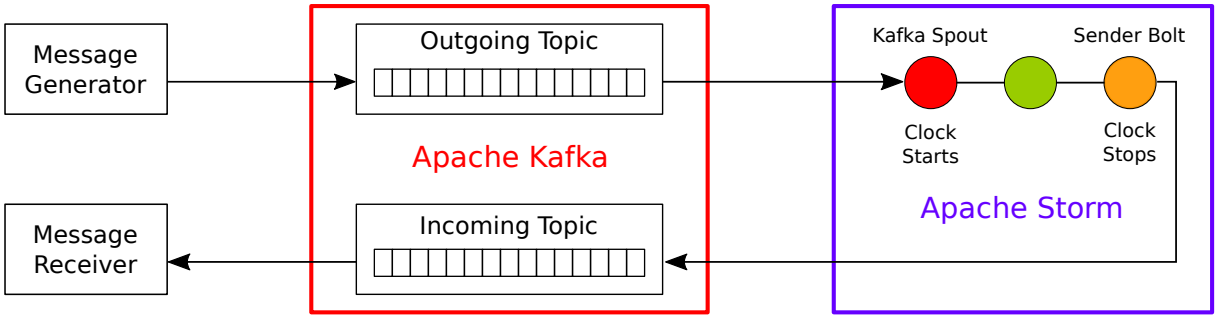

Figure 5.3: Schematic of the messaging framework used by the test topology.

5.3.1 Linear topologies

The most common form of distributed stream processing system (DSPS) topology is a linear arrangement of components defining a single pipeline of operations. In order to test the majority of the modelling approaches discussed in chapter 4, several different linear test topologies have been defined:

Fields to fields

Figure 5.4: Four component linear topology with an I/O ratio of 1.0 and consecutive fields grouped components.

The fields to fields linear topology, shown in figure 5.4, tests the routing probability calculations (described in section 4.4) by providing consecutive fields grouped components. It contains two intermediate components, the first of which (bolt A) is connected to the spout via a shuffle grouped connection. This takes the input tuple from the spout (that contains a unique message identifier, the entry timestamp and the path string) and converts this into a Java class with attributes matching the fields of the input tuple. It appends its component and task identifiers to the path string in the Java object and then converts this object into a JSON string representation and, along with the entry timestamp, emits this in its output tuple which is sent via a fields grouped connection to the next component. This bolt will also add a *key* field to its output tuples, which

takes the form of a single letter of the alphabet (A-P). The key for each output tuple is assigned according to the distribution shown in figure 5.5. This makes it much more likely that tuples will be given keys in the lower alphabet range, leading to an unbalanced key distribution into the executors of the downstream component. The second intermediate component (bolt B) is the same as the first and will simply deserialise the JSON object, add the path information, serialise it again and issue an output tuple via a fields grouped connection using the key generation described above to the final Kafka sender component. Both the intermediate components (bolts A and B) generate one output tuple for each input tuple, as denoted by the ratios above them in figure 5.4.

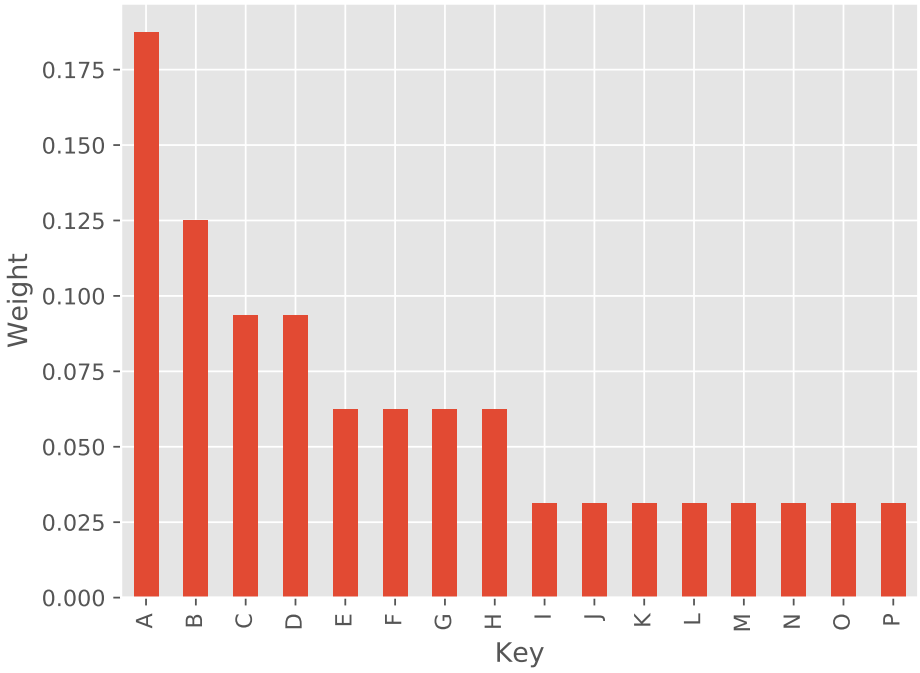

Figure 5.5: Probability that a given key will be chosen for the fields grouped connections.

Multiplier

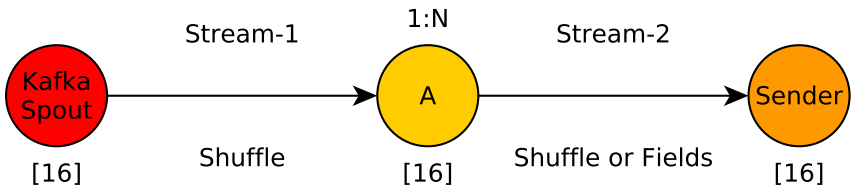

Figure 5.6: Three component linear topology with an I/O ratio greater than 1.0.

The multiplier linear topology, shown in figure 5.6, uses a similar design to the fields-to-fields test topology but with only one intermediate component where for every input tuple it will create n output tuples, where n is a value chosen from a normal distribution with

mean N and standard deviation σ. This replicates the behaviour of the intermediate node in a word counting or tweet processing topology that is splitting up blocks of text into individual words. The multiplier component adds keys to outgoing tuples in the same way that the fields to fields topology does and so the connection to the sender bolt can be either shuffle or fields grouped.

Windowing

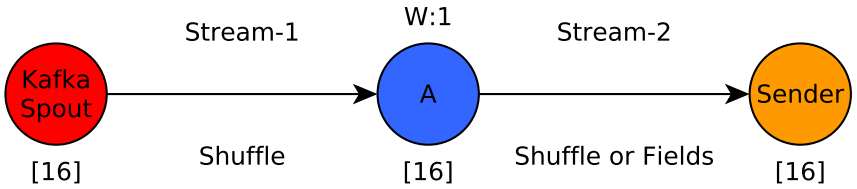

Figure 5.7: Three component linear topology with an I/O ratio less than 1.0.

The windowing linear topology, shown in figure 5.7, uses a similar design to the multiplier topology described above. However, instead of producing N output tuples for every input, the intermediate component in this topology will window W incoming tuples and produce a single output tuple. This topology is intended to test the modelling approach laid out in section 4.13.1.

The intermediate component (bolt A) uses Storm's windowing application programming interface (API) with a tumbling window arrangement (see section 2.12 for more details). The value of W is determined at topology submission and is fixed for the lifetime of the topology. In order to ensure that the batching of tuples does not lose information about the origin spout task and entry timestamp, the windowing component will sort the W input tuples by the value of their *path* field. It then chooses a path at random and one tuple at random from all those that have followed that path: this tuple will then be used as the basis for the output tuple. If there is more than one tuple following the chosen path then the entry timestamps of all those tuples are averaged and this is used as the emitted tuple's entry timestamp. As the connection into this component is a shuffle grouping, this random selection should not adversely affect the distribution of paths logged in the message receiver.

All in one

The *all-in-one* linear topology, shown in figure 5.8, combines the multiplier and windowing components, described above, with consecutive fields grouped components to create a more complex topology which will test many aspects of the modelling system in a single topology.

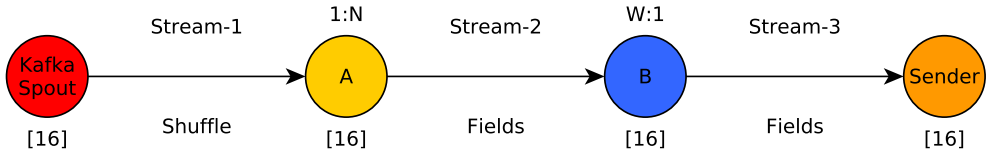

Figure 5.8: Four component linear topology with both multiplying (I/O ratio > 1) and windowing (I/O ratio < 1) components which have consecutive fields grouped connections.

5.3.2 Join and split topology

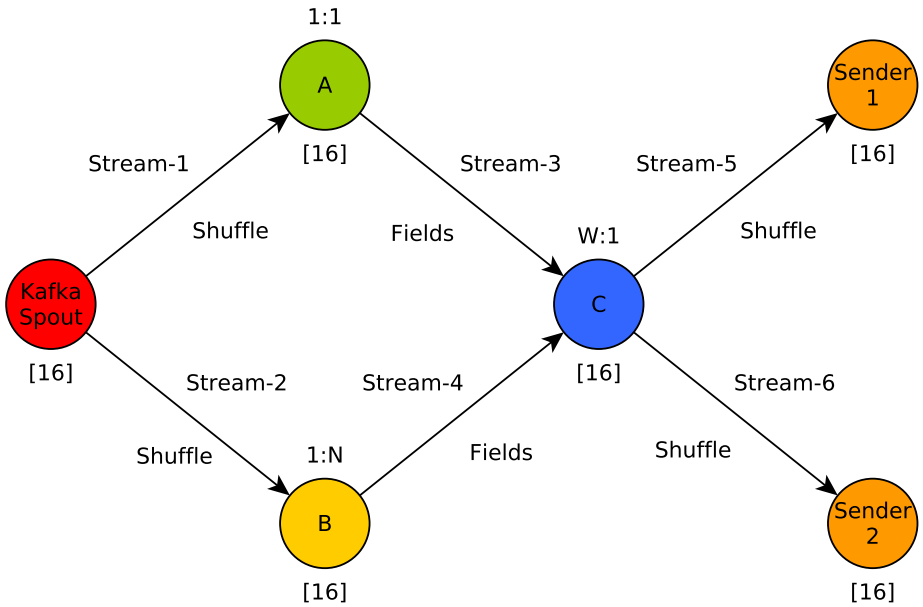

Figure 5.9: Six component topology (all with an I/O ratio of 1.0) with a combined stream join and split.

As well as linear topologies, Apache Storm provides the ability for bolts to consume multiple incoming streams and join the tuples of those streams to produce outputs which themselves can be emitted onto multiple outgoing streams. The join and split test topology, shown in figure 5.9, uses two separate intermediate bolts — one of which (bolt A) is a one-to-one component (the same as bolt A from the fields to fields topology) and the other (bolt B) is a multiplier component (the same as bolt A from the multiplier topology). These components then feed into a single join bolt (C) which combines the incoming streams and then emits onto two output streams. This arrangement tests the input to output (I/O) ratio predictions, described in section 4.6, by providing multiple input and output streams.

Storm provides a join API[2], which provides abstract classes and functions to easily join separate streams in a single bolt. This requires the incoming streams to be windowed

[2] https://storm.apache.org/releases/1.2.2/Joins.html

into batches (see section 2.12). All the tuples arriving in a given duration or count on both streams will be collected and passed to the joining bolt's `execute` method as a single collection.

The join bolt performs a similar operation to bolt A in the windowing topology. It chooses tuples at random from the input collection. However, as there are two downstream components it will pick two at random to send to sender component 1 and a third random tuple to send to sender component 2. This ensures a clear difference in the output stream amounts to aid in the I/O ratio testing.

5.3.3 Test topology summary

Table 5.1 summaries all the topologies described in the sections above. The code for these test topologies can be seen in the *StormTimer* folder in the digital media attached to this report or online[3].

Table 5.1: Summary of the test topologies used in the performance modelling evaluation.

Topology name	No. Components	Figure
Fields to fields	4	5.4
Multiplier	3	5.6
Windowed	3	5.7
All-in-one	4	5.8
Join-split	6	5.9

5.3.4 Test configuration

All of the test topologies share an input message rate, into the Kafka broker, of approximately 16 messages per second (unless stated otherwise). All components were assigned 16 tasks each, this is to match both the number of partitions of the Kafka message topic handling the outgoing and incoming messages from the topology and also so that at maximum scale out one key would map to one task for any of the fields grouped connections.

Generally, the configuration of each topology experiment involved five steps. Starting with each component having a single replica in the first step and then, for each following step, doubling the number replicas of each component until each has 16 in the final step.

Unless otherwise stated, the three-component topologies used two worker nodes with two worker processes each (four in total) for each experiment. For the four component all-in-one topologies, three worker nodes with two worker processes each were used and for

[3] https://github.com/tomncooper/StormTimer

the join-split topology four worker nodes with two worker processes each. This ensured a good mix of connection types between components and avoided overloading of worker nodes during experiment steps with higher numbers of replicas. The full experimental configuration used with each of these test topologies can be seen in Appendix E.

5.3.5 Evaluation process

Generally, in the evaluation of the prediction of the various parameters described in the sections below, each of the experiment steps are used as sources for the metrics used to predict the performance of the physical plans from each of the other experiment steps. For example, the metrics from step one will be used to predict the performance of the topology physical plan from steps zero, two, three and four.

For each of the predictions of the performance parameters of a given step, the prediction is compared to the actual measured performance of the particular parameters in that step. The *relative error* (ϵ) is used to assess the accuracy of each prediction and is defined as:

$$\epsilon = \frac{P - A}{A} \tag{5.1}$$

Where P is the predicted value and A is the actual measured value of the parameter. This means that errors will be negative if P is an under-prediction and positive for an over-prediction. As errors can be negative, the absolute error value is used whenever summaries of the error are calculated. The calculation of the *actual* parameter value varies according to what is being tested. How this value is obtained is described in the sections below.

5.4 Arrival Rates

The arrival rate prediction methods, described in section 4.8, depend on the accurate prediction of several key parameters, namely the routing probabilities between each pair of connected executors and the I/O ratio of each executor.

5.4.1 Stream routing probabilities

The fields-to-fields test topology includes shuffle as well as consecutive fields grouped connections. This requires using the processes outlined in section 4.5.1 to predict the routing probabilities of Stream-2 and Stream-3 (see figure 5.4).

To validate the routing probability predictions, the stream routing probability (SRP) for each of the logical connections in each of the proposed tuple flow plans was estimated and

the relative error calculated according to equation 5.1.

Table 5.2: The median absolute error across all experiment steps for each of the streams in the fields-to-fields test topology.

Stream	Absolute Error (%)
Stream 1	6.2
Stream 2	10.5
Stream 3	9.4

Table 5.2 shows the median absolute error across all connections in all source and proposed step combinations. The median absolute error is used in order to remove the effect of outliers in the experimental steps which have high parallelism and therefore contain many more connections. This table shows that the error on the shuffle grouping is (3-4%) lower than for the fields grouped connections. Figure 5.10 further breaks down the SRP prediction errors for each source and proposed step combination. In this figure the median absolute error in the SRP predictions is calculated using each step as a metrics source to predict every other step. Step 0 is not shown as this has all components with a parallelism of one, therefore the SRP must always be 1.0 and so the prediction error is always zero. Figure 5.10 shows that predicting topologies with lower levels of parallelism (see Appendix E for configurations) can lead to very low prediction errors of less than 1% for step 1 and less than 3% for step 2. However, the error increases as the level of parallelism increases. Each step's topology has twice the parallelism of the previous step and this progression in the parallelism is reflected in the progression of the SRP prediction error. Interestingly step 4, which has the highest level of parallelism, shows the lowest error in predicting the other step's physical plan SRPs. This is likely due to predictions using step 4 as a source always predicting a scale down situation and, as each task in step 4 is in its own executor, it has the most detailed task to task routing information.

5.4.2 Input/Output ratios

The I/O ratio calculations, described in section 4.7, were tested using both the all-in-one linear topology (see figure 5.8) and the join-split topology (see figure 5.9). The settings used for both of these evaluations are described in section E.

To validate the estimates of the input stream I/O coefficients, the predicted coefficient values for a proposed physical plan were multiplied by the measured total input count (from the experiment step where the proposed physical plan was running) on each input stream of each executor to calculate a corresponding predicted output amount on each output stream. These predicted output counts were then compared to the measured output

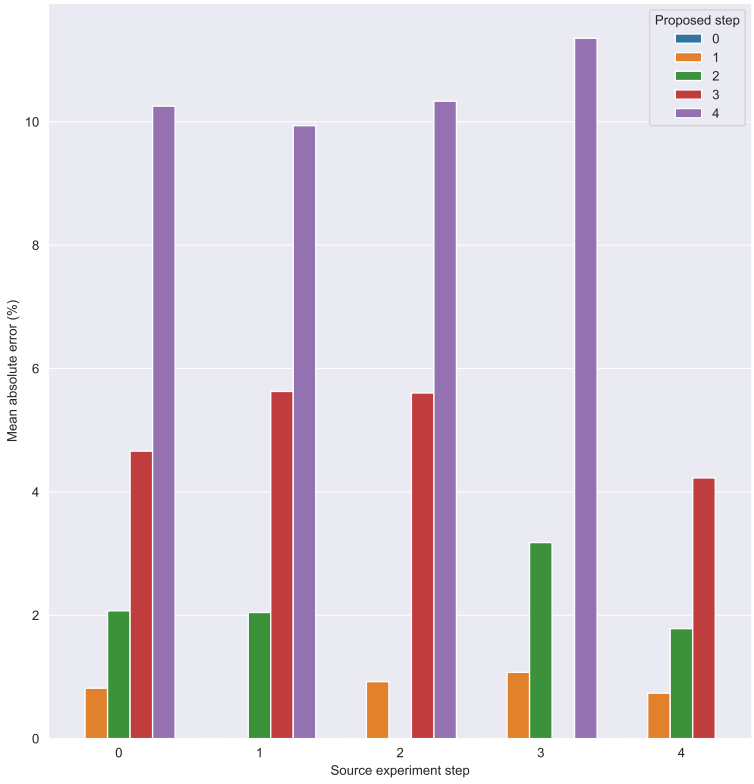

Figure 5.10: Comparison of the stream routing probability prediction error for the fields to fields test topology.

counts and an error calculated using equation 5.1.

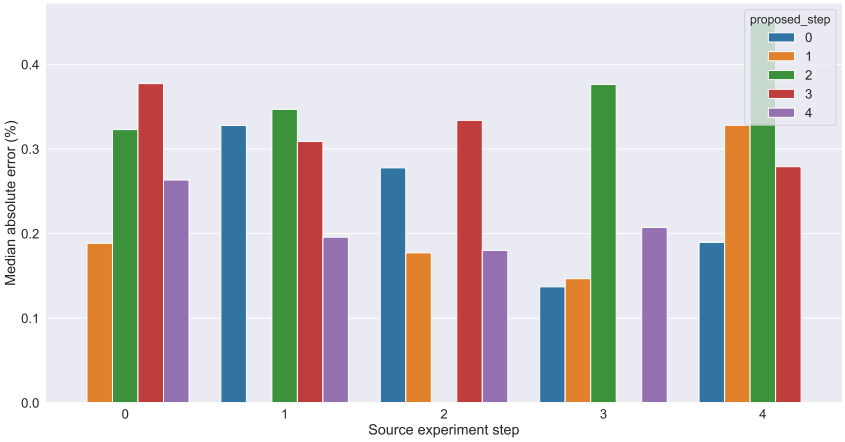

Figure 5.11: Median absolute error in the I/O ratio predictions for the all-in-one test topology.

The results show that, for the all-in-one linear topology, errors were well below 1% across all source and proposed experiment step combinations, see figure 5.11. This is not surprising as the all-in-one topology (shown in figure 5.8) has only a single input and output stream for each component and the windowing bolt (B) is using a count based window of a fixed size. Also, although the multiplication bolt (A) is multiplying by a random amount, this value is based on a normal distribution with a fixed mean, so over a sufficient window (20 mins in the case of this experiment) it will converge to the mean multiplication factor.

The join-split topology (shown in figure 5.9) is a more complex test case. The one-to-one and multiplying bolts both feed into a bolt which windows and combines the streams into a single batch of tuples (based on count) and then outputs onto two separate streams. This tests the effectiveness of the least squares regression method, described in section 4.7, in identifying the coefficients for multiple input streams.

Figure 5.12 shows the median absolute error percentage for the I/O ratio predictions for each source and proposed experiment step combination for the join-split topology. The results shown here used a count window of 100 and for every window a tuple was emitted on each output stream. Figure 5.12 shows that the errors do increase with the parallelism of the source physical plan, but are less than 6% and are typically less than 1%.

Figure 5.13 shows the I/O ratio errors for the join-split topology using a time based window of 2 seconds and a more complex routing behaviour. For this experiment the join component will emit one tuple onto Stream-5 if the count of tuples, in the window, from the input streams (Stream-3 and Stream-4) are both even or both odd. If this is not the case it will emit a tuple onto Stream-6. This more complex behaviour is not a simple linear relationship and figure 5.13 shows significantly higher errors compared to the simpler case

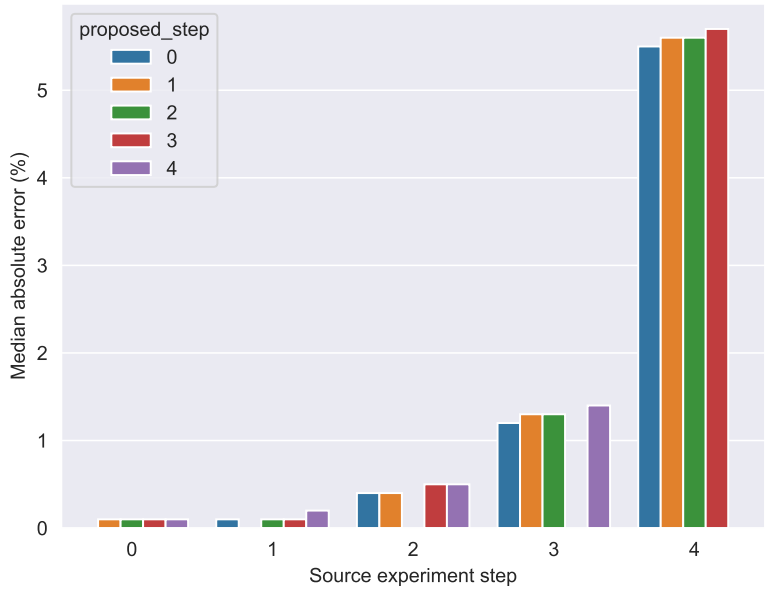

Figure 5.12: Median absolute error in the I/O ratio predictions for the join-split test topology using a simple routing behaviour.

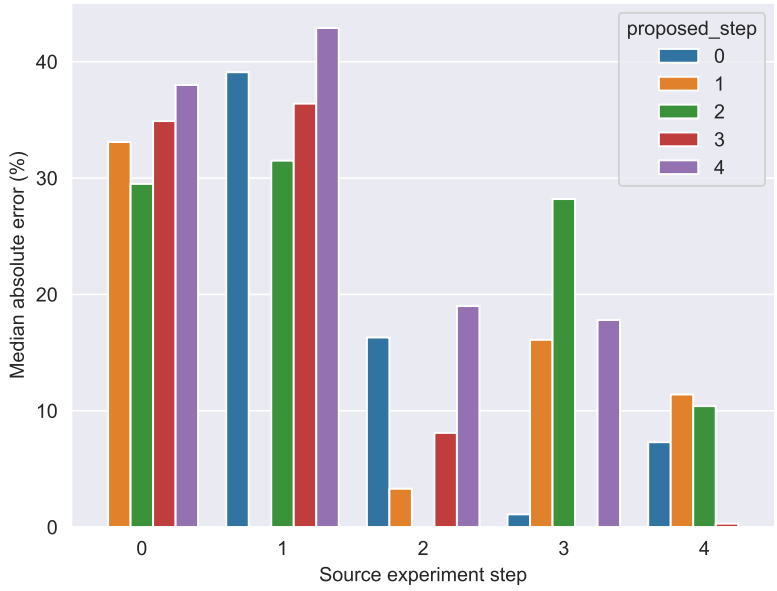

Figure 5.13: Median absolute error in the I/O ratio predictions for the join-split test topology using a more complex routing behaviour.

shown in figure 5.12. The error reduces as the parallelism of the source topology physical plan increases. This is likely due to the estimation having a greater variety of input data to average across. These results suggest that the least square approach is sufficient for simpler topologies, however a more robust approach such as Gaussian Process regression which is better able to account for non-linear behaviour may be more appropriate in the case of more complex routing behaviour.

5.4.3 Executor arrival rates

Similarly to the I/O ratio validation described above, the arrival rate prediction methods described in section 4.8 were tested against the all-in-one and join-split test topologies. Figures 5.14, 5.15 show, for the all-in-one and join-split topologies (with simplistic routing) respectively, the median absolute error when using each experiment step as the source metrics for the prediction of the arrival rates at each executor of every other experiment step. As discussed in section 4.3, the incoming workload (the emission rate of every spout executor) is taken from the actual measured values of the topology running in the predicted experiment step. As such this is a *perfect* (or at the very least highly accurate) prediction of the incoming workload.

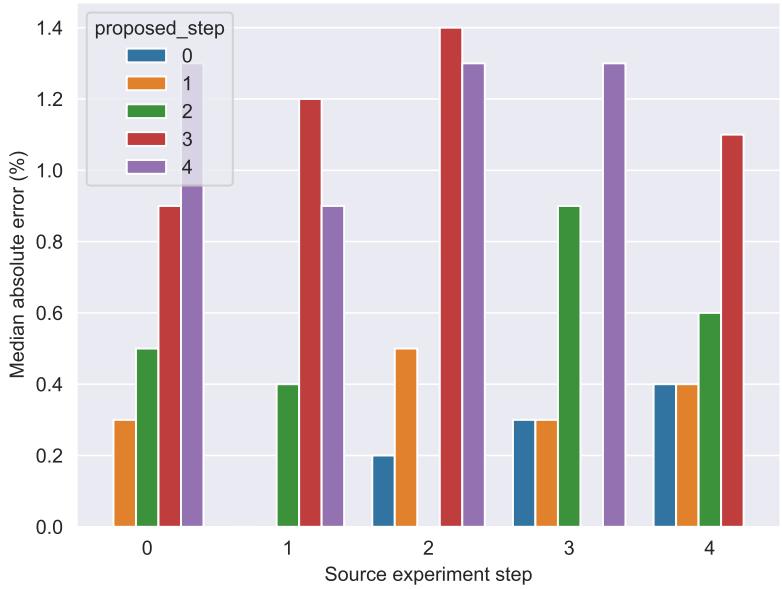

Figure 5.14: Median absolute error in the arrival rate predictions for the all-in-one topology.

Both these figures show that the error in the arrival rate calculations is generally below 2.5%, being larger for the predicted steps with higher parallelism.

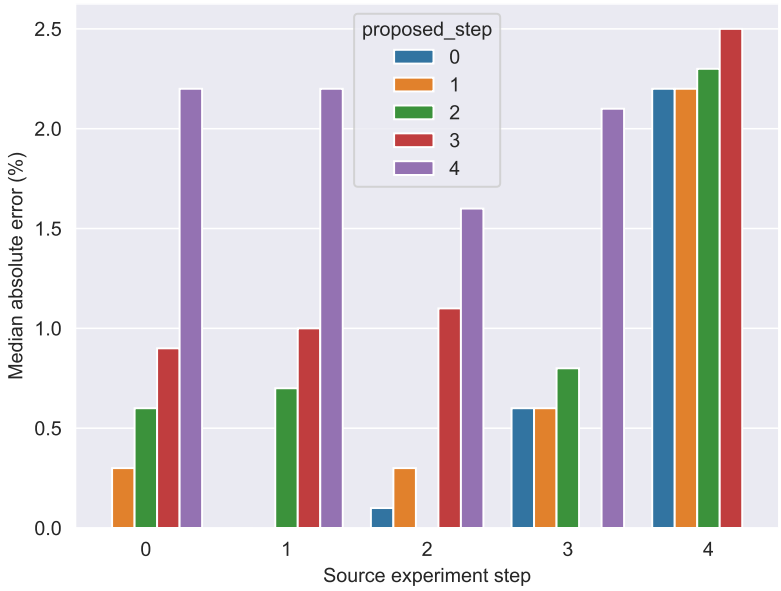

Figure 5.15: Median absolute error in the arrival rate predictions, for the join-split topology.

5.5 Service Times

Section 4.10 describes the approach taken in estimating the service time for the executors of a proposed physical plan. As discussed in that section, the modelling approach is based on a weighted average of the services times from each executor of a given component in the source physical plan. Figure 5.16 shows the median absolute error in the service time predictions for the all-in-one test topology.

What is immediately apparent from this chart is that the prediction errors, ranging from 20% to 300% in the extreme, can be significant. The errors increase with the parallelism of the source physical plan. This is most likely a result of co-location effects, which were discussed in section 4.10. In step 0 there is only one executor for each component and therefore each worker process has a small number of co-located executors. The service time metrics from these executors will be subject to less thread pausing. However, for step 4, there are 16 executors for each component and therefore many executors in each worker process. The service times from these executors are likely to be inflated, as long pause periods will extend the service times compared to the same code running with the same input on a worker process with fewer assigned executors. When these metrics are used to predict the service times for executors (in a proposed physical plan) with a lower level of parallelism they will tend to over-estimate the actual service time. Conversely, using service time metrics from a physical plan with low co-location, to predict one with high co-location, will mean that the prediction will tend to under-estimate the actual service time.

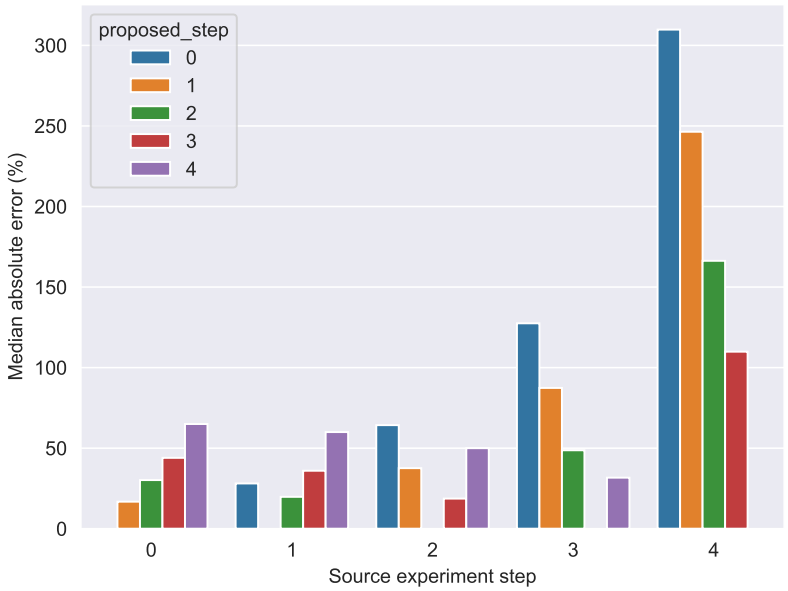

Figure 5.16: Median absolute error in the service time predictions for the all-in-one topology.

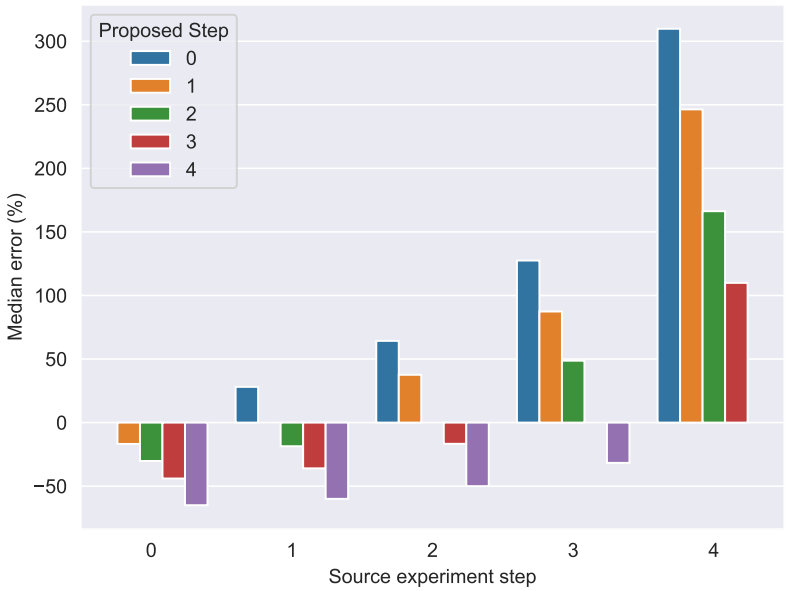

Figure 5.17: Median relative error in the service time predictions for the all-in-one topology.

This effect is confirmed in figure 5.17, which shows the median relative error (as opposed to the absolute error shown in the earlier charts) for the all-in-one topology, which will be negative for under-predictions and positive for over-predictions. This shows that, as the difference in the level of parallelism of each predicted physical plan compared to the source physical plan increases, so does the error. It also shows that for "scale up" predictions (using a low parallelism physical plan to predict a high parallelism one) the service times are under-predicted and vice versa. This pattern is also present in the other test topologies.

Regarding the effect of the service time prediction errors, on the prediction of the executor sojourn times, we make two observations: firstly, that positive errors in the predicted service time (b_P) imply that the predicted service rates ($\mu_P = \frac{1}{b_P}$) are smaller than the actual service rates (μ_A). Using an $M/M/1$ queueing system as an example (see appendix A for more details), the sojourn time (W) across the queuing system is given by:

$$W = \frac{1}{\mu - \lambda} \tag{5.2}$$

Therefore, when the predicted service rates are smaller, the predicted sojourn times (W_P) will be longer than the actual sojourn times (W_A).

Secondly, the effect on the sojourn times also varies with the difference between the service and arrival rates. When the difference is very large and the service rate is much greater than the arrival rate, the effect of service rate errors on the predicted sojourn times is reduced. Typically, in the experiments listed in this chapter, the service rate is several orders of magnitude larger than the arrival rate.

Figure 5.18 shows a comparison of the results of the queue simulator (described in section B.3) with differing levels of service time errors. In this comparison the arrival rate was fixed at two different levels (0.1 & 1.0) and various levels of error were applied to the actual service time of 0.1, which equates to an actual service rate of 10. The latency error was calculated against the simulator results using the actual arrival and service rates. For the 0.1 arrival rate, figure 5.18 shows that the service time error of 300%, shown in figure 5.17 for the case when experiment step 4 is used as the metrics source, leads to sojourn time errors of 50%. However, most of the cases shown in figure 5.17 have service time errors between -50 and 100% and this leads to sojourn time errors of between -10% and +20%.

The 1.0 arrival rate line, shown in figure 5.18, illustrates how the service time error can have more of a pronounced effect (steeper curve) when the actual arrival rate is closer to the actual service rate (10), in this case an order of magnitude closer. As mentioned previously, the experiments in this chapter have arrival rates several orders of magnitude

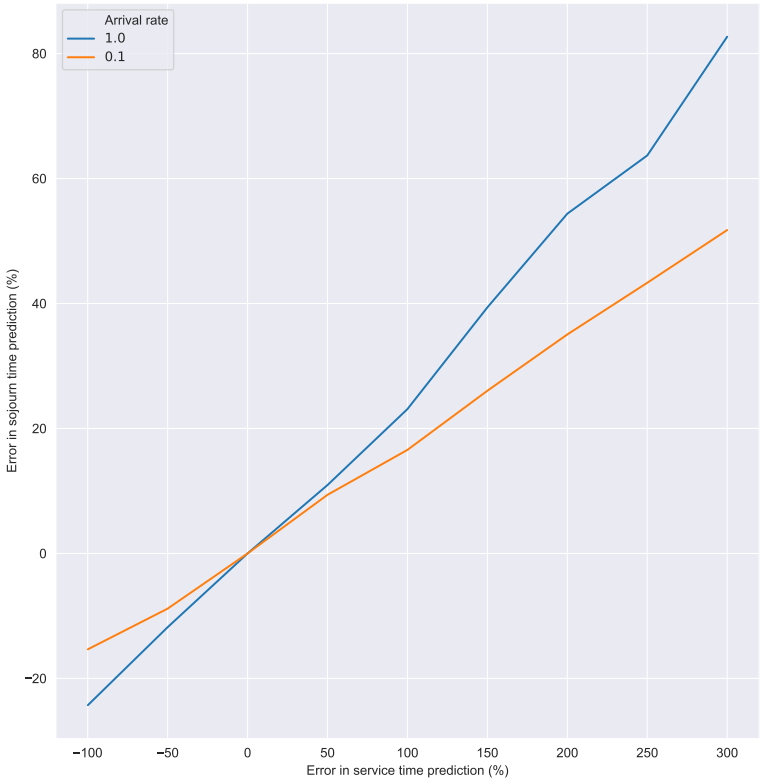

Figure 5.18: Effect of errors in the service time on the queue soujour time simulation with an arrival rate of 0.1 and 1.0.

lower than the service rates and so the service time error effect curve is likely to be shallower than those shown in figure 5.18.

The results discussed in this section imply that the modelling system may be less accurate in scale down situations where the service time is more likely to be over-estimated and also in situations where the actual arrival and service rates are close. These results lead to the requirement that thread pausing due to high co-location of executors in the worker processes needs to be carefully accounted for in the service time predictions. Such a requirement is discussed as future work in section 6.3.4.

5.6 Tuple Input List Size

Section 4.12 describes the approach to modelling the number of tuples expected in each list arriving into the executor receive queue (ERQ) (see section 2.7 for more details). This involves predicting several additional parameters related to the flow of tuples through the executors (see section 2.8.1) including the arrival rate of individual tuples at the executor send queue (ESQ) and the number of tuples in each map sent to the worker process transfer queue (WPTQ). In order to validate the predictions of these values it was necessary to modify the version of Apache Storm running in the experimental cluster. By default, Apache Storm does not provide metrics on the number of tuples in the objects arriving at each Disruptor queue, of course if it did there would be no need to predict the value in the first place. The source code for Apache Storm version 1.2.2 was altered[4] to add a metric to the standard Disruptor queue which provides an average *tuples per object* reported every metric bucket period (which is configurable and is 10 seconds by default). This metric is reported to the custom metrics consumer along with the other default metrics.

Figure 5.19 shows the validation results for the input list size predictions using the all-in-one topology (see experiment All-In-One-1 in section E). This shows that the accuracy achieved for scale up scenarios is much higher than for scale down. This is likely due to the fact that the input rate for these experiments is fixed and therefore, as the parallelism of each component is increased, the input list size tends towards one and as the prediction has a floor of one then the true value and prediction will converge as the arrival rate into each executor reduces.

To avoid a situation where the input list sizes are likely to be one, or close to one, a second experiment was performed with the Disruptor queue flush interval (δ – see section 2.7) set to a much longer (1000 ms) period than the default 1 ms. As the Disruptor queue batch size remained at the default of 100 tuples this ensured that, with the total topology input rate of 16 tuples per second, the input lists would contain multiple tuples. The results of

[4]The altered version of Storm can be seen at: https://github.com/tomncooper/storm/tree/batch-size

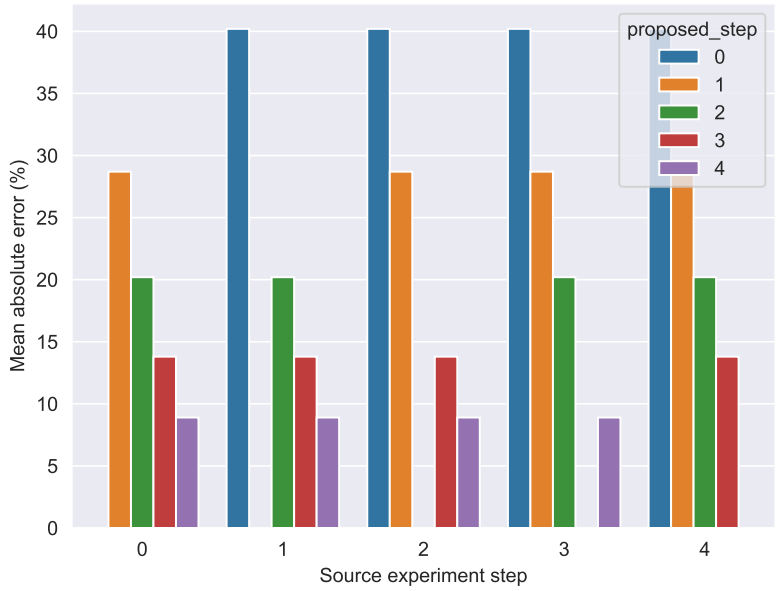

Figure 5.19: Comparison of the mean absolute error in the prediction of the input list size into each ERQ for the all-in-one topology.

this second experiment are shown in figure 5.20.

From these results, we can see a similar pattern to that seen in figure 5.19. The scale up scenarios have higher accuracies, and the higher the parallelism in the proposed step the lower the overall error. This suggests that the convergence to one tuple per input list is not driving this lower error, for the high parallelism physical plans, as this experiment had multiple tuples per input list. These results show that there must be additional processes in the tuple flow which are affecting the list sizes arriving at the ERQs.

However, for most scenarios the error is less than 30% and for most low to medium throughput situations (e.g. less than 1000 tuples per second per executor, using the default settings) the input tuple list will contain only a single tuple. However, for high throughput situations the current modelling approach suffers from larger prediction errors. This provides motivation for adding the input list size measurement to the default Disruptor queue metrics, in order to remove the need to estimate these values. This issue is discussed further in section 6.3.1.

5.7 End-to-end Latency

The final stage of validation involves performing the full end-to-end latency calculation described in section 4.13. This includes predicting all the variables described in the sections above and analysing the predictions for each path through the proposed topology physical plan.

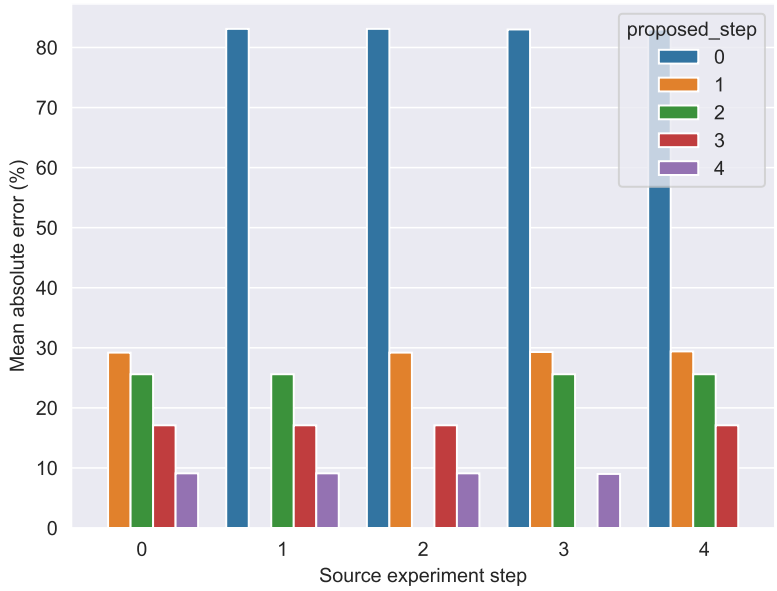

Figure 5.20: Comparison of the the mean absolute error in the prediction of the input list size into each ERQ with a flush interval of 1 second.

5.7.1 Ground truth latency

As discussed in section 2.13.2, the complete latency is highly susceptible to skew from straggler tuples. The predictions, made by the full end-to-end latency calculations, are an expected average end-to-end latency through each path in the topology tuple flow plan. This makes it difficult to validate these predictions against the measured complete latency values reported by Apache Storm, as the complete latency will be skewed heavily towards the worst case end-to-end latency for a given set of paths from a spout.

As discussed in section 4.13.5, the end-to-end latency predictions can be adapted to better match the complete latency by looking at the longest path through the tuple flow plan. However, this will not account for stragglers skewing the measured complete latency values. Therefore, to allow better validation, a separate latency measure is required. As discussed in section 5.3, the test system used for this validation includes a timestamp of when each message is first pulled off the message broker in the topology's spout executors and this timestamp is passed along in every tuple produced by the components of each test topology. The final sink component in the topology will then compare the entry timestamp to the current time and report the difference as the latency of the path that particular tuple took through the topology. This is then sent to the message broker along with a string recording the component and task instances the tuple passed through.

This *ground truth latency* can only be trusted where both the spout and sink tasks are within executors on the same worker node. Also, the elements which contribute to this ground truth latency differ from those of the complete latency. The ground truth latency

begins within the user logic thread (ULT) of each spout and therefore this latency will include the queueing delay at the spout's executor send thread (EST). This is unlike the complete latency which only begins when the Acker component receives the *ack_init* tuple from the spout. The ground truth latency ends in the sink executor's ULT and so will not contain that executor's EST delay or any delay from transfer and processing at the Acker.

The modelling process predicts values for each of the delays in a physical path through the tuple flow plan (executor ERQ & ULT, EST, worker process send thread (WPST) and remote transfer) separately and sums them appropriately for each path. Therefore, predicting the ground truth latency is simply a matter of assembling these elements in the appropriate configuration. In this way both the complete latency and ground truth latencies can be calculated as part of the same prediction.

5.7.2 Validation process

For each pair of source and proposed physical plans (from each step of the experiment), a ground truth latency and complete latency prediction is made for every possible physical path through the topology's tuple flow plan.

Complete latency

As described in section 2.13.2, the complete latency is highly susceptible to straggler tuples skewing the complete latency reported for a given metric bucket period. This can lead to long tails in the measured complete latency distribution and make using the mean of that distribution to represent the *actual* complete latency value unrepresentative. For this reason, the median value of the measured complete latency distribution is taken as the *actual* complete latency for each source spout used in the validation.

The actual complete latency value, for a given source spout, is then compared to the 90th percentile of the predicted complete latencies for each of the source spouts. As described in section 2.13.2, the complete latency is a measure of the worst case latency and so will be dominated by the longest predicted path latencies. Using the maximum path latency may be most appropriate, however this could be susceptible to outliers in the predicted complete latency values and so the 90th percentile is used to exclude the most extreme values.

Ground truth latency

For the ground truth latency, the path string recorded in each tuple as it passes through the test topology is used to filter out any paths that do not begin and end on the same worker node. These paths are then further analysed to remove outliers. Latency measures in computer networks are particularly susceptible to straggler packets causing large outlier

values in the data. Figure 5.21 shows an example distribution of raw ground truth latency measurements from the Multiplier test topology. In this particular example the vast majority of measurements are of the order of tens of milliseconds. However, the distribution has a long tail (up to 5 seconds or more) which could be due to a variety of reasons, but is most likely caused by dropped packets during sending across the network and/or failed tuples that are resent. The latency of resent tuples will not contribute to the complete latency, as failed tuples are not recorded for that metric. However, for the ground truth latency, all tuple end-to-end latencies through the topology are recorded, whether they have been resent at the network level or failed and replayed at the topology level.

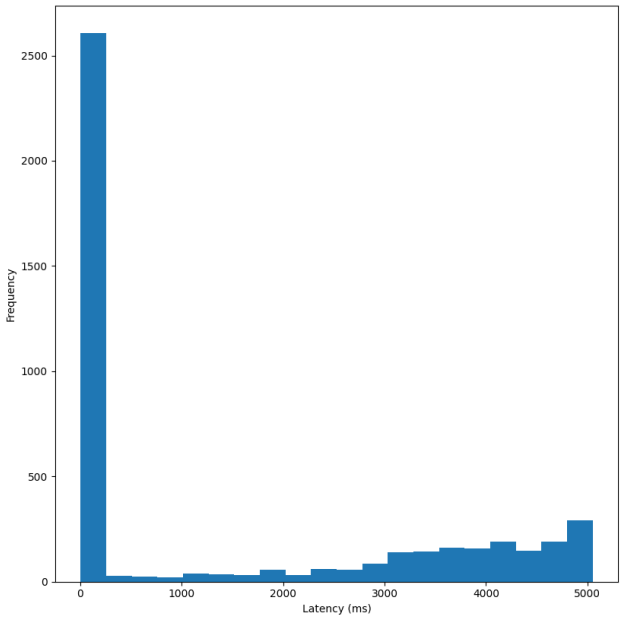

Figure 5.21: An example of the distribution of ground truth latency measurements for a single experiment step of the multiplier test topology.

To account for these outlier measurements we use the complete latency as an upper bound. As discussed in section 2.13.2, the complete latency can be considered to be a measure of the worst case latency through the topology tuple flow plan. As such, any ground truth latency measurement that is longer than the complete latency is likely to be an outlier. Figure 5.22 shows the distribution from figure 5.21 with any measurement above the 90th percentile of the measured complete latency removed. This shows a normally distributed set of latency measurements. Not all ground truth latency distributions will follow this pattern. Again, as discussed in section 2.13.2, the complete latency is prone to influence by stragglers and so using it as an upper limit does not always eliminate the long tail of the distribution. In these cases further cleaning of the ground truth latency measurements may be required; additional details of where this occurs are given in the discussions below.

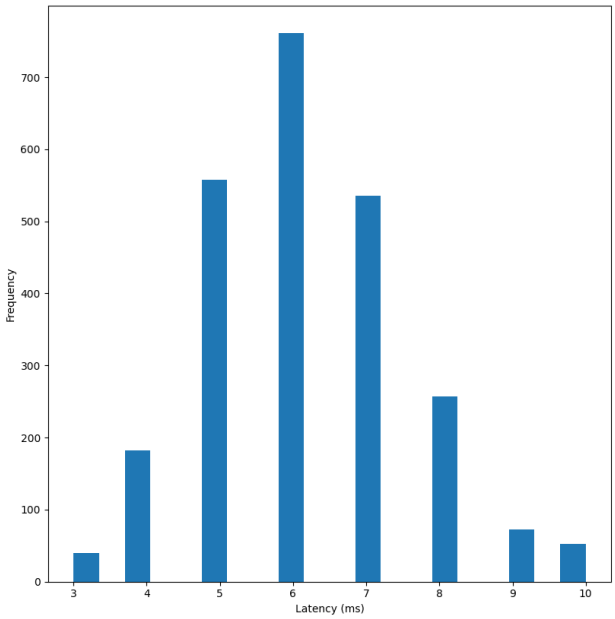

Figure 5.22: An example distribution from figure 5.21 with any measurement above the average complete latency removed.

Once the cleaned ground truth latency distribution is obtained, this is used as the *actual* measured ground truth latency for the proposed physical plan. The predicted ground truth latencies for each physical path in the proposed tuple flow plan are then filtered so that any path that was excluded from the measured values (as they did not start and end on the same worker node) is also excluded from the predicted values. Then each predicted path is given a weight, based on the sum of routing probabilities along that path. This weight is then divided by the sum of all weights in the filtered predicted paths to gain a relative weight for that path. This relative weight is then multiplied by the predicted ground truth latency and these values are summed across all paths to obtain a weighted average predicted ground truth latency. This weighted average allows the uneven routing of tuples, due to fields grouped connections, to be taken into account in the average predicted ground truth latency value.

5.7.3 Results

All of the test topologies listed in table 5.1 were tested with the five experimental steps (each topology configuration using double the parallelism for each component as the one before). The full experimental configuration used with each of these test topologies can be seen in Appendix E.

Fields-to-fields topology

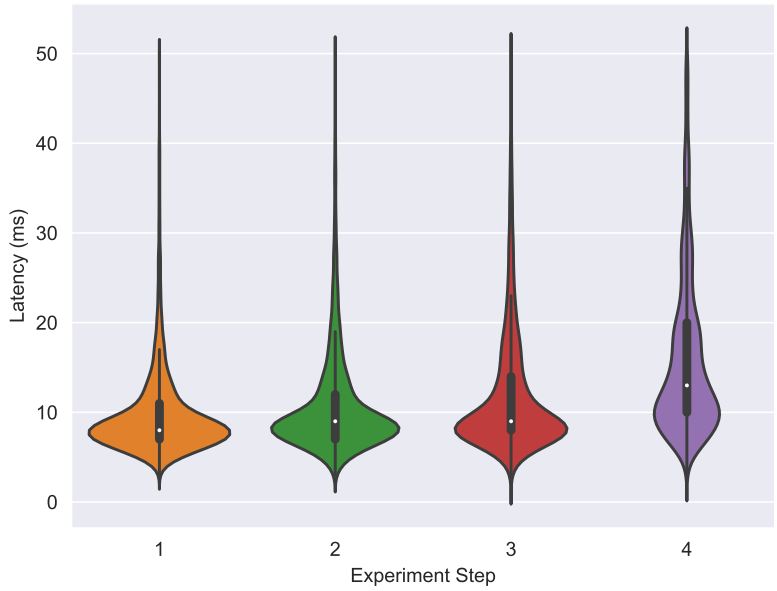

Figure 5.23: The measured ground truth latency for the fields-to-fields test topology using two worker nodes with two worker processes each.

Ground truth latency The fields-to-fields topology tests the routing probability prediction methods described in section 4.5. Figure 5.23 shows violin plots of the distributions of measured ground truth latencies for each experimental step, after being cleaned according to the method described in section 5.7.2. The shape of the distributions is shown[5] along with the median as a white dot and the inter-quartile range shown as a thick black line on the central axis of each plot. Step 0, as it only has one replica of each component, has only a single path that does not have a source and sink component on the same worker node and so has been excluded.

Figure 5.23 shows that the ground truth latency does not alter significantly from steps 1 to 3. However, step 4 shows a more diffuse distribution with a higher variance and median ground truth latency. The arrival rate for all experiment steps was fixed and so this increase in variance and overall end-to-end latency was not due to higher workload. Indeed, for step 4 there were 16 copies of each component and so the incoming load on each executor would be low compared to the other steps. However, this high level of replication is likely the reason for the increase in measured latency. The fields-to-fields topology has four components and during step 4 has 16 copies of each, if you include the one Acker and metrics consumer executor per worker process, this results in 72 executors in total for the step. The initial fields-to-fields experiment has only two worker nodes with two worker processes each, which results in 18 executors per worker process. The worker nodes were virtual machines (VMs) with only two central processing unit (CPU) cores and so

[5]The distribution is a mirror image along the vertical axis

this implies a high level of process sharing between the executors. This means that many of the executors would be *paused* during process sharing with the other threads of the worker process's Java Virtual Machine (JVM), extending the processing times. This will also result in inaccurate service time estimations when trying to estimate the performance for step 4, or when using step four as a source of metrics.

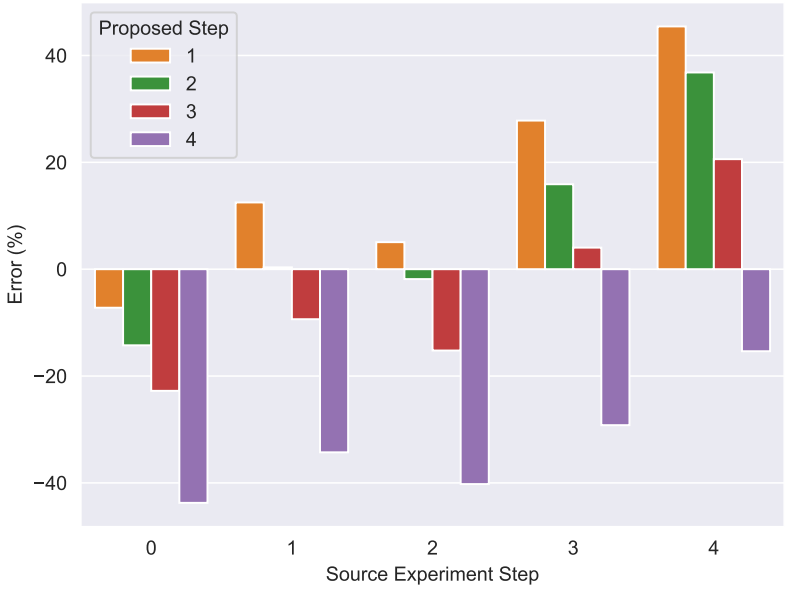

Figure 5.24: The relative error between the predicted weighted average ground truth latency and the average measured ground truth latency, using two worker nodes with two worker processes each.

Figure 5.24 shows the relative error (negative for an under-prediction and positive for an over-prediction) between the predicted weighted average ground truth latency and the measured ground truth latency for each proposed physical plan, using the metrics from each source physical plan. This shows that, when predicting step 4 or using it as a metrics source, the estimates of the ground truth latency can suffer errors of up to ±40%.

If we look at the error in the service time predictions, shown in figure 5.25, we can clearly see the effect of the high co-location of executors. Using step 4 as a metrics source leads to large over-estimations in the service time predictions. As discussed previously, this is most likely due to the thread pausing extending the measured service time used as the source data for predicting the service time of each executor.

To better gauge the accuracy of the modelling approach we can remove the parameter prediction and simply use the measured values from an experiment step to predict the performance of that step. Figure 5.26 shows the error between the predicted weighted average and the average measured ground truth latency for each experiment step using only measured values. This shows errors of 10% or lower, indicating that the modelling

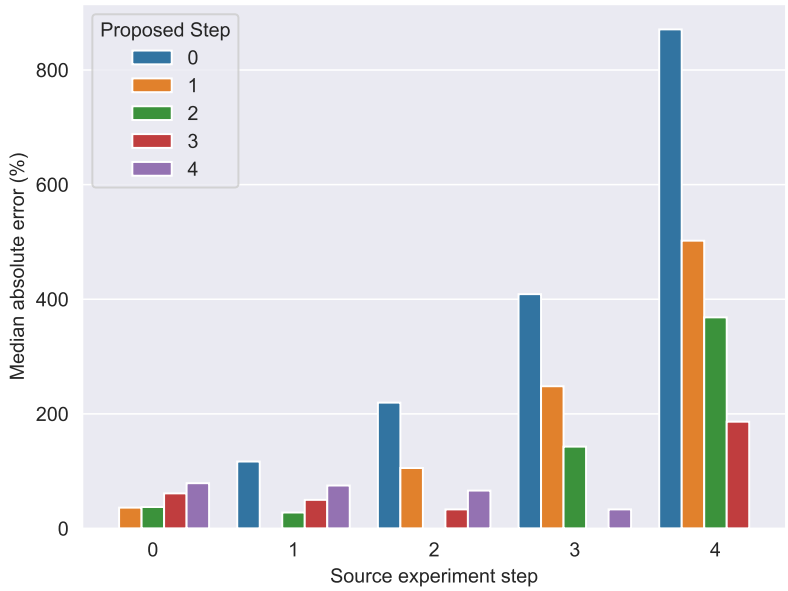

Figure 5.25: The error in the service time predictions for the fields-to-fields topology, using two worker nodes with two worker processes each.

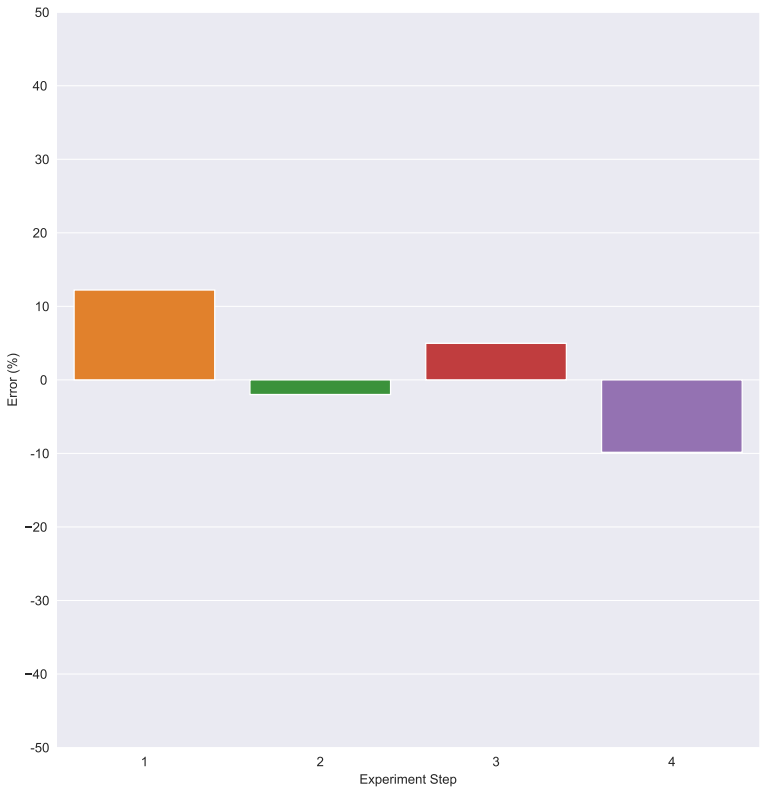

Figure 5.26: The relative error between the predicted weighted average ground truth latency (using measured parameters) and the average measured ground truth latency.

approach is valid but also how input metrics can significantly affect the predictions.

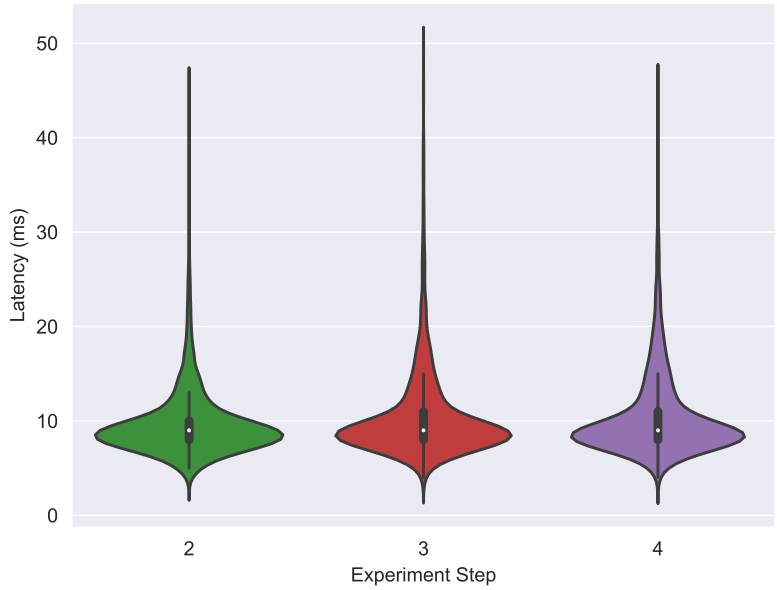

Figure 5.27: The measured ground truth latency for the fields-to-fields test topology using four worker nodes with two worker processes each.

For comparison, a second experiment using the fields-to-fields topology was performed but this time with four worker nodes with two worker processes each (eight in total for the topology). Figure 5.27 shows the distribution of measured ground truth latencies for this second fields-to-fields experiment. In this case neither step zero nor step one had any paths that began and ended on the same worker node, something made more likely the more worker nodes are used. Figure 5.27 shows that with more worker nodes and therefore lower levels of executor co-location, step 4's measured ground truth latency distribution more closely matches that of the other experimental steps.

Figure 5.28 shows the results of using the four worker node setup in the predictions. With lower co-location the prediction error is 20% or lower with most steps having errors of 10% or less. This shows what a significant effect the thread pausing, caused by high co-location, can have on the modelling results. The thread pausing typically affects all aspects of a tuple's end-to-end latency, when being served in an executor's ULT as well as when waiting in the various queues of the Apache Storm system.

These results show that the modelling system can produce good prediction for consecutive fields grouped connections, something many previous systems have not even been capable of modelling (see chapter 3). However, the effect of high levels of co-location on the measured latency results have implications for how this modelling system can be applied. Future work should look at modelling the effect of high co-location on the executors in order to provide accurate predictions for densely packed physical plans, see section 6.3.4

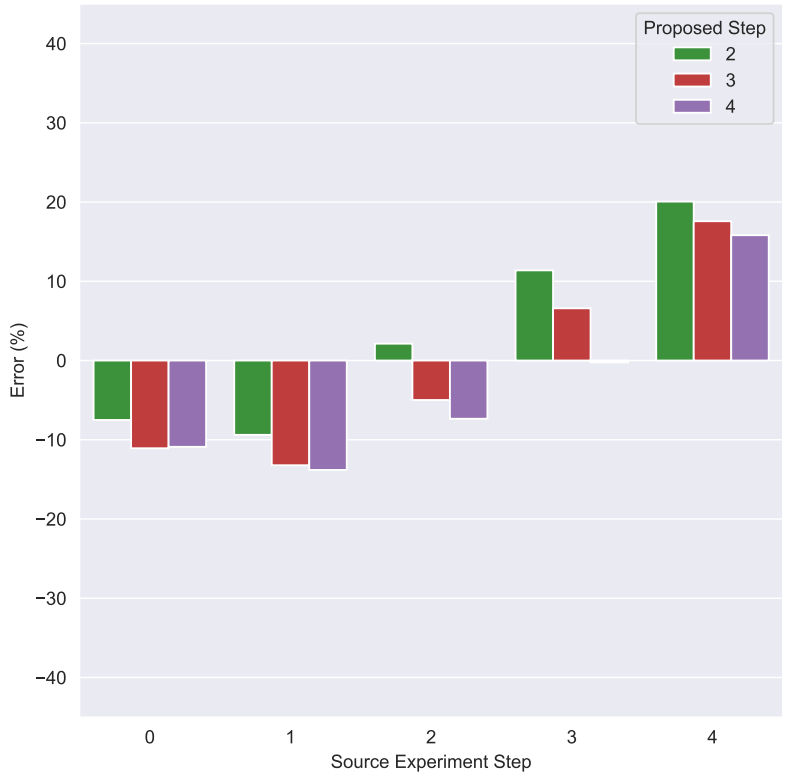

Figure 5.28: The relative error between the predicted weighted average ground truth latency and the average measured ground truth latency, using four worker nodes with two worker processes each.

for further discussion.

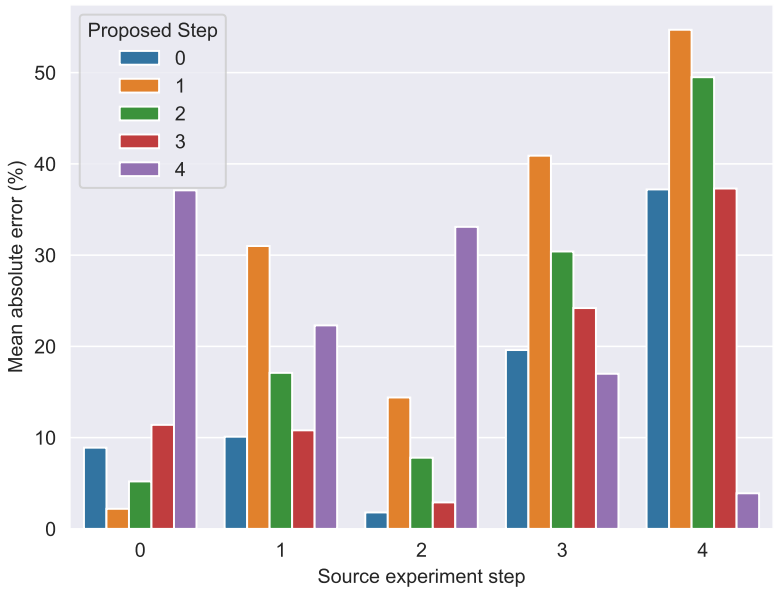

Figure 5.29: The percentage mean absolute error in the complete latency predictions for the fields-to-fields test topology, using two worker nodes with two worker processes each.

Complete latency The two worker node and four worker node results for the complete latency predictions are shown in figure 5.29 and figure 5.30 respectively. Comparing these two results sets shows that, as with the ground truth latency predictions, the results are less accurate for the two worker node experiment configuration. Again, this is likely a result of high co-location of the executors. It is worth noting that in the four worker node case, the complete latency errors are higher for the lower experimental steps than in the two worker node case. This is likely due to the higher level of remote connections. As the complete latency is highly susceptible to straggler tuples, slow network connections or resent packets will increase the measured complete latency. As the network transfer latency modelling is only based on a median round trip latency measurement between the worker nodes, the modelling will not account for these effects. This suggests that for more accurate complete latency modelling a different summary statistic, such as a higher percentile (75th or 90th), would be a more appropriate measure to use to model the expected remote transfer latency.

These results show that, provided high levels of executor co-location are avoided, the complete latency can be predicted with errors of less than 30% and in most cases less than 15% errors for consecutive fields grouped components with a one-to-one tuple ratio.

Multiplier topology

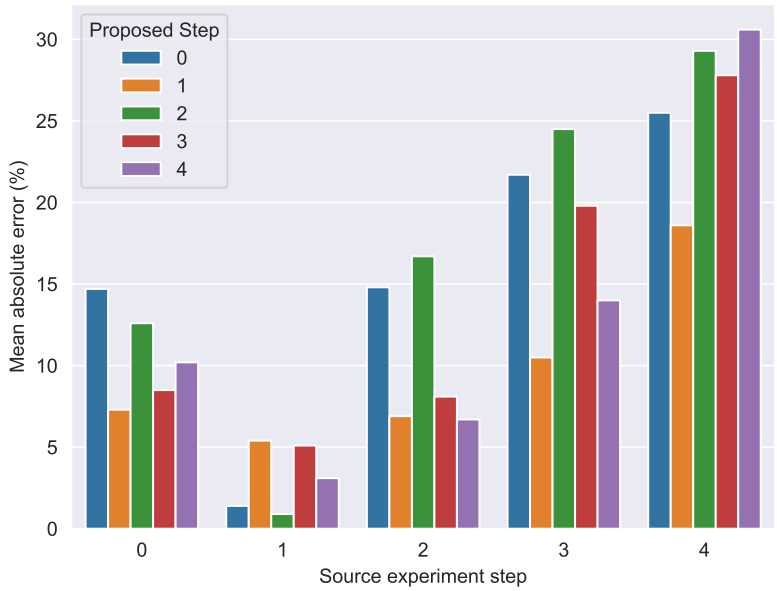

Figure 5.30: The percentage mean absolute error in the complete latency predictions for the fields-to-fields test topology, using four worker nodes with two worker processes each.

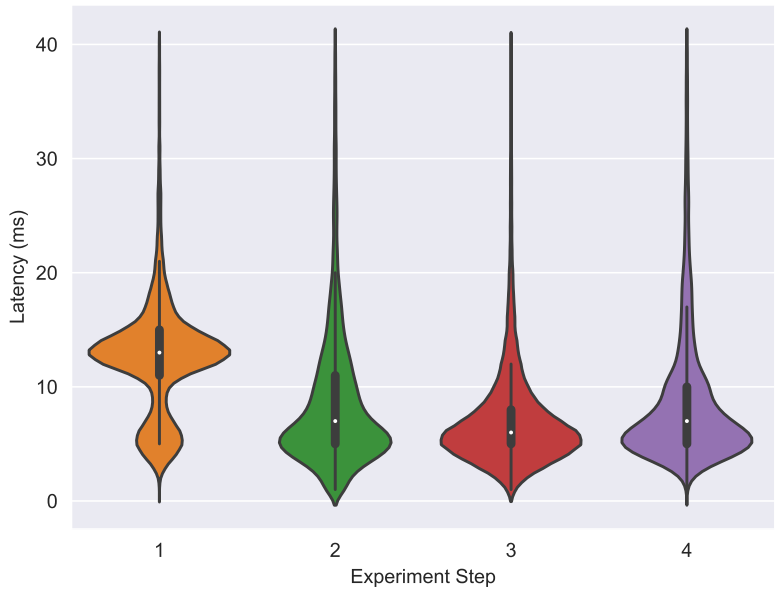

Figure 5.31: The measured ground truth latency for the multiplier test topology.

Ground truth latency Figure 5.31 shows the distribution of measured ground truth latencies for the multiplier test topology shown in figure 5.6. As with all the experiments described in this section, step 0 had only a single path and this did not start and stop on the same worker node and therefore is excluded. The multiplier test topology is formed of three components, where the middle component emits multiple tuples for every received tuple and the multiplier amount is randomly selected from a normal distribution with mean 10 and standard deviation of 1. The measured ground truth latency distribution of step 1 shows a different pattern to the other steps. This is likely due to this step only having two copies of each component. The 16 ts^{-1} input rate would then equate to an average of 80 ts^{-1} into the replicas of the final component. As the final component is performing the send operation to the Kafka message broker it has a relatively long service time, and so the high arrival rate causes increased queueing delays at that component. The Kafka client library implements its own internal batching and so this second peak may equate to tuples waiting within that batch before being processed. As the input rate into the topology is fixed, scaling up the components results in lower individual arrival rates at each of the replicas of the final component. As a result, the distributions of the experimental steps with higher parallelism more closely resemble those of the fields-to-fields tests topology shown in figure 5.27.

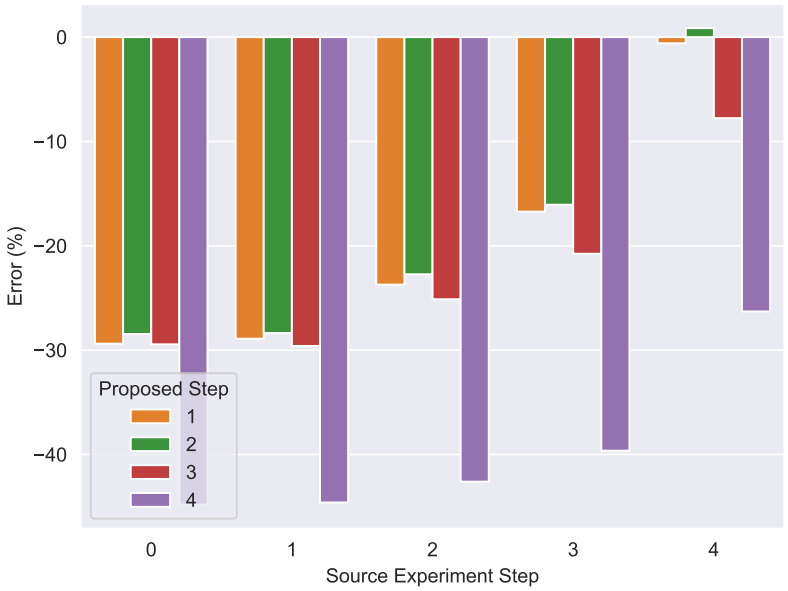

Figure 5.32: The relative error between the predicted weighted average and average measured ground truth latency for the multiplier test topology.

Figure 5.32 shows the relative error between the predicted weighted average and the average measured ground truth latency for the multiplier topology. This shows that the error for the experimental steps with low parallelism, with higher arrival rates into the final sink component, is greater compared to the steps with higher parallelism with lower

arrival rates into the final component. The error is predominately negative, indicating an under-prediction of the ground truth latency. As the error is larger for the experimental steps with lower parallelism, this is unlikely to be an issue with executor co-location and indicates that the modelling process is not capturing additional latency as a result of higher arrival rates. This effect could be a result of the Kafka client library used in the final component as it uses its own queue and batch send implementations, which may interrupt the executor processing under high load in a way that is not covered by the modelling process. Unfortunately, as the Kafka client is integral to the validation process, gauging performance without it being present was not possible.

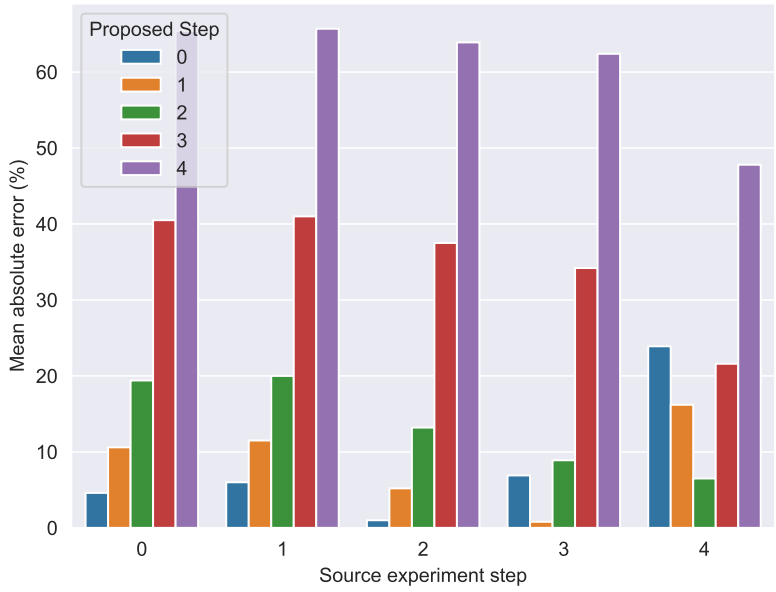

Figure 5.33: The percentage mean absolute error in the complete latency predictions for the multiplier test topology.

Complete latency Figure 5.33 shows the mean absolute error between the predicted complete latency (90th percentile) and the measured median complete latency. This shows that, whilst the errors for the lower experimental steps with low parallelism are 20% or less, the error in the prediction rises significantly for the experimental steps with higher parallelism. The reason for this can be seen by looking at figure 5.34, which compares the distribution of the measured ground truth latencies to that of the measured complete latencies from each experimental step. This shows that for steps 3 and 4 the complete latency is significantly higher than the ground truth latency. We would expect the complete latency to be slightly longer than the ground truth latency as it includes the additional delay at the Acker component. However, these distributions have median values at or above the 3rd quartile of the measured ground truth latency. Again this supports the notion that the complete latency is a measure of the worst case latency for a topology.

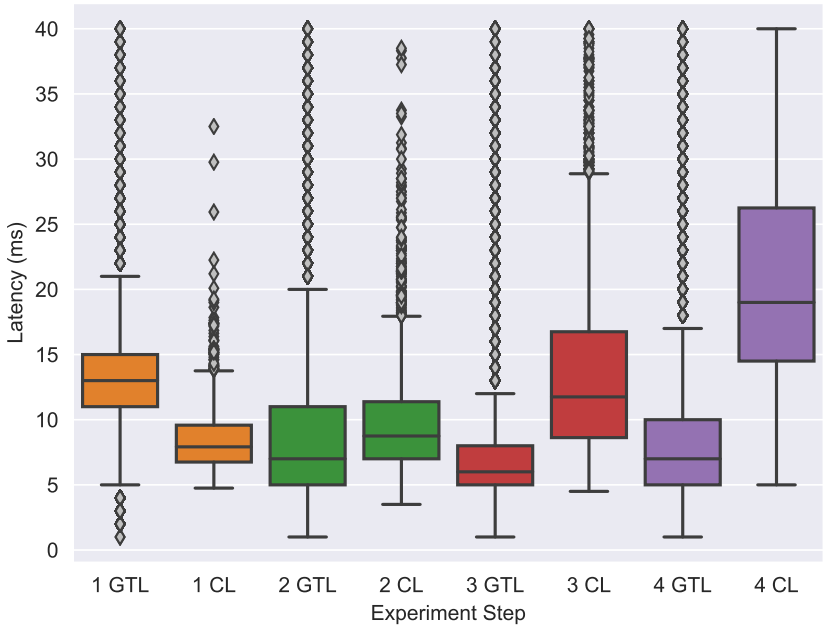

Figure 5.34: Comparison of the measured ground truth and complete latencies for each experiment step of the multiplier test topology.

As the number of component replicas within a physical plan increases, the number of network connections also increases. This in turn increases the chance of packets hitting a slow connection and skewing the average measured complete latency. It is also possible that co-location effects could further extend the measured complete latency compared to the predicted. As the complete latency prediction is based on average performance measures along each path, the further the measured complete latency moves from the average ground truth latency the worse the estimations for the complete latency will become. This suggests that complete latency predictions should be based on *worst case* metrics throughout the modelling process, 90th percentile service times and arrival rates for example, in order to better approximate the true behaviour of the complete latency.

Another interesting point to observe from figure 5.34 is that for step 1 the measured complete latency is lower than the ground truth latency. This is most likely due to the tuples arriving at the final component being acknowledged (stopping the complete latency clock) and the tuple processing being delayed in the Kafka client library due to the internal batching this library uses. In later steps, as described above, the arrival rate into the final Kafka sending component is reduced and so the batching effect is lessened.

Windowing topology

Ground truth latency Figure 5.35 shows the measured ground truth latency for the windowing test topology shown in figure 5.7. The distributions show that the median

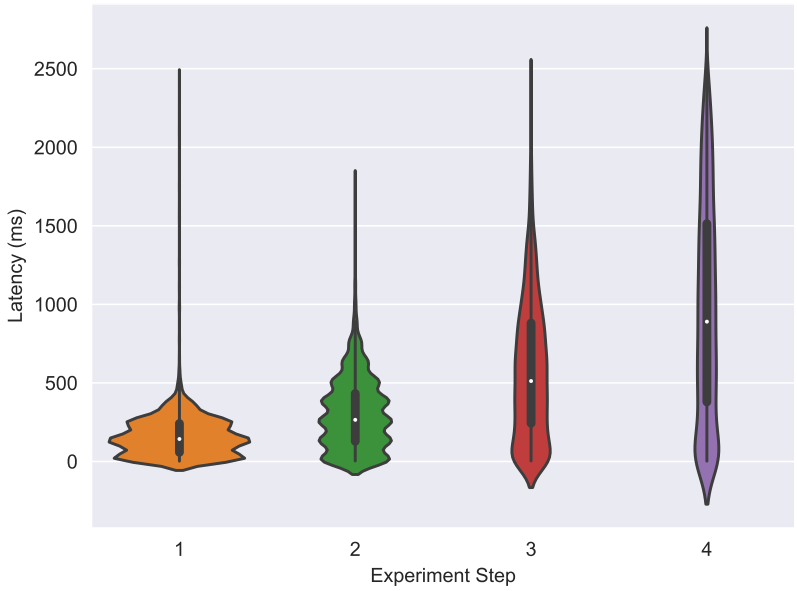

Figure 5.35: The measured ground truth latency for the windowing test topology.

ground truth latency (indicated by the white dot) increases with the parallelism of the experimental steps, as does the variance of the distributions. This increase is due to the fixed total arrival rate into the topology (16 ts^{-1}) and the fixed window size of 10 tuples. As the windowing component is scaled out, the fixed total arrival rate is divided between an increasing number of executors and therefore the arrival rate into the individual windowing executors drops. As the window is of fixed size and the arrival rate is dropping, it takes longer to complete each window. The increased variance is due to tuples falling either side of window boundaries and either being processed quickly or having to wait. With more copies of the windowing component there is more variance in the window boundaries (at each replica of the windowing component) and so more variation in the time tuples have to wait.

Figure 5.36 shows the relative error between the predicted weighted and the measured average ground truth latency for the windowing test topology. This shows consistent results, of 12% error or less, for all experimental steps with a slight increase in error when experimental steps with higher parallelism are used as metrics sources. For all steps the model is overestimating the latency, suggesting that there may be mechanisms within Storm's windowing implementation that speed up the processing compared to the assumptions of the model. However, these could also be errors due to averaging the source metrics and/or errors from other parameter estimations (service time, arrival rate, etc.).

Compared to the multiplier topology results shown in figure 5.32, the errors for the windowing topology are generally much lower. The arrival rate into the final component of the windowing topology (which sends the messages to the Kafka message broker) is

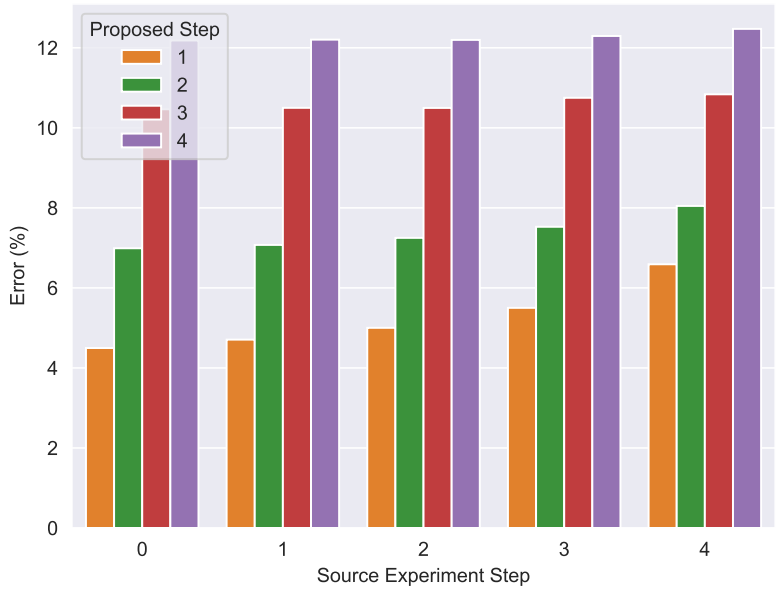

Figure 5.36: The relative error between the predicted weighted average and average measured ground truth latency for the windowing test topology.

significantly (around ten times) lower than the multiplier topology. As this seems to result in much better predictions, this does lend some weight to the theory that the under-prediction in the multiplier topology experiment may be due to the effects of high arrival rates into the Kafka sending component, resulting in additional delays that are not currently being captured by the modelling process.

Complete latency The complete latency results for the windowing test topology are shown in figure 5.37. These show consistent 60-70% errors regardless of the source or predicted physical plan. Whilst these are shown as absolute errors, as they are based on a mean of the complete latency predictions from all source spouts, they are in fact all under-predictions. The reason for this can be seen in figure 5.38, which shows the difference between the measured ground truth and complete latencies for each step of the windowing topology experiment. These show the complete latency distributions to be significantly higher than the measured ground truth latency in each case, much higher than would be expected by the additional delay in transfer and queuing at the Acker components. As discussed previously, the additional latency may be due to network delays skewing the complete latency average measurements. However, as the errors appear consistent across experiment steps, regardless of the number of remote connections, this suggests that network delays are not the main cause. Apache Storm's windowing API may be another source of error. The API will automatically acknowledge all tuples within an input window after that window has *completed*. The trigger for completing depends on the type of windows being used. For count based tumbling windows (like those used in this

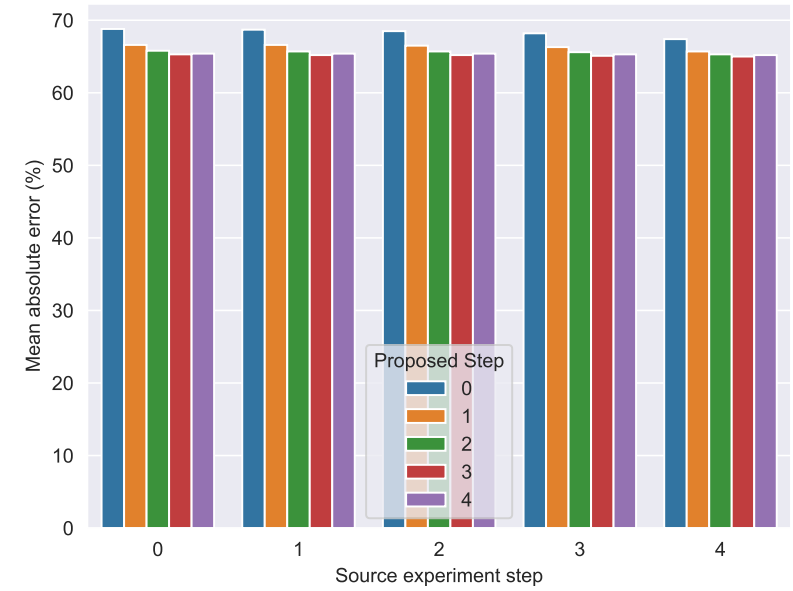

Figure 5.37: The percentage mean absolute error in the complete latency predictions for the windowing test topology.

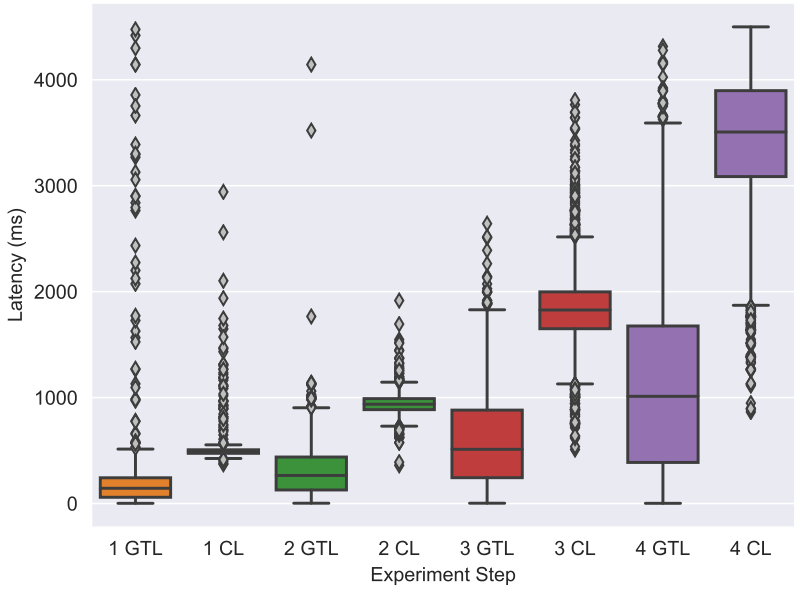

Figure 5.38: Comparison of the measured ground truth and complete latencies for each experiment step of the windowing test topology.

experiment) this completion should be almost instantaneous, however if this completion is delayed this would not be included in the model. As the error is consistent across experimental steps and the window size is fixed, this does suggest that some process within the windowing implementation, that is not captured in the modelling process, may be the reason for the error.

All-in-one topology

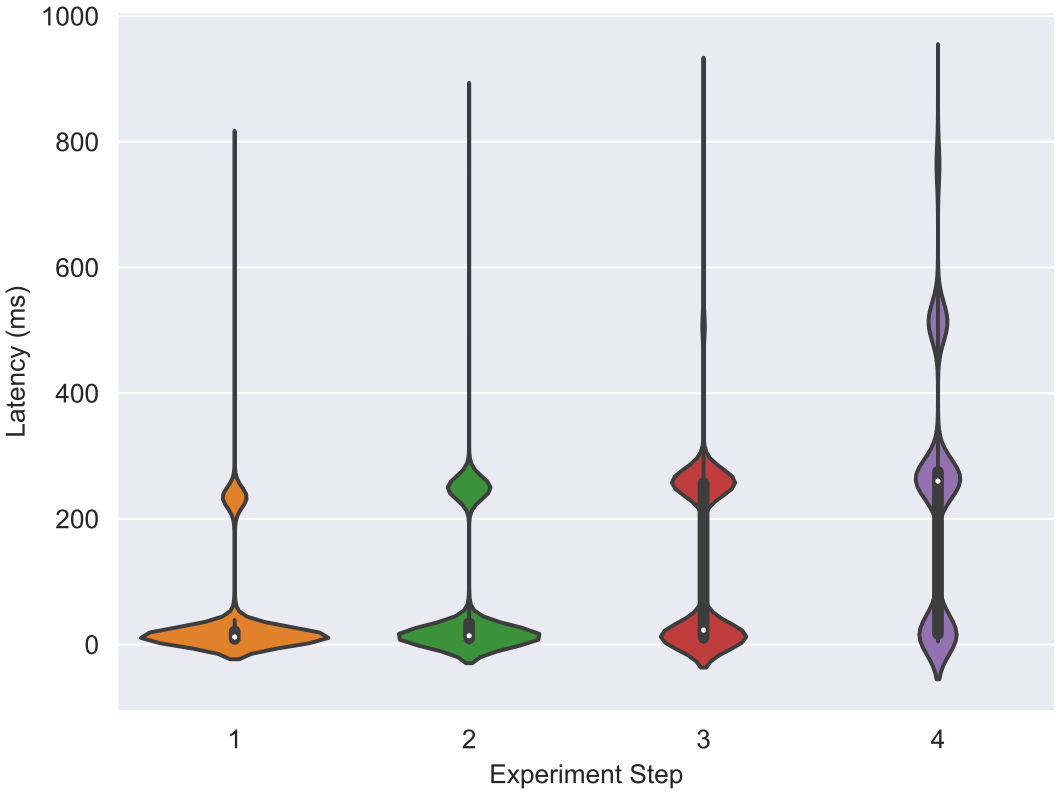

Figure 5.39: The measured ground truth latency for the all-in-one test topology.

Ground truth latency Figure 5.39 shows the measured ground truth latency for the all-in-one test topology shown in figure 5.8. For this experiment, the multiplier component multiplied the number of input tuples by a random integer drawn from a normal distribution with mean 20 and standard deviation of 1.0. The windowing component had a tumbling count based window of size 10. The final two components had fields grouped connections. Like the windowing topology measured ground truth latency distributions shown in figure 5.35, the average latency and the variance in the distribution rise with increasing parallelism of the experimental step's physical plan. As in the windowing topology case, this can be attributed to the fixed arrival rate across experimental steps leading to lower individual arrival rates into the executors of the windowing component for experimental steps with higher parallelism. These lower arrival rates and the fixed window size lead to

longer periods waiting for windows to be filled.

The distributions shown in figure 5.39 exhibit a clear multi-modal pattern and these separate latency peaks are an interesting feature. The length of the inter-peak intervals in each distribution remains constant, approximately 200ms, regardless of the parallelism of the physical plan. The only common factors in the configuration of the experimental steps for this topology are the multiplier component settings and the tumbling window size. Indeed, these inter-peak periods seem to correlate with the process latency reported by Apache Storm for each windowing executor. As discussed in section 4.13.1, the process latency reported by Storm for windowing executors equates roughly to the time taken for a window of tuples to be fully processed. These peaks, therefore, seem to equate to discrete window processing times.

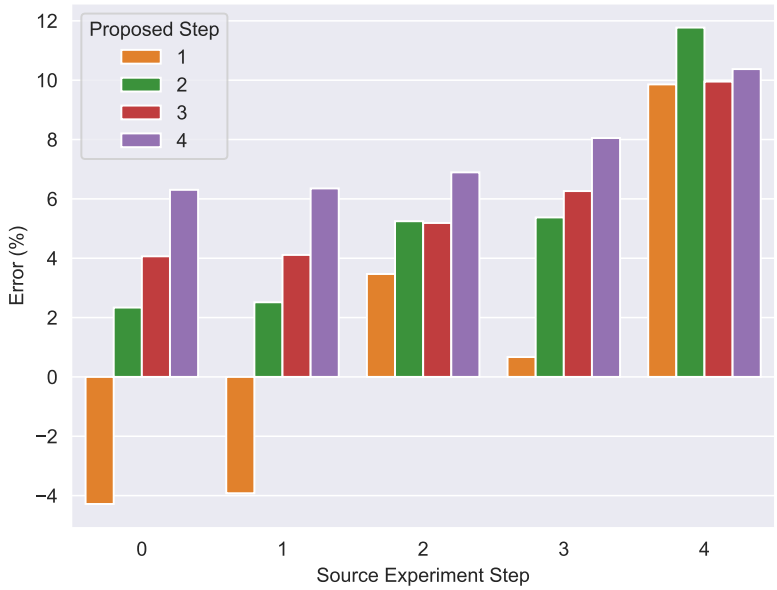

Figure 5.40: The relative error between the predicted weighted average and average measured ground truth latency for the all-in-one test topology.

Figure 5.40 shows the relative error between the predicted weighted and the measured average ground truth latency for the all-in-one test topology. These results show prediction errors of typically 10% or less. In this test topology the arrival rate into the final Kafka sending component is much lower than in the multiplier topology case and has much better accuracy. This gives further weight to the theory that additional processes within the Kafka Client API, when under high load such as in the multiplier topology experiment, are not being captured by the modelling process.

Complete latency Figure 5.41 shows the absolute errors between the predicted and the measured complete latency for the all-in-one topology. This shows significant errors of approximately 80% or higher. As with the windowing topology experiment, these

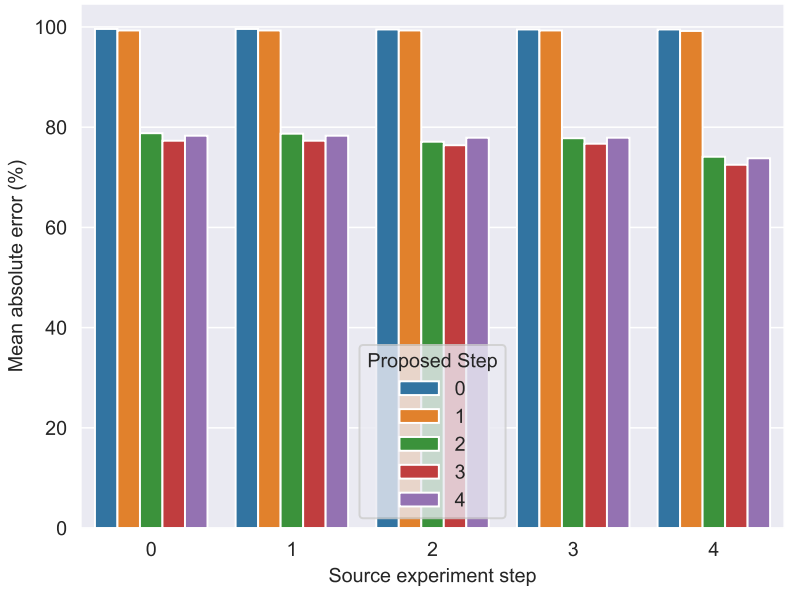

Figure 5.41: The percentage mean absolute error in the complete latency predictions for the all-in-one test topology.

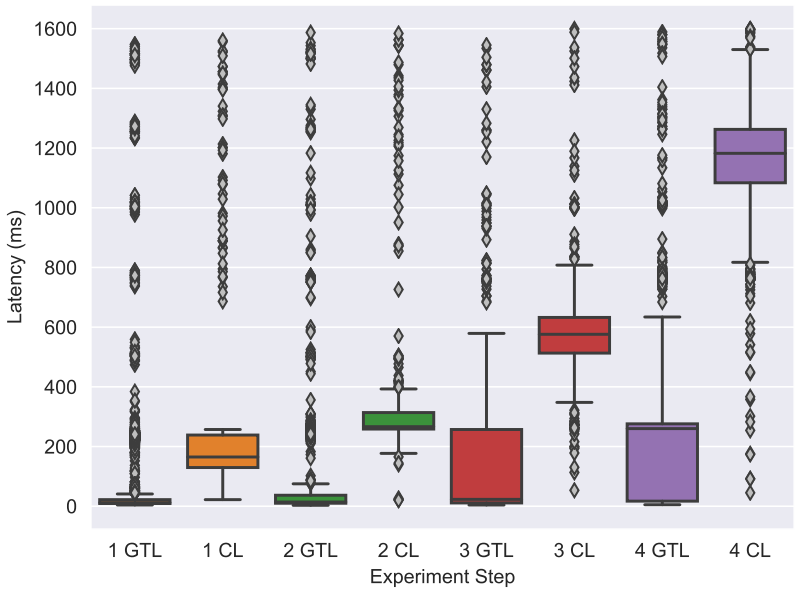

Figure 5.42: Comparison of the measured ground truth and complete latencies for each experiment step of the all-in-one test topology.

estimations are all under-predictions. The comparison, shown in figure 5.42, between the measured ground truth and complete latency distributions, shows the latter being significantly higher. The difference between them is even more pronounced than in the windowing topology case. This can be explained by the presence of a multiplying component being followed by a windowing component in the all-in-one topology. All new tuples created in the multiplier component are anchored to the original input tuple; these multiple child tuples will then be sent to the windowing component. If one of these child tuples arrives into a later window than the others and therefore waits for one or more additional window periods to pass, this will extend the completion of the tuple tree for the original input tuple. This kind of delay is not captured in the modelling process and likely explains some of the prediction error for the complete latency.

Join-split topology

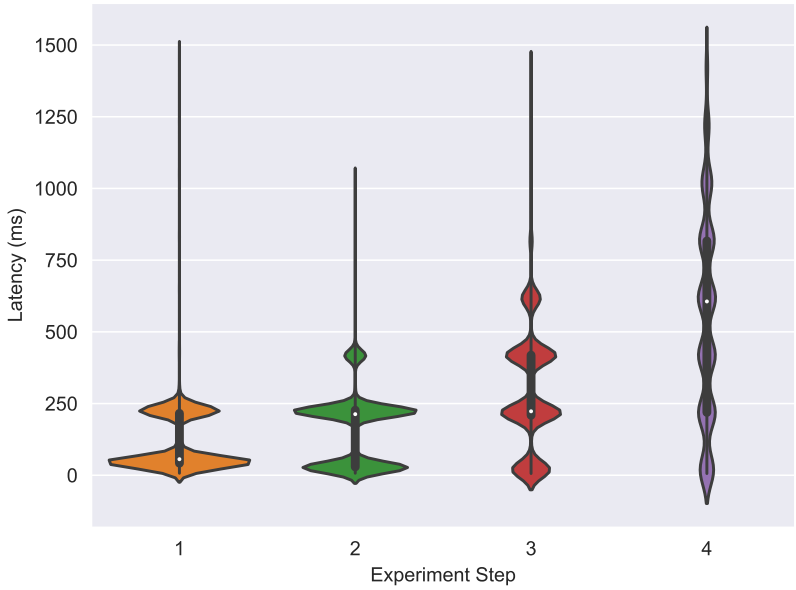

Figure 5.43: The measured ground truth latency for the join-split test topology.

Ground truth latency Figure 5.43 shows the measured ground truth latency for the join-split test topology shown in figure 5.9. This distribution shows a similar pattern to that of the all-in-one topology shown in figure 5.39. This is to be expected as both these topologies contain a multiplying component followed by a windowing component (in this case the stream joining bolt). Again the inter-peak period is the same, regardless of the parallelism of the experimental step's physical plan, suggesting this is linked to the common tumbling window count size. As the processing is identical and all windows have the same number of tuples, the window processing time for each step will be similar.

Figure 5.44 shows the relative error between the predicted weighted and the measured

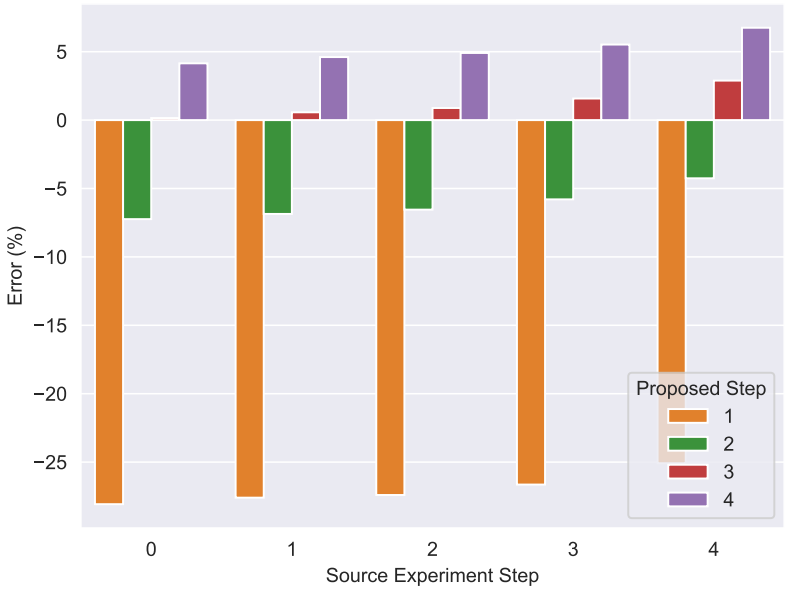

Figure 5.44: The relative error between the predicted weighted average and average measured ground truth latency for the join-split test topology.

average ground truth latency for the join-split test topology. These results show good accuracy, with errors of 7% or lower for predicting steps 2 to 4 using all other steps as sources. Predictions of the performance of step 1, however, show an under-prediction of approximately 25%. This mirrors the level of under-prediction seen in the multiplier test topology and may be similarly explained by high arrival rates into the sending components. In the case of the join-split test topology, the overall input rate was 25 ts^{-1}. The topology (shown in figure 5.9) included an approximately 10 times multiplier and a pass through component before the joining component. In step 1 this meant that the combined arrival rate at the joining component was over 275 ts^{-1}. The joining component windows that stream into batches of 20 tuples and sends two tuples to each sending component (one from each input stream to ensure we have enough paths samples). This means the arrival rate into the executors of the final Kafka sending component was high compared to the other experiments described in this chapter, and close to those of the experimental steps with low parallelism in the multiplier topology test. As the input rate was fixed, the doubling of parallelism for the next step halved the arrival rate into the joining component and the under-prediction dropped to approximately 5%. These results suggest that there are additional sources of latency occurring in either the Windowing API or Kafka Client library, when subjected to higher arrival rates.

Complete latency Figure 5.45 shows the absolute errors between the predicted complete latency and the measured for the join-split topology. Similar to the all-in-one topology complete latency results, shown in figure 5.41, this shows significant errors of approximately

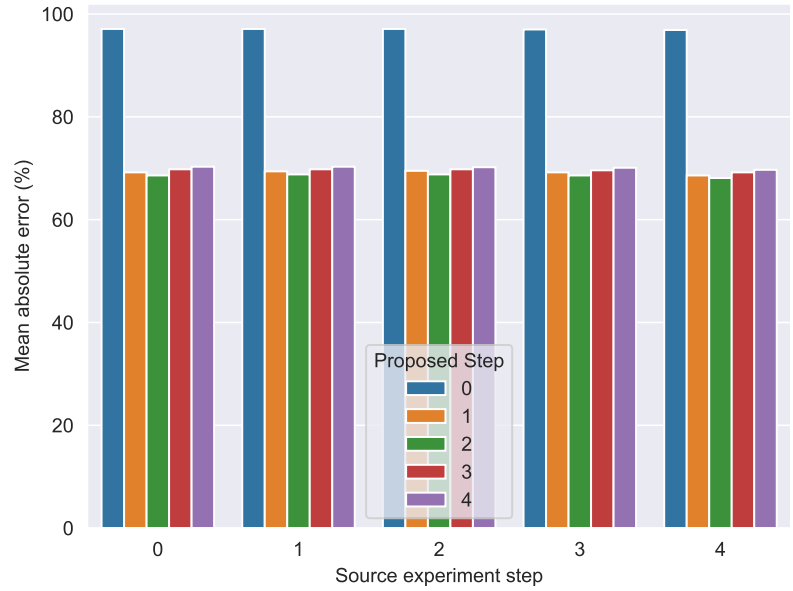

Figure 5.45: The percentage mean absolute error in the complete latency predictions for the join-split test topology.

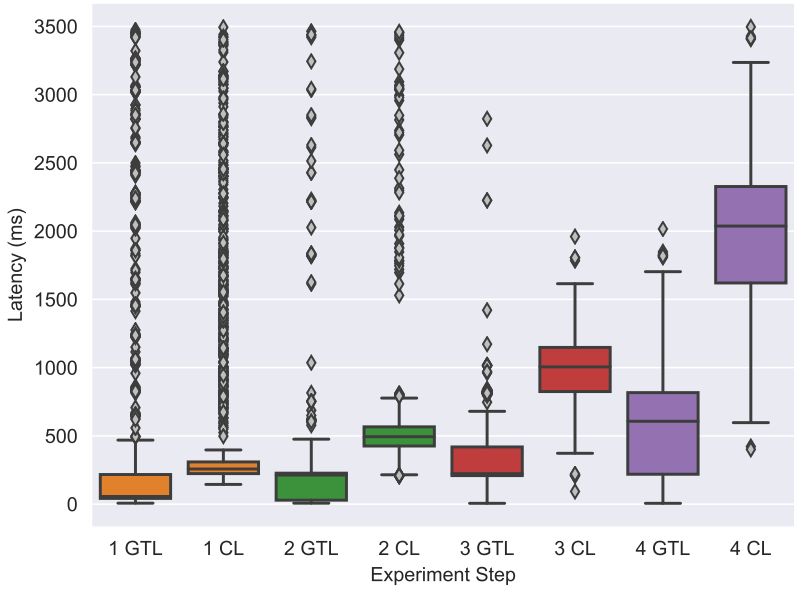

Figure 5.46: Comparison of the measured ground truth and complete latencies for each experiment step of the all-in-one test topology.

70% or higher. As with the windowing and all-in-one topologies, these estimations are all under-predictions. The comparison shown in figure 5.46, between the measured ground truth latency and complete latency distribution, shows a similar situation to the all-in-one topology and, as the join-split topology has a similar multiplier component feeding into a windowing component, the discussion above for that result is applicable here.

5.8 Summary

5.8.1 Arrival rate

The arrival rate prediction methods, described in section 4.8, were evaluated in three sections: stream routing probabilities, I/O ratio predictions and the arrival rate predictions themselves:

Stream routing probabilities The stream routing predictions show good results with errors of typically less than 10% even when consecutive fields grouped connections where used.

I/O ratios The I/O ratio predictions show low errors (<1%) for linear topologies where components have a single input and output stream and also for multi-stream components where the routing behaviour is simplistic (<6%). However, for more complex routing behaviour the error can be much higher (30-40%) where the source physical plan lacks sufficient variety to predict the routing for more complex physical plans. Conversely, more complex physical plans can predict the complex routing behaviour of less complex physical plans with a lower error (<15%).

Arrival rate The arrival rate predictions, tested against the fields-to-fields and join-split (with simple routing) show low errors of less than 2.5%.

The results of the evaluation show that the prediction methods are able to predict relatively complex routing patterns as the result of significant changes to a topology's physical plan. There are some limitations of the I/O ratio predictions for complex relationships between multiple input and output streams. These may be better modelled using non-linear techniques.

5.8.2 Service time

The service time predictions were shown to produce errors between 20 and 300% depending on the parallelism of the source physical plan used. The results showed that high co-location of executors within worker processes can lead to thread pausing which extends the measured service time. This leads to over-estimations of the executor service times when predicting the performance of a scale down proposal and under-estimating the service time

for scale up proposals. In order to improve the service time predictions a more advanced modelling method will be required to account for the effect of changing co-location levels. The effect of the service time errors is most significant when the predicted service time leads to a service rate close to the arrival rate, and so is more of an issue for scale down proposals than scale up.

5.8.3 Tuple input list size

The test version of Apache Storm was modified to record the average number of tuples within the objects arriving at each Disruptor queue. This allowed the predicted tuple input list size to be validated and showed that for most situations the prediction error was below 30%. However, higher errors were seen in scale down situations. Due to the fixed overall arrival rate into the test topologies, scale down proposed physical plans (with lower component parallelism) meant higher individual executor arrival rates. It was suggested that the higher errors could be due to batching within the network library that occurs under higher load. A better approach to reducing prediction error would be to measure the input batch size in the default metrics and use this to produce prediction models for proposed physical plans.

5.8.4 End-to-end latency

The evaluation of the end-to-end latency predictions focused on two metrics: the ground truth latency and the complete latency. The ground truth latency is measured by the evaluation test system and is the time from when a message was received in the test topology spouts to when it is processed in the final sink component. The complete latency is a metric provided by Apache Storm and is a measure of the time from when a spout tuple is first acknowledged by the Acker component to when all child tuples of that source tuple are processed. See section 2.13.2 for more details.

Ground truth latency

The results for the ground truth latency predictions show that generally errors are below 20% and typically 10% or lower for the majority of test topologies. The exception to this is for the multiplier topology with high arrival rates into the final sending component. It was theorised that the under-prediction (typically around 30%) was due to an unaccounted for behaviour in the Kafka client library, used in the final sending component, performing some batching action under higher load which was not present in the other test topology under lower load.

Generally, using a physical plan with high parallelism and executor co-location resulted in worse prediction performance. This over-prediction of latency was attributed to thread

pausing artificially extending the service times used as source metrics for the other steps with lower parallelism. Similarly, when predicting the performance of highly co-located physical plans using metrics from physical plans with lower co-location, higher errors were seen.

Complete latency

The results for the complete latency predictions for the various experiments show significant errors. The linear fields-to-fields topology with one-to-one I/O ratios gives the best accuracy, however the presence of multiplier or windowing components can cause the complete latency distributions to be significantly higher than the ground truth latency. The modelling process is built around predicting the average end-to-end latency of each path within the topology tuple flow plan. In order to model the complete latency we use the 90th percentile of these predictions to account for the worst case nature of the complete latency. The results in this evaluation show this is not sufficient, and that in order to model the complete latency the reasonable worst case of all parameters (90th percentile service time and arrival rate, for example) should be used. However, even using this may not be enough for accurate modelling of the complete latency in cases such as the all-in-one topology where a multiplying component is followed by a windowing component.

The design of the complete latency metric, as described in section 2.13.2, does allow the time required to process each input tuple to be gauged. However, it also makes it a poor measure of overall topology performance and highlights how Apache Storm would benefit from additional performance measures. This topic is discussed further in section 6.3.1.

5.8.5 Factors undermining prediction accuracy

In the discussion above we identified several key factors affecting the accuracy of our end-to-end latency predictions:

Non-linear input/output relationship In the calculation of the I/O ratios, the least-squares regression approach was shown to be sufficient for simple relationships between input and outputs streams. However, complex, non-linear relationships with time based windows and complex routing behaviour were not well covered by this approach.

Co-location effects The multi-threaded nature of Storm's systems meant that the simple weighted average approach, used to predict the expected service time of executors in a proposed physical plan, resulted in poor prediction accuracy (see section 5.5). The error was larger for scale down situations, going from physical plans with high levels of executor co-location to proposed physical plans with lower parallelism. The effect of this service time prediction error was particularly pronounced when the service

and arrival rates were relatively close.

Input tuple list size The input tuple list size evaluation (see section 5.6) showed that whilst errors were typically 30% or less, some situations led to errors of 80% or more. This suggests that the current modelling approach does not include all the processes in the tuple flow and that it may be better to explicitly measure this value in the default Storm Disruptor queue metrics.

External libraries In the evaluation of the end-to-end latency predictions, significant under-prediction of the latency was seen when the executors of the sending component were subjected to high arrival rates. It was theorised that the missing latency could be due to additional processes in the Kafka client code, which is known to perform additional batch sending in separate threads. This was only an issue for the ground truth latency predictions as this aspect would not be included in the complete latency tuple flow. However, it does highlight a need to have a way to account for arbitrary code processes in the tuple flow plan.

5.8.6 Conclusion

Despite the sources of error listed above, for most experimental configurations the error in the end-to-end latency predictions was 20% or lower and in cases where adjacent experimental steps were predicted (the difference in parallelism was only double or half of the source step) these errors were typically 10% or lower. These results are equivalent to, or better than, the best performing machine learning based approaches discussed in section 3.2.2, but using a fraction of the required training data of those approaches. Each of the experimental predictions in this chapter used only 20 minutes of metrics data, compared to the many hours and multiple topology configurations required by the machine-learning based approaches.

Chapter 6

Discussion

6.1 Thesis Summary

6.1.1 Chapter 1 — Introduction

Most distributed stream processing systems (DSPSs) provide functionality to scale their stream processing topologies, the directed graph of logical operators that process a continuous stream of data packets (tuples). However, automatic scaling is not possible except in Apache Heron. This forces users of these systems to perform a *scaling decision loop*; deploying the initial topology physical plan, waiting for it to stabilise, analysing the metrics, altering the physical plan to improve performance and repeating this cycle until the desired performance is reached. The deployment and stabilisation of large production topologies can take hours, which means the process of finding the correct topology configuration to meet a given latency service level agreement (SLA) can take many days to complete. This long scaling period often means that, once a topology configuration is found to meet the SLA, the scaling process is not repeated to adapt to new incoming workloads. To speed up this process a way to short circuit the scaling decision loop is needed. We propose a performance modelling system that is able to assess the likely performance of a proposed physical plan before it is deployed, so that a viable plan can be iterated to without paying the time penalty of deployment. Such a system would not only provide faster convergence towards a viable plan, but would also allow the effect of future workloads on the current topology physical plan to be assessed and also for multiple proposed physical plans to be compared in parallel.

6.1.2 Chapter 2 — Apache Storm architecture

One of the most powerful and widely used DSPSs at the start of this research was Apache Storm. This was chosen (see section 2.1) as the test DSPS for this research project and

its internal structure and operations were studied in detail in chapter 2. The copies of each topology's components are distributed (scheduled) onto a series of worker processes (see section 2.4.3). These host a number of executors (see section 2.4.1) which run the user defined code (tasks — see section 2.4.2). Each executor had a series of internal queues (see section 2.7) which have complex operational behaviour. The flow of tuples through the various elements of the Apache Storm system was mapped (see section 2.8) and other operations such as guaranteed message processing (see section 2.9), rebalancing the topology's physical plan (see section 2.11) and windowing tuples into batches (see section 2.12) were also explained in detail.

6.1.3 Chapter 3 — Related work

The previous research into DSPS auto-scaling systems was reviewed in chapter 3. Most of the early research in this area focused on replacing the human user in the scaling decision loop by triggering some form of optimising physical plan scheduler when a performance threshold was reached (see section 3.1). Whilst these proposed systems provided a way to remove human supervision, they did not reduce the time taken to find a valid physical plan and most of this research was focused on the performance of the scheduler implementations.

More recently, several studies have focused on using some form of performance model to speed up physical plan selection. Some of these were based on a queuing theory approach (see section 3.2.1), using various queuing models to approximate the operators of a stream processing topology. However, none of the studies covered key based routing or the low-level batch processing logic present in most DSPS implementations. Whilst these studies did utilise performance modelling, most (Lohrmann et al., 2015; De Matteis & Mencagli, 2016; Vakilinia et al., 2016) did not evaluate the modelling accuracy by comparing predicted latencies with measurements from an implemented system. The few that did (Fu et al., 2015) had errors of 30-90% and did not adequately explain the reasons for such a large difference.

As well as a queuing theory approach, several studies used machine learning techniques to train models on metrics from a running topology (see section 3.2.2). These reported much better accuracy than the queuing theory models, however details on the time taken to train the models and the amount of training data required are rarely provided. Where they were, hours of data were needed to predict the performance of simple linear topologies.

6.1.4 Chapter 4 — Modelling approach

The performance modelling approach, laid out in chapter 4, aimed to provide a comprehensive modelling system that was able to predict a wide range of topology designs with only a small amount of observable data. This obviously ruled out using machine-learning based

methods and therefore a queueing theory based approach was used. Analysis of the internal queues used by the executors in the Apache Storm system showed that most common queueing theory models would not be applicable. Instead, a discrete-event simulation (DES) approach was used to simulate the Disruptor queue and executor user logic thread (ULT) (section 4.2.1). This required several parameters of a proposed physical plan to be predicted:

- The routing probabilities (section 4.5)
- Input to output (I/O) ratios (section 4.7)
- Arrival rates (section 4.8)
- Service times (section 4.10)
- The number of tuples in each input batch arriving at the executor receive queues (ERQs) (section 4.12)

In addition to the delay at the executors, the transfer time between remote worker nodes needed to be predicted (section 4.11.1), as well as the expected delays due to the other elements tuples would pass through on a physical path. These included windowing components and elements of the worker process (section 4.13). Finally, we showed how the end-to-end latency can be estimated once all the above parameters are estimated.

6.1.5 Chapter 5 — Evaluation

Chapter 5 reports the results of testing several topology designs from simple linear to complex multi-path topology (see section 5.3). The results showed that the modelling process could predict arrival rates for proposed physical plans with a high degree of accuracy (section 5.4). Service times, however, were much harder to predict due to the effect of pausing as a result of executor co-location affecting the reported service time (section 5.5).

In order to better gauge the end-to-end latency, without the drawbacks of the complete latency metric (see section 2.13.2), a custom *ground truth latency* metric was measured. The results for the ground truth latency predictions (section 5.7.3) show that errors were typically below 20% and often 10% or lower for the majority of test topology configurations. The exception to this is for multiplier topologies with high arrival rates into the final sending component, which may have been a result of unaccounted-for behaviour in the communication library used to send messages in the final sink components.

Generally, using a physical plan with high parallelism and executor co-location, as a source of metrics for predicting plans with lower parallelism (scale down), resulted in worse prediction accuracy. This over-prediction of the end-to-end latency was attributed to thread pausing artificially extending the service times used as source metrics for the other steps which had lower parallelism.

The results for the complete latency predictions showed significant errors. The presence of multiplier or windowing components can cause the complete latency distributions to be significantly higher than the ground truth latency. The modelling process is built around predicting the average end-to-end latency of each path within the topology. In order to model the complete latency we use the 90th percentile of these predictions to account for the worst case nature of the complete latency. The results in this evaluation show that this is not sufficient, and that in order to model the complete latency the reasonable worst case of all parameters (90th percentile service time and arrival rate, for example) should be used. However, even using this may not be enough for accurate modelling of the complete latency in cases such as the all-in-one topology, where a multiplying component is followed by a windowing component, due to increased likelihood of child tuples being placed in later windows.

6.2 Summary of Contributions

The aim of this thesis was to investigate the creation of a performance modelling system for DSPS topologies in an effort to shorten the time taken to find a physical plan which meets a given end-to-end latency requirement. Broadly, this goal has been achieved and our modelling system improves on previous work in this field in several key areas:

Improved accuracy Many of the studies discussed in section 3.2 did not give details of the accuracy of their performance modelling systems. However, those that did gave errors of 30%-90% for the simple, linear, shuffle-connection-only topologies that they tested. The modelling system evaluation in chapter 5 showed that, for simple topologies, we can achieve errors below 10% even when predicting a physical plan with eight times the parallelism of the source physical plan, using only 20 minutes of metrics data.

For more complex topologies, against which most previously proposed modelling methods have not been tested, our system was able to achieve errors of below 30% for most topology configurations.

Fields (key) based connections Many of the queueing theory based modelling approaches, discussed in section 3.2.1, used shuffle (random or round-robin) based connections only. These previous studies made no attempt to predict the likely routing pattern through a proposed physical plan with key based connections or predict the effect unbalanced arrival rates would have on the topology's performance. As section 5.4 shows, we have not only been able to predict the arrival rate for topologies with one key based connection but also for those with consecutive key based connections, to a high degree of accuracy (typical errors of less than 5%).

Section 5.7.3 shows that we can generally predict the latency of proposed physical plans, for topologies with key based connections, with errors of 20% or lower and typically less than 10%.

Diverse component types This research covered test topologies with multiple types of streaming components: one-to-one, multiplying and windowing. All the previous queueing theory based research, discussed in section 3.2.1, dealt with simple one-to-one streaming components only. Despite the issues with multiplier components, highlighted in section 5.7.3, typically errors of 30% or less can be achieved for multiplier and 10% or less for windowing and one-to-one component topologies across a wide range of topology configurations.

The machine learning studies, discussed in section 3.2.2, do cover more diverse topology component types than the queueing theory based studies. However, the results of our modelling system are comparable for the linear, shuffle connection only topologies used by those studies that reported modelling accuracy.

Distributed test environment Many of the test environments for the previous work discussed in section 3.2.1 used a single worker node and/or did not take network transfers or whole system tuple flow into account. By contrast, our system was tested on clusters with multiple worker nodes and took the effect of network transfer latency into account in the end-to-end latency predictions.

Small amount of input data As this work is based on a queueing theory approach, many of the comparisons discussed above are related to previous queueing theory studies. However, this work has shown comparable results to that of the machine learning based approaches discussed in section 3.2.2, whilst covering a more diverse set of component types and only using 20 minutes of input data.

Using our modelling approach, a DSPS auto-scaling system could assess a proposed physical plan from any Storm scheduler implementation and provide an estimate of its end-to-end latency before the physical plan was deployed. This allows the scaling decision loop to be shortened and the topology to be scaled much faster upon first deployment and facilitates faster rebalancing in the face of changing workload.

6.3 Future Research

As highlighted in section 5.8.5, there are several factors which affect the accuracy of the current modelling system and would benefit from additional research. These factors, along with ways in which the modelling systems could be expanded, are discussed in the sections below:

6.3.1 Additional metrics

Complete latency

Section 2.13.2 describes how the complete latency, Storm's measure of end-to-end latency, is calculated and also highlighted how it is a *worst case* latency measure. Because of the way the complete latency is implemented, it is highly susceptible to outlier measurements skewing the recorded metrics, as can be seen in the results shown in section 5.7.3. This makes it a poor measure of the average or typical performance of a topology. However, it is common for performance SLAs to be defined in terms of a high percentile rather than an average; often service up-time and latency targets are set in terms of the "five nines" or the 99.999th percentile. In this regard the complete latency is the appropriate default Storm metric for defining SLAs, however it is even more of a worst case measure than the "five nines" latency and could be considered overly restrictive.

The complete latency prediction results discussed in section 5.7.3 show that our attempt to predict the complete latency by using the 90th percentile of the predicted path complete latencies did not yield accurate results. We were under-predicting the complete latency by some margin. As discussed in chapter 5, a more accurate prediction of the complete latency would probably require using the worst case version of every modelling parameter. However, a better approach would be to change the Storm complete latency measurement to include more detail.

Instead of a simple average across every metric bucket period (see section 2.13), descriptive statistics (variance, median, mean, quartiles) for the complete latency distribution of each metric bucket period could be produced. Furthermore, there are many statistical approaches to approximate the overall population statistics from streaming samples and these could be investigated with the aim of describing the complete latency distribution for a topology whilst minimising the memory and processing overhead. Such information would allow not just the worst case complete latency to be reported but also other statistical measures, which would give users (and performance modelling systems) more tools to assess the performance of their topology designs.

Additionally, a measure such as the ground truth latency (see section 5.7.2), could easily be added to Storm to provide latency tracing data as part of the default metrics. This would allow integration with standardised, open source, latency tracing implementations such as the Open Tracing project[1].

[1] https://opentracing.io/

Queue metrics

Section 5.6 discusses the results of the evaluation of the incoming tuple list size prediction methods detailed in section 4.12. These show that whilst the error in these predictions was typically below 30%, in some situations it can be as high as 80%. The method for predicting the incoming tuple list size is quite complex and involves multiple stages, however estimates of these values are required because Storm does not provide a measurement of these values by default.

The complexity of the prediction methods for these values could be reduced if measures of the incoming tuple list size were added to the default Disruptor queue metrics. Indeed, a measure of these values was added to the version of Storm running on the test cluster in order to allow validation of the prediction method. This metric would also benefit (as with the improvements to the complete latency metric discussed above) from the inclusion of descriptive statistics for the distribution of tuple counts for the objects entering the Disruptor queues. This is important as, even with this metric being added, there will still need to be a prediction of this value for the ERQs in the proposed tuple flow plans and the more data available, the easier this prediction will be.

Metric distributions

Estimation of both complete latency and Disruptor queue metrics would be more accurate if more information about the distributions of their measurements within the metric bucket periods was available. In general, the performance modelling of DSPS topologies would benefit greatly from this information being available for all performance metrics. However, delivering this level of detail, without overly increasing the memory and processing overhead and while maintaining statistical validity, is an interesting research challenge.

6.3.2 Workload prediction

Section 4.3 described how the expected incoming workload into a topology is a key factor in its performance. Any performance modelling system would need to be able to predict what the expected workload into a proposed physical plan would be in order to properly gauge its suitability. Workload prediction is also a key feature of the pre-emptive scaling discussed in section 1.3.3.

The prediction of network traffic is an entire field of study in itself (Huifang Feng & Yantai Shu, 2005; Herbst et al., 2017). Whilst the modelling system was designed to accept a workload prediction, unfortunately there was not time to properly investigate this area. Future research could focus on which of the many approaches in the literature are most appropriate for DSPS workloads. Some preliminary work in this area was performed by the author during their internship at Twitter, resulting in a publication (Kalim et al.,

2019) in collaboration with the Twitter Real time compute team. More details of this work can be seen in appendix D.

6.3.3 Routing key distribution

In the prediction of the routing probabilities for fields grouped (key based) connections, discussed in section 4.5, we have assumed that the field value (key) distribution is fixed and will remain the same as in the metrics from the source physical plan. This is a reasonable assumption over the short term. However, it cannot be assumed for all situations, particularly if a longer prediction time horizon is required. For example, if a pre-emptive scaling system is predicting several hours in the future, in order to find a physical plan which will maintain performance in the presence of a predicted workload spike.

The contents of the incoming tuple stream to the topology and those within the tuple flow plan may display seasonal variation in the routing probabilities of field grouped connections. For example, certain sensors could be active at different times of day or measurements with a particular key may be more likely in the evening than in the morning. Time series prediction methods, similar to those discussed in section 6.3.2, could be investigated in order to provide a more robust routing probability prediction.

6.3.4 Service time prediction

Section 5.5 showed that the current service time prediction method, detailed in section 4.10, can suffer from significant errors when using metrics from source physical plans with high levels of co-located executors. It was hypothesised that the multi-threaded nature of the worker processes meant that the wall-clock latency measures reported by Storm included periods where the executors were paused, whilst other executors were running. Simply using a weighted average of the source physical plan service time metrics was found to be insufficient for accurately predicting this complex behaviour.

A method to predict the expected service time in the presence or absence (depending on the level of co-location) of this multi-threaded pausing will need to be investigated in order to improve the accuracy of the modelling approach. This is particularly important for predictions using the metrics from, or when predicting the end-to-end latency of, a densely co-located physical plans. Although there are process sharing queueing models (Mitrani, 1998; Gross et al., 2008), the complex behaviour of the Java Virtual Machine (JVM) and operating system (OS) thread scheduling will likely require sophisticated, non-linear modelling techniques, in order to provide accurate predictions. This will also require data from many different deployed physical plans to provide sufficiently diverse input to these models.

6.3.5 Serialisation delay

One of the sources of delay in the physical paths of the topology tuple flow plan, which was not covered in the current modelling approach, was the serialisation of tuples for transfers across the network. This aspect of the tuple flow was discussed in section 4.11.1 and it was decided that, for the relatively simple payloads used in the modelling system evaluation (strings and numeric values), the delay due to SerDes would be negligible. However, in production settings the data transferred in the tuples can be large, complex and require complicated unpacking into custom data types. A way to estimate this SerDes delay should be investigated. It could take the form of calibration measurements or alterations to Storm's code to allow metrics for this aspect to be reported.

6.3.6 Network transfer time

Section 4.11.1 detailed the current approach used by the modelling system to predict the expected network transfer latency for remote physical connections in the tuple flow plan. This uses the median latency of the measured round trip time between the worker nodes and is sufficient for Storm clusters that do not change. However, Storm has the ability to add worker nodes dynamically to its cluster. Schedulers could produce physical plan proposals requiring additional nodes to be added. It is also possible that a worker node could fail and a topology rebalance be triggered as part of failure recovery.

Changes in the network topology are not currently covered by the modelling system and so future research could look at incorporating the prediction of the effect of the addition or removal of worker nodes from the Storm cluster. Systems such as Google's Vivaldi (Dabek et al., 2004) framework already allow the prediction of latency to proposed nodes in a network and so could form the basis of this research.

6.3.7 Analytical solution

Calculating the delay due to the complex behaviour of the executor's Disruptor queues and ULT required the use of DES, as there were no appropriate queueing models available in the literature to cover the behaviour of the Disruptor queue (see section 4.2). Whilst the DES approach did yield good results, it also meant that the simulator had to be run over many iterations in order to gain accurate estimations of the executor sojourn time. This meant that the performance modelling process was prolonged, compared to using an analytical model.

Recently, the author collaborated on preliminary work in this area (Cooper et al., 2019). This paper proposes both exact and approximate solutions to modelling the performance of queueing systems like those of Storm's Disruptor queues. The paper also looks at using

the analytical model to optimise the various parameters of the queueing system, such as the flush interval. This highlights another area where the faster results from an analytical solution could allow further optimisation of DSPS topologies.

Future research could look at integrating the proposed analytical model into the modelling processes described in this thesis, as well as looking at creating analytical solutions for other DSPS implementations (see section 6.3.11).

6.3.8 Resource usage

The current focus of the modelling systems is on end-to-end latency performance. However, another key aspect which concerns users of DSPS is the resource requirements of the topologies they design. DSPS clusters are formed of worker nodes with finite resources (memory and processing cycles) and therefore in addition to the expected end-to-end latency of a proposed physical plan, the resources that proposed plan requires are also an aspect of its viability.

Extrapolating how the changes in parallelism will affect the required memory and processing requirements of a topology's executors is not an easy proposition. For example, certain keys in a fields grouped connection could cause more processing than others, and changes in the task assignment in a proposed physical plan could significantly alter the processing load on certain worker nodes. Memory load for some components could show exponential or distinctly non-linear growth, with increases in parallelism. These issues represent an interesting research challenge, the solutions to which would make the performance modelling system more complete.

6.3.9 Hybrid approach

One of the aims of the research described in this thesis was to provide a way to estimate physical plan performance without the need for extensive historical data (see section 1.4.1). Whilst this has been achieved, several of the factors affecting the accuracy of our performance predictions (see section 5.8.5) were linked to the non-linear and complex behaviour of the Storm systems which were not captured by our approach.

A modelling approach using reinforcement-learning or other machine-learning based approaches could yield higher accuracy by allowing the complex interactions of many of the DSPS's systems to be learned over a sufficient spread of historic data. However, another of the stated aims of this research (see section 1.4.1) was to provide performance estimates without the need to first obtain large amounts of training or calibration data.

The solution, which could yield higher accuracy without the need to first perform many scaling operations to gain training data, is to investigate a hybrid approach. Initially, when

a topology is first deployed, a system such as ours could be used to provide performance estimates for proposed physical plans on only a few minutes of metrics data. Concurrently, a reinforcement-learning based system could be running, which uses the topology metrics data over the course of several scaling cycles to create a more complex performance model. After every scaling cycle the performance results of each system could be validated against the deployed plan and once the reinforcement-learning system is delivering consistently better accuracy, the auto-scaling system could switch over to using that approach in future scaling cycles.

6.3.10 Estimation of error

The performance predictions provided by our modelling system are point estimates of the average performance of a proposed physical plan. As described in section 1.2, large, complex topologies of the kind often employed in industry can require significant time to complete even a single scaling operation. Therefore, it would be advantageous in these situations to attach a level of uncertainty to the performance estimations. With such a measure, an auto-scaling system could make an informed decision as to whether a proposed physical plan was worth deploying. If an SLA is particularly strict, the auto-scaling system could aim for a comfortable margin of error before deploying a proposed physical plan.

The current modelling system does not provide a measure of uncertainty for its predictions. However, using distributions instead of point values (such as means, medians, 90th percentiles, etc.) for the parameters (arrival rates, service times, routing probabilities, etc.) could be investigated in order to create a distribution of possible performance values instead of a single average. Other performance modelling methods, such as those used by Jamshidi & Casale (2016) (see section 3.2.2), can provide uncertainty measures by default and so could provide another direction for this future research.

6.3.11 Other DSPSs

Section 2.1 laid out why Apache Storm was chosen as the example DSPS for this research. In the years since this investigation began, several other DSPSs have come to the fore. Apache Samza and Flink have matured into solid and performant systems, with the latter gaining particular traction in the fields of data science and finance. Twitter have replaced their Storm clusters with Heron, a system which is backwards compatible with Storm, which they developed in-house and have since released under an open source licence.

These systems all have their own sets of design decisions, advantages and trade-offs. However, most DSPSs share many common characteristics. They will have ways of representing their streaming queries as query, logical and physical plans. They will have basic processing units like Storm's executors, ways to host and coordinate these processing

units like Storm's worker processes, they will be able to route certain keys to certain processing units and so on. There are many ways that the modelling system detailed in this thesis could be made more generic to allow it to be applied to many different DSPSs. Future research could focus on identifying these areas of commonality and uniqueness with the aim of creating a general framework for DSPS performance modelling.

Whilst undertaking their internship with Twitter, detailed in appendix D, the author began working towards this goal by creating *Caladrius*, a generic software framework for DSPS performance modelling. Their experience with Apache Storm was used to create a performance modelling system for Heron and this framework was used in collaboration with other researchers to investigate further aspects of DSPS performance modelling (Kalim et al., 2019).

6.4 Conclusion

At the start of this thesis, in section 1.4.1, we laid out the key aims for this research:

1) Create a performance modelling system for DSPS topologies.
2) Ensure that the system could handle a wide range of topology component and connection types.
3) Design the systems to be able to perform the modelling whilst only using a small amount of input data.

Given the results shown in chapter 5, the author believes that they have broadly achieved these goals, as well as identifying the main sources of error (section 5.8.5) and advocating ways in which accuracy could be improved.

The performance modelling system detailed in this thesis is by no means complete and, as section 6.3 lays out, there are many areas where the system could be improved and expanded. However, we believe it offers a formidable basis for future research in this area. Furthermore, through the researching of DSPS design and operations, as well as through meeting many of the creators and users of these systems (see appendix D), the author is convinced of the pressing need to add more intelligence into the operations of these business critical systems. With regard to DSPSs, we are not currently making the most of the flexibility and efficiency that cloud computing offers.

Enabling sensible, and above all optimal, elastic scaling decisions is not just about reducing costs. More efficient systems will play a vital role in reducing the projected 3.4 terawatt hours of energy that will be used to power cloud data centres by 2025, a fifth of the world's total energy production (Andrae, 2017, p.15). This is made all the more vital, as currently only 20% of data centre energy needs are supplied by renewable sources (Vidal, 2017). The research and development of more efficient systems is not just about going faster, it is also

about making better use of what we already have, something we could all be better at.

Bibliography

Abadi, D.J., Ahmad, Y., Balazinska, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y. & Zdonik, S. (2005) 'The Design of the Borealis Stream Processing Engine', in *Proceedings of the 2005 CIDR Conference*. 2005. [Online]. 5 January 2005 pp. 277–289.

Ahmad, Y., Tatbul, N., Xing, W., Xing, Y., Zdonik, S., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O. & Rasin, A. (2005) 'Distributed operation in the Borealis stream processing engine', in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data - SIGMOD '05*. [Online]. 14 June 2005 Baltimore, Maryland: ACM Press. pp. 882–884.

Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K. & Warneke, D. (2014) The Stratosphere platform for big data analytics. *The VLDB Journal*. 23 (6), 939–964.

Allen, S.T., Pathirana, P. & Jankowski, M. (2015) *Storm Applied: Strategies for Real-time Event Processing*. 1st edition. Dan Maharry, Aaron Colcord, & Elizabeth Welch (eds.). New York: Manning Publications.

Andrae, A.S.G. (2017) *Total Consumer Power Consumption Forecast*. [Online] [online]. Available from: https://www.researchgate.net/publication/320225452_Total_Consumer_Power_Consumption_Forecast (Accessed 22 May 2019).

Aniello, L., Baldoni, R. & Querzoni, L. (2013) 'Adaptive Online Scheduling in Storm', in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems - DEBS '13*. [Online]. 29 June 2013 pp. 207–218.

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U. & Widom, J. (2016) 'STREAM: The Stanford Data Stream Management System', in Minos Garofalakis, Johannes Gehrke, & Rajeev Rastogi (eds.) *Data Stream Management*. [Online]. Berlin, Heidelberg: Springer Berlin Heidelberg. pp. 317–336.

Balazinska, M., Balakrishnan, H., Madden, S. & Stonebraker, M. (2005) 'Fault-Tolerance

in the Borealis Distributed Stream Processing System', in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data - SIGMOD '05*. [Online]. 14 June 2005 pp. 13–24.

Barrett, E., Howley, E. & Duggan, J. (2013) Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency Computation Practice and Experience*. 25 (12), 1656–1674.

Bu, X., Rao, J. & Xu, C.Z. (2013) Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Transactions on Parallel and Distributed Systems*. 24 (4), 681–690.

Calzarossa, M.C., Massari, L. & Tessera, D. (2016) Workload Characterization: A Survey Revisited. *ACM Computing Surveys*. 48 (3), 1–43.

Carbone, P., Haridi, S., Pietzuch, P., KTH & Skolan för elektroteknik och datavetenskap (EECS) (2015) Scalable and Reliable Data Stream Processing. [Online].

Chen, L. & Avizienis, A. (1995) 'N-Version Programming: A fault tolerance approach to reliability of software operation', in *Proceedings of the International Symposium on Fault-Tolerant Computing*. [Online]. June 1995 Pasadena, CA, USA:. pp. 113–119.

Ching, A., Edunov, S., Kabiljo, M., Logothetis, D. & Muthukrishnan, S. (2015) One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*. 8 (12), 1804–1815.

Cooper, T. (2016) 'Proactive scaling of distributed stream processing work flows using workload modelling: Doctoral symposium', in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems - DEBS '16*. [Online]. 13 June 2016 Irvine, California: ACM Press. pp. 410–413.

Cooper, T., Ezhilchelvan, P. & Mitrani, I. (2019) 'A queuing model of a stream-processing server', in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. [Online]. October 2019 pp. 27–35.

Dabek, F., Cox, R., Kaashoek, F. & Morris, R. (2004) 'Vivaldi: A decentralized network coordinate system', in *ACM SIGCOMM Computer Communication Review*. [Online]. 30 August 2004 pp. 15–26.

Dean, J. & Ghemawat, S. (2008) MapReduce: Simplified data processing on large clusters. *Communications of the ACM*. 51 (1), 107.

De Matteis, T. & Mencagli, G. (2016) Keep calm and react with foresight: Strategies for low- latency and energy-efficient elastic data stream processing. *ACM SIGPLAN*

Notices. 51 (8), 1–12.

Eskandari, L., Huang, Z. & Eyers, D. (2016) 'P-Scheduler: Adaptive hierarchical scheduling in Apache Storm', in *Proceedings of the Australasian Computer Science Week Multiconference on - ACSW '16*. [Online]. 1 February 2016 pp. 1–10.

Farahabady, M.R.H., Samani, H.R.D., Wang, Y., Zomaya, A.Y. & Tari, Z. (2016) 'A QoS-aware controller for Apache Storm', in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. [Online]. 1 October 2016 pp. 334–342.

Fernandez-Baca, D. (1989) Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*. 15 (11), 1427–1436.

Floratou, A., Agrawal, A., Graham, B., Rao, S. & Ramasamy, K. (2017) 'Dhalion : Self-regulating stream processing in Heron', in *Proceedings of the VLDB Endowment*. [Online]. 1 August 2017 pp. 1825–1836.

Foroni, D., Axenie, C., Bortoli, S., Hassan, M.A.H., Acker, R., Tudoran, R., Brasche, G. & Velegrakis, Y. (2018) 'Moira: A goal-oriented incremental machine learning approach to dynamic resource cost estimation in distributed stream processing systems', in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. [Online]. 27 August 2018 pp. 1–10.

Fu, T.Z.J., Ding, J., Ma, R.T.B., Winslett, M., Yang, Y. & Zhang, Z. (2015) 'DRS: Dynamic resource scheduling for real-time analytics over fast streams', in *2015 IEEE 35th International Conference on Distributed Computing System*. [Online]. 29 June 2015 pp. 411–420.

Gautam, B. & Basava, A. (2019) Performance prediction of data streams on high-performance architecture. *Human-centric Computing and Information Sciences*. 9 (1), 2.

Gordon, W.J. & Newell, G.F. (1967) Closed queuing systems with exponential servers. *Operations Research*. 15 (2), 254–265.

Graham, B., Floratau, A. & Agrawal, A. (2017) *From rivulets to rivers: Elastic stream processing in Heron*. [Online] [online]. Available from: https://conferences.oreilly.com/strata/strata-ca-2017/public/schedule/detail/55639.

Gross, D., Shortle, J.F., Thompson, J.M. & Harris, C.M. (2008) *Fundamentals of Queueing Theory*. 4th edition. Vol. 627. John Wiley & Sons.

Gupta, P., Goel, A., Lin, J., Sharma, A., Wang, D. & Zadeh, R. (2013) 'WTF: The who to follow service at twitter', in *Proceedings of the 22nd International Conference on World Wide Web*. [Online]. 13 May 2013 Rio de Janeiro, Brazil:. pp. 505–514.

Hadoop, A. (2019) *Apache Hadoop*. [Online] [online]. Available from: http://hadoop.apache.org/.

Heinze, T., Jerzak, Z., Hackenbroich, G. & Fetzer, C. (2014) 'Latency-aware elastic scaling for distributed data stream processing systems', in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. [Online]. 26 May 2014 pp. 13–22.

Heinze, T., Ji, Y., Jerzak, Z., Pan, Y., Grueneberger, F.J. & Fetzer, C. (2013) Elastic complex event processing under varying query load. *BD3@ VLDB*. 101825–30.

Heinze, T., Pappalardo, V., Jerzak, Z. & Fetzer, C. (2014) 'Auto-Scaling Techniques for elastic data stream processing', in *2014 IEEE 30th International Conference on Data Engineering Workshops*. [Online]. 31 March 2014 pp. 296–302.

Heinze, T., Zia, M., Krahn, R., Jerzak, Z. & Fetzer, C. (2015) 'An adaptive replication scheme for elastic data stream processing systems', in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS '15*. [Online]. 24 June 2015 pp. 150–161.

Herbst, N., Amin, A., Andrzejak, A., Grunske, L., Kounev, S., Mengshoel, O.J. & Sundararajan, P. (2017) 'Online Workload Forecasting', in Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, & Xiaoyun Zhu (eds.) *Self-Aware Computing Systems*. [Online]. Cham: Springer International Publishing. pp. 529–553.

Hesse, G. & Lorenz, M. (2015) 'Conceptual survey on data stream processing systems', in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. [Online]. 14 December 2015 pp. 797–802.

Huifang Feng & Yantai Shu (2005) 'Study on network traffic prediction techniques', in *Proceedings. 2005 International Conference on Wireless Communications, Networking and Mobile Computing, 2005.* [Online]. 26 September 2005 pp. 1041–1044.

Jackson, J.R. (1957) Networks of waiting lines. *Operations Research*. 5 (4), 518–521.

Jamshidi, P. & Casale, G. (2016) 'An uncertainty-aware approach to optimal configuration of stream processing systems', in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. [Online]. 19 September 2016 pp. 39–48.

Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M. & Roscoe, T. (2018) 'Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows', in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. [Online]. 2018 Carlsbad, CA, USA:. pp. 783–798.

Kalim, F., Cooper, T., Wu, H., Li, Y., Wang, N., Lu, N., Fu, M., Qian, X., Luo, H., Cheng, D., Wang, Y., Dai, F., Ghosh, M. & Wang, B. (2019) 'Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems', in *Proceedings of the 35th IEEE International Conference on Data Engineering*. [Online]. 8 April 2019 Macau SAR, China:. pp. 1886–1897.

Kendall, D.G. (1953) Stochastic processes occuring in the theory of queues and their analysis by the method of the imbedded Markov chain. *The Annals of Mathematical Statistics*. 24 (3), 338–354.

Kingman, J.F.C. (1961) The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society*. 57 (October), 902.

Kleppmann, M. (2017) *Designing Data Intensive Applications*. O'Reilly Media Inc.

Kreps, J. (2014) *Questioning the Lambda Architecture*. [Online] [online]. Available from: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (Accessed 19 January 2019).

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K. & Taneja, S. (2015) 'Twitter Heron', in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. [Online]. 27 May 2015 pp. 239–250.

Lax, R., Chernyak, S. & Akidau, T. (2018) *Streaming Systems*. O'Reilly Media, Inc.

Li, J., Pu, C., Chen, Y., Gmach, D. & Milojicic, D. (2016) 'Enabling elastic stream processing in shared clusters', in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. [Online]. 27 June 2016 pp. 108–115.

Li, T., Tang, J. & Xu, J. (2016) Performance modelling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*. 2 (4), 353–364.

Little, J.D.C. (1961) A proof for the queuing formula: L = lambda W. *Operations Research*. 9 (3), 383–387.

Liu, X. & Buyya, R. (2017) 'D-Storm: Dynamic resource-efficient scheduling of stream processing applications', in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. [Online]. 15 December 2017 pp. 485–492.

Lohrmann, B., Janacik, P. & Kao, O. (2015) 'Elastic stream processing with latency guarantees', in *2015 IEEE 35th International Conference on Distributed Computing Systems*. [Online]. 29 June 2015 pp. 399–410.

Lombardi, F., Aniello, L., Bonomi, S. & Querzoni, L. (2018) Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel*

and Distributed Systems. 29 (3), 572–585.

Lombardi, F., Muti, A., Aniello, L., Baldoni, R., Bonomi, S. & Querzoni, L. (2019) PASCAL: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems*. 98342–361.

Lorido-Botran, T., Miguel-Alonso, J. & Lozano, J.A. (2014) A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*. 12 (4), 559–592.

Marz, N. & Warren, J. (2015) *Big Data - Principles and Best Practices of Scalable Realtime Data Systems*. New York: Manning Publications.

Masotto, X., Gupta, S. & Sanbhadti, R. (2015) 'Dynamic Topology Scaling in Apache Storm', in *Technical Report*. [Online]. 2015 University of Illinois at Urbana-Champaign.

Meyn, S.P. & Down, D. (1994) Stability of generalized Jackson networks. *The Annals of Applied Probability*. 4 (1), 124–148.

Mitrani, I. (1998) *Probabilistic Modelling*. Cambridge University Press.

Neumeyer, L., Robbins, B., Nair, A. & Kesari, A. (2010) 'S4: Distributed stream computing platform', in *2010 IEEE International Conference on Data Mining Workshops*. [Online]. 13 December 2010 pp. 170–177.

Peng, B. (2015) Elasticity and resource aware scheduling in distributed data stream processing systems. [Online]. University of Illinois.

Peng, B., Campbell, R., Hosseini, M., Hong, Z. & Farivar, R. (2015) 'R-Storm: Resource-aware scheduling in Storm', in *Proceedings of the 16th Annual Middleware Conference*. [Online]. 24 November 2015 pp. 149–161.

Reiser, M. & Lavenberg, S.S. (1980) Mean-value analysis of closed multichain queuing networks. *Journal of the ACM*. 27 (2), 313–322.

Sasikala, S. & Indhira, K. (2016) Bulk service queueing models - A survey. *International Journal of Pure and Applied Mathematics*. 106 (6), 43–56.

Singhal, A. (2012) *Introducing the Knowledge Graph: things, not strings*. [Online] [online]. Available from: https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html (Accessed 29 May 2019).

Sun, D. & Huang, R. (2016) A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access*. 48593–8607.

Sun, D., Yan, H., Gao, S., Liu, X. & Buyya, R. (2018) Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. *The*

Journal of Supercomputing. 74 (2), 615–636.

Taylor, S.J. & Letham, B. (2018) Forecasting at scale. *The American Statistician*. 72 (1), 37–45.

Tesauro, G., Jong, N.K., Das, R. & Bennani, M.N. (2006) 'A hybrid reinforcement learning approach to autonomic resource allocation', in *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*. [Online]. 12 June 2006 pp. 65–73.

Thompson, M., Farley, D., Barker, M., Gee, P. & Stewart, A. (2011) 'Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads', in *Technical Report*. [Online]. May 2011 LMAX.

Toshniwal, A., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K. & Fu, M. (2014) 'Storm@twitter', in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. [Online]. 18 June 2014 pp. 147–156.

Vakilinia, S., Zhang, X. & Qiu, D. (2016) 'Analysis and optimization of big-data stream processing', in *2016 IEEE Global Communications Conference (GLOBECOM)*. [Online]. 4 December 2016 Washington, DC, USA: IEEE. pp. 1–6.

Van Der Veen, J.S., Van Der Waaij, B., Lazovik, E., Wijbrandi, W. & Meijer, R.J. (2015) 'Dynamically scaling Apache Storm for the analysis of streaming data', in *Proceedings - 2015 IEEE 1st International Conference on Big Data Computing Service and Applications, BigDataService 2015*. [Online]. 30 March 2015 pp. 154–161.

Vidal, J. (2017) *'Tsunami of data' could consume one fifth of global electricity by 2025*. [Online] [online]. Available from: https://www.climatechangenews.com/2017/12/11/tsunami-data-consume-one-fifth-global-electricity-2025/ (Accessed 22 May 2019).

Wang, Y., Wang, H., Jia, Y. & Liu, B. (2006) 'Closed Queueing Network Model for Multi-tier Data Stream Processing Center', in Xiaofang Zhou, Jianzhong Li, Heng Tao Shen, Masaru Kitsuregawa, & Yanchun Zhang (eds.) *Frontiers of WWW Research and Development - APWeb 2006*. Lecture Notes in Computer Science. [Online]. 2006 Berlin, Heidelberg: Springer. pp. 899–904.

Xu, J., Chen, Z., Tang, J. & Su, S. (2014) 'T-storm: Traffic-aware online scheduling in Storm', in *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*. [Online]. 30 June 2014 pp. 535–544.

Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. & Stoica, I. (2010) 'Spark : Cluster computing with working sets', in *HotCloud '10*. [Online]. 22 June 2010 p. 95.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. & Stoica, I. (2013) 'Discretized

Streams: Fault-tolerant streaming computation at scale', in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. [Online]. 3 November 2013 pp. 423–428.

Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R. & Stoica, I. (2008) Improving MapReduce performance in heterogeneous environments. *OSDI*. 8 (4), 7.

Appendix A

Queuing Theory Primer

A.1 Queueing Theory Notation

Queueing systems are made up of one or more queues attached to one or more servers. Items, usually referred to as *jobs*, arrive into the queue at an average rate λ (arrival rate) and are processed by the server at rate μ (service rate). Sometimes the service time (b) of the server is used to denote its performance, which is equivalent to $\frac{1}{\mu}$. A queueing system experiences an *offered load* (ρ) which describes the probability that a server is busy serving a job:

$$\rho = \frac{\lambda}{\mu} = \lambda b$$

A single server queueing system is said to be *stable* if $\rho < 1$. If this is not true then the system is said to be *unstable* and the queue will grow indefinitely. The performance measures for a queueing system include: the average number of jobs waiting for service (l); the total number of jobs in the system (L); the average waiting time in the queue (w); and the average sojourn time (W), which is the time from a job arriving at the queue to it completing service. The performance measures are illustrated in figure A.1 and the values are related by the following formulas:

$$W = w + b$$

$$L = l + 1$$

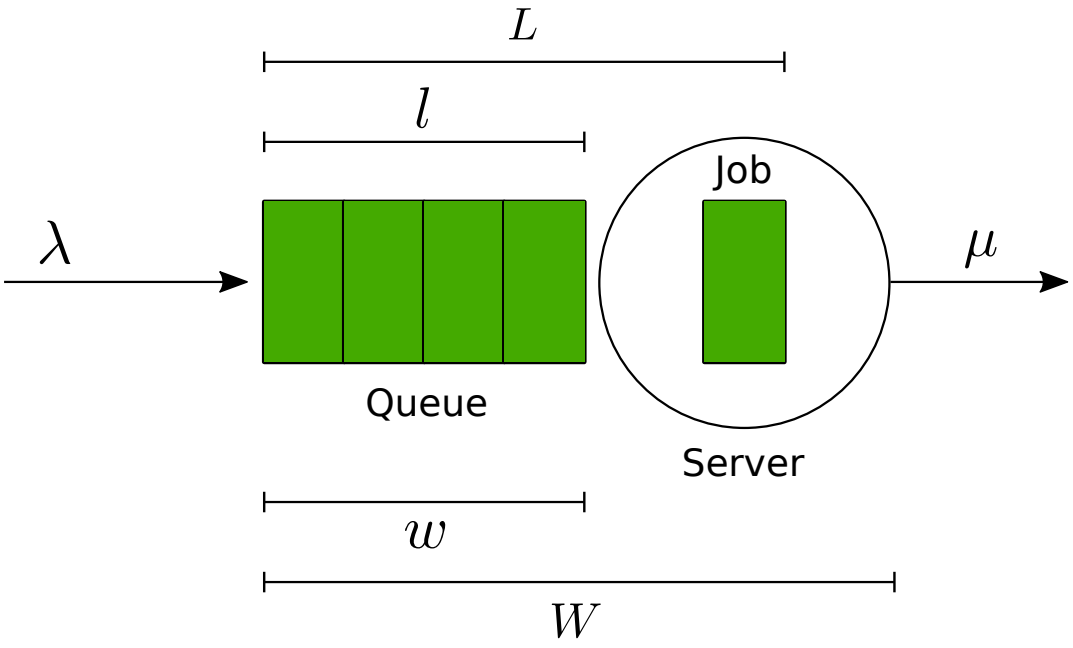

Figure A.1: The standard performance measures for a queueing system.

A.2 Queue Categorisation

Queueing systems are categorised according to a system first formalised by Kendall (1953). Kendall's original notation denotes the type of queueing node using three terms: $A/S/c$ where A denotes the distribution of the time between arrivals into the queue, S the service time distribution and c the number of servers at the node. This has been extended in recent years into a more detailed form:

$$A^X/S^Y/c/K/N/D$$

A^X The arrival process describes the type of distribution the arrival rate of jobs follows. Typical values are M for memoryless (Poisson), D for deterministic (fixed), G for general and E_k for Erlang. The optional superscript X usually refers to batch arrivals and the value indicates how many jobs arrive in each batch.

S^Y Similar to the arrival process, the service time distribution has typical values such as M, D, G and E_k. The optional superscript Y indicates bulk (batch) service with the value representing the number of jobs served at one time.

c The number of servers at this queueing node. This number represents how many jobs can be processed in parallel at this node.

K The capacity of the system. This number, if present, indicates the maximum number of jobs in the queueing node. This usually includes those in service but may sometimes simply be the queue limit. Once the queue node population reaches this value new arrivals will be turned away. If this number is not present it is assumed that $K = \infty$.

N The size of the job source population. A small population will significantly affect the effective arrival rate, because as more jobs queue up there are fewer left available to arrive into the system. If this number is not present then it is assumed that $N = \infty$.

D The queue discipline describes in what order jobs in the queue are served. *FIFO* for first in first out (first come first served), *LIFO* for last in first out (last come first served), *SIRO* for service in random order, *PN* for priority service and *PS* for processor sharing where all jobs in the queue are served at the same time (time to process all jobs in the queue is the same as other disciplines but all jobs will finish at the same time). If this value is not present then the queue is assumed to be *FIFO*.

Appendix B

Executor Simulator Implementation

In order to predict the latency a tuple will experience whilst passing through the executor's user logic thread (ULT), a discrete-event simulation (DES) for this element was created. The sections below describe the process and algorithms employed in the simulator.

B.1 Simulation Process

In order to simplify the simulator we have assumed that all the possible events that may occur for the ULT (input tuple list arrival, timer flush signal or tuple service completing) happen at rates where the inter-arrival time between events is exponentially distributed. This allows us to combine the probabilities of all three events into a single spectrum:

$$P(\text{arrival}) = \frac{\lambda}{\lambda + \tau + \mu}$$

$$P(\text{flush}) = \frac{\tau}{\lambda + \tau + \mu}$$

$$P(\text{service}) = \frac{\mu}{\lambda + \tau + \mu}$$

These events can be simulated by sampling from a uniform distribution between 0 and 1 and comparing that sample value to the relative probabilities of each event. Figure B.1 illustrates the probability spectrum for the ULT. A randomly chosen value (x) will indicate an arrival if $0 \leq x < A$, a flush signal if $A \leq x < B$ and tuple service completing if $B \leq x < 1$.

The simulator is run until a predetermined number of arrivals have occurred. For each iteration, once the type of event has been chosen, the effect of that event on the state

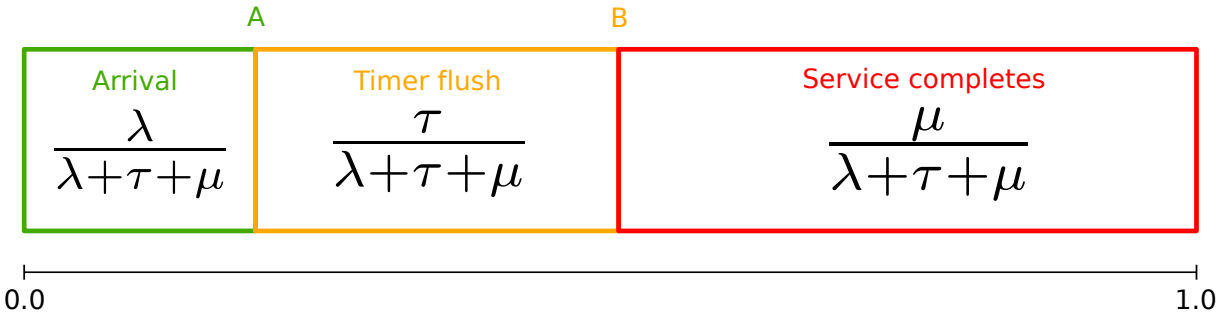

Figure B.1: The event probabilities used in the ULT DES.

of the ULT can be calculated. This is detailed in section B.2 below. For each arrival event the number of arriving tuples (in the input tuple list — see section 4.12) is added to the total arrival count and the total number of tuple currently within the ULT system is added to an aggregate total system population count. At the end of the simulation run the total system population count is divided by the total arrival count to give the average number of tuples (L) in the ULT system when a new tuple arrives. This is then used in conjunction with Little's Law (Little, 1961), see equation B.1 below, to calculate the average sojourn time (W) for tuples through the ULT by dividing the average tuples in system (L) by the tuple arrival rate (γ).

$$L = \gamma W \tag{B.1}$$

B.2 Full System Simulator

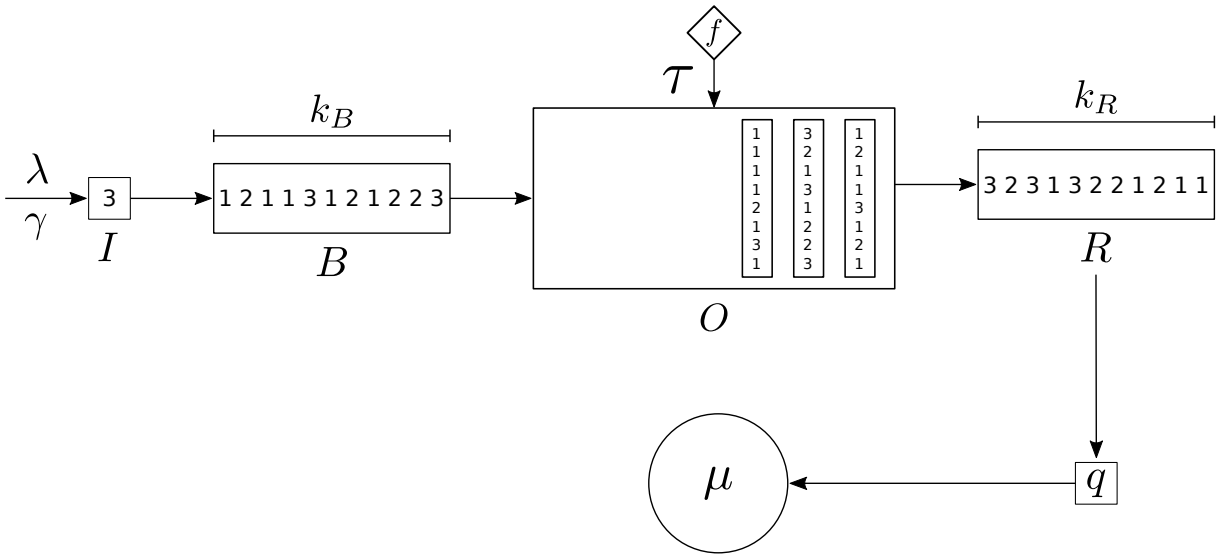

Figure B.2: Elements of the executor ULT showing the state variables used in the DES.

To simulate the ULT's operation, the effect of each of the possible events (tuple list arrival,

flush signal or tuple service completion) on the state of the ULT system needs to be defined. To do this we first need to define the parameters of the ULT state:

$$(\mathbf{B}, \mathbf{O}, \mathbf{R}, q, f)$$

Where:

B Is a vector of numbers indicating the number of tuples in each input tuple list in the Disruptor queue's input batch. The length of this vector must be less than or equal to the input batch limit k_B ($|\mathbf{B}| \leq k_B$).

O Is a vector of B vectors, which represents the size of batches currently in the Disruptor queue's overflow queue.

R Is a vector of numbers, where each value represents a slot in the ring buffer and indicates the number of tuples in the list in that slot. The length of this vector must be less than or equal to the Ring Buffer size limit k_R ($|\mathbf{R}| \leq k_R$).

q Is a value indicating the number of tuples in the internal buffer.

f Is a boolean value indicating if the Disruptor queue is currently in a flushing state.

Figure B.2 illustrates how these parameters are related to the elements of the ULT. In addition to the state variables above, the following variables are also part of the ULT simulator:

k_B Is the size limit of the Disruptor queue input batch and is the value at which input batches will either be moved to the Ring Buffer or the overflow queue if there is insufficient space. This is the limit for the length of the simulator's **B** vector.

k_R Is the size limit for the Disruptor queue Ring Buffer. This is the limit for the length of the simulator's **R** vector.

I Is the number of tuples in each arriving tuple list.

λ Is the arrival rate of tuple lists into the executor receive queue (ERQ).

γ Is the equivalent tuple arrival rate into the ERQ.

τ Is the rate at which a flush signal is sent to the overflow queue.

μ Is the rate at which individual tuples are served from the internal buffer by the task instance.

There are three distinct events that can occur in the context of the executor's ULT: the arrival of an input tuple list; the overflow queue periodic flush signal is issued; and a tuple completes service within the task. The effects on the Disruptor queue's state, due to each

of these events, are discussed in sections 2.7.3, 2.7.4, 2.7.5 respectively. The sections below detail how each of these events affect the simulator's state parameters listed above.

B.2.1 Tuple list arrival

Figure 2.5 shows a flow diagram of the possible state changes as a result of a tuple list arrival. These stages of the flow diagram have been converted into the procedure, shown in Algorithm 1, which relates these steps to the ULT system state shown above.

Input: Incoming tuple list size (I)
Add I to **B**;
if $|\mathbf{B}| \geq k_B$ **then**
  if $|\mathbf{O}| > 0$ **then**
    Move **B** to **O**;
  else
    if $k_r - |\mathbf{R}| \geq |\mathbf{B}|$ **then**
      Move values from **B** to **R**;
    else
      Move **B** to **O**;
    end
  end
end
while $|\mathbf{O}| > 0$ **do**
  Get the first (oldest) batch vector (**b**) in **O**;
  if $k_r - |\mathbf{R}| \geq |\mathbf{b}|$ **then**
    Add values from **b** to **R**;
  else
    Exit the while loop;
  end
end
if $|\mathbf{O}| \leq 0$ **then**
  Set f to False;
end

Algorithm 1: Tuple list arrival operation.

B.2.2 Flush interval completes

Figure 2.6 shows a flow diagram of the possible state changes as a result of a timed flush interval completing. These stages of the flow diagram have been converted into the procedure, shown in Algorithm 2, which relates these steps to the ULT system state shown

above.

Add current **B** to **O**;
while $|\mathbf{O}| > 0$ **do**
    Get the first (oldest) batch vector (**b**) in **O**;
    if $k_r - |\mathbf{R}| \geq |\mathbf{b}|$ **then**
        Move values from **b** to **R**;
    else
        Set f to True;
        Exit the while loop;
    end
end
if $|\mathbf{O}| \leq 0$ **then**
    Set f to False;
end

Algorithm 2: State change operations when timed flush interval completes.

B.2.3 Tuple completes service

Figure 2.7 shows a flow diagram of the possible state changes as a result of a tuple completing service with the ULT task. These stages of the flow diagram have been converted into the procedure, shown in Algorithm 3, which relates these steps to the ULT system state shown above.

if $q > 0$ **then**
    Move 1 tuple from q to the server;
else
    if $|\mathbf{R}| > 0$ **then**
        Move the sum of all values in **R** and add that to q;
        Move 1 tuple from q to the server;
    end
end
if f *is True* **then**
    Perform timed flush operations;
end

Algorithm 3: State change operations when a tuple completes service.

B.2.4 Continuous flush operation

During the Disruptor queue's operation there is the possibility of a flush of the overflow queue to continue (linger) for some time. This can occur if the arrival and service rates are such that when a flush operation is triggered, new batches are added to the overflow

queue faster than they can be cleared into the ring buffer. This situation means that the flushing operation never stops and batches will be placed onto the ring buffer as soon as space becomes available.

In the algorithms above we simulate the possibility of a lingering flush operation by setting a boolean flag (f) to true during the timer flush completion algorithm, at the point where the ring buffer is full but there are still batches in the overflow queue.

During the service completion algorithm the boolean flag is checked and the flush interval completion procedure is run again. During an arrival the flush interval code is run as part of the arrival process, however the value of f is set to false if the overflow queue is cleared. If the overflow queue is not cleared then the value of f stays as it was. In this way, if the system is in a *flushing* state (f is True), then the flush procedure will always be run during each simulated event. As the overflow queue can only be cleared when space is available on the ring buffer and this can only happen after service completes to create space in the ULT internal buffer, there is no way that the overflow queue could become empty between the events defined by the simulator (arrival, flush or service) and so the DES approach remains valid in the presence of the Disruptor queue flushing behaviour.

B.3 Simplified System Simulator

Whilst the full system simulator, described in section B.2, simulates all elements of the Disruptor queue and task instances within the executor ULT, the state representation is complex and involves retaining many values (vectors of vectors) in memory. Whilst the full system simulation allows the investigation of the inter-play between the various elements of the ULT, it is slow to operate and memory intensive.

As discussed in section B.1, in order to model the sojourn time (W) across the ULT all we need to know is the average tuple arrival rate (γ) and the average number of tuples in the system at the time of arrival (L), see equation B.1. If we focus solely on the number of tuples in the system, then the simulator state can be simplified.

Once a tuple has been added to either the overflow queue, or the ring buffer if the overflow queue was empty and there was space, then that tuple can only leave via service in the task instance. How and where that tuple is stored, or whether the system is flushing or not, is immaterial to the calculation of L. The input batch, however, can prevent tuples from entering into the overflow queue and therefore ultimately being processed. If the arrival rate is low and/or the service rate is high then tuples added to the overflow queue/ring buffer could be cleared very quickly, but tuples must wait in the input batch for either a flush interval to complete or for the batch to fill up. Therefore, thanks to the input batch, arriving tuples could see a higher population (with tuples in the input batch but all other

elements empty) than if the input batch was not present. To capture this behaviour we need to simulate the input batch, however all other elements can be reduced to a single value.

The above observation reduces the state of this simplified simulator to two parameters:

$$(\mathbf{B}, u)$$

Where $\mathbf{B}$ is the input batch vector holding the number of tuples in each input batch, as in section B.2, and u is the combined number of tuples in all the other elements of the ULT (namely the overflow queue, ring buffer and the internal buffer), this simplified state significantly reduces the complexity of the state transitions that result from each event.

B.3.1 Tuple list arrival

Input: Incoming tuple list size (I)
Add I to $\mathbf{B}$;
if $|\mathbf{B}| \geq k_B$ **then**
    Add the sum of $\mathbf{B}$ to u;
    Clear $\mathbf{B}$;
end

Algorithm 4: Simplified tuple list arrival operation.

B.3.2 Flush interval completes

Add the sum of $\mathbf{B}$ to u;
Clear $\mathbf{B}$;

Algorithm 5: Simplified state change operations when timed flush interval completes.

B.3.3 Tuple completes service

if $u \geq 1$ **then**
    Reduce the value of u by 1;
end

Algorithm 6: Simplified state change operations when a tuple completes service.

B.4 Comparison of Simulators

The simulator described in section B.3 is significantly more straightforward to implement and understand than that shown in section B.2. However, the latter was used during the investigation of the Disruptor queue's behaviour in order to aid in confirming assumptions about its performance. Section C.5.5 discusses the implementation details of both these

simulation approaches and compares the results and performance of both under various conditions.

Appendix C

Modelling System Implementation

This appendix details the implementation of the modelling approaches laid out in chapter 4. It also gives details of the various systems and programs used to gather metrics from the Apache Storm distributed stream processing system (DSPS).

Because naming things is one of the two hardest problems in computer science[1], significant thought went into the naming of the umbrella project under which all the code for this doctoral research would sit. As the test DSPS is called Storm and in an early version the metrics gathering approach involved issuing *tracer* packets into the topologies, we settled on the name *Storm-Tracer*. This has the additional benefit of rhyming with *storm chaser*, a term for people who follow massive storms (at significant risk to their personal safety) in order to study them. This seemed apt as this was essentially what the author had been doing, metaphorically speaking, for the better part of four years. Although with significantly less risk[2] to their physical safety.

An overview of the Storm-Tracer system is given in section C.1, with details of the major components and processes detailed in sections C.2, C.4, C.5.

All the code referenced in this chapter, as well as the validation code discussed in section 5.2, is available for review in the digital copy attached to this document.

C.1 Storm-Tracer system overview

Storm-Tracer consists of three main components: metrics gathering; topology structure storage and analysis; and performance modelling. Figure C.1 shows how these major components interact with an Apache Storm cluster.

When the Storm scheduler creates a new physical plan, this is issued to the Storm-Tracer

[1] The others being cache invalidation and off by one errors.
[2] Mental well-being not withstanding.

system which will store this in the topology structure store. The performance modelling system will then enact the procedures outlined in chapter 4 by querying the metrics system to obtain the various summary statistics needed to performed the modelling. Once an estimate of the end-to-end latency for the proposed physical plan has been calculated, this can be returned to the Storm scheduler which can then decide if it wishes to deploy the plan or create a new one.

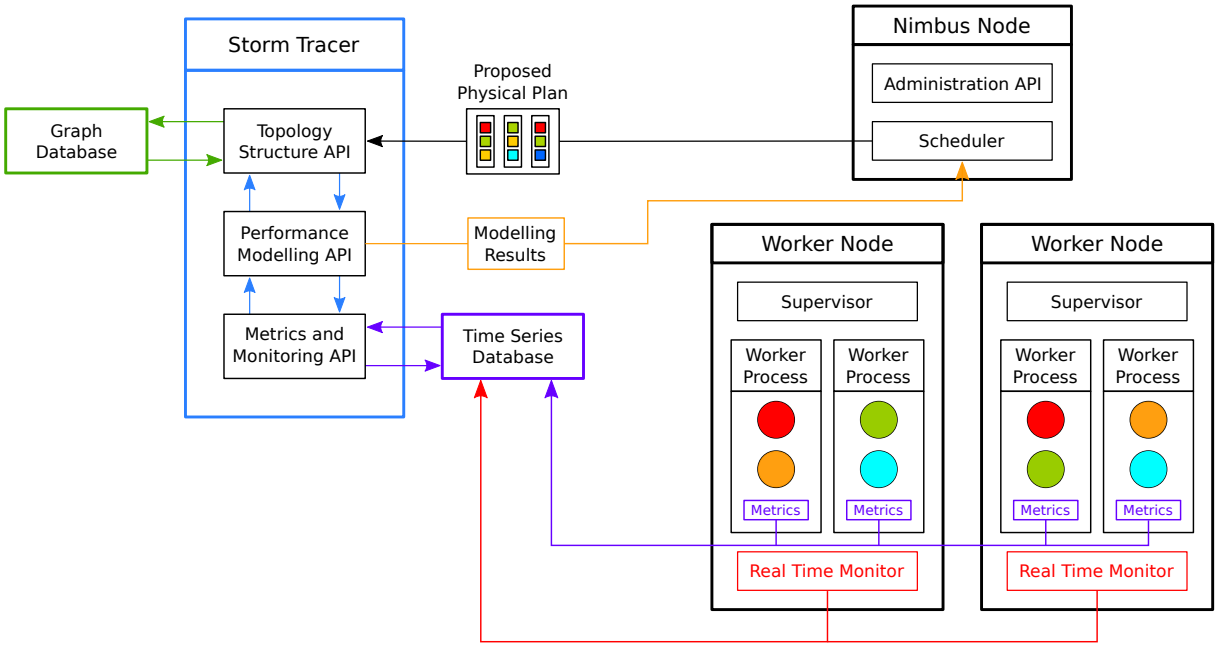

Figure C.1: The various components of the Storm-Tracer system.

C.2 Metrics Gathering and Storage

Metrics are the key to accurate modelling of Apache Storm topologies. As discussed in section 2.13, Storm provides a powerful and extensible metrics framework[3].

C.2.1 Time series database

In order to properly investigate the performance of Storm topologies, the metrics from the various different test cases have to be retained in an easily accessible form. Metrics from Storm are reported at a constant rate and much of the analysis detailed in chapter 4 requires summary statistics to be calculated over a defined time period when a topology was configured according to a particular physical plan. There are many options for storing metrics data, from continuously appending to a series of text based comma separated values (CSV) files, to constructing a whole separate stream processing pipeline to receive,

[3]Is should be noted that, as of Storm version 1.2, a second metrics API was introduced with a more advanced feature set. Storm-Tracer was designed for versions of Storm that used the old API, and as there is no difference in the measurements reported between the two, Storm-Tracer continues to use the older (now depreciated) metrics API.

process and save the metrics values. However, the simplest approach, that provided good flexibility, was to use a time series database (TSDB).

TSDBs are a special class of databases that focus on indexing data by time and providing optimised functionality to query that data. It is possible to configure traditional relational database systems to store time indexed data, however this often comes with trade-offs in the form of query performance. TSDBs allow data from defined time periods to be quickly retrieved and summarised, which is a key requirement for the modelling process.

There are several dedicated TSDBs available. As with the DSPS selection, only open-source options were considered. In 2015, when the metrics gathering system was being designed, the most popular open source option was InfluxDB[4]. This system provided a HTTP REST application programming interface (API) to quickly add data to a database, allowed separating metrics into different measurements and allowed additional indexed tag fields to be added to the metrics. This allowed each of the metrics that Storm provided (execute latency, emit count, etc.) to be placed in their own measurement in the database and tagged with the topology, task and worker process identifier information. This meant that obtaining summaries of particular metrics was a simple SQL-like query away. For example, the average execute latency for each task of each component, in a given topology, in a given time period, could be found using the following command:

```
SELECT MEAN(latency) FROM "execute-latency"
WHERE topology = 'Test1'
AND time >= '2020-03-17T14:24:09'
AND time <= '2020-03-17T14:34:11'
GROUP BY component, taskID
```

Using a TSDB has the added advantage of unifying the storage and querying aspect of metrics gathering into one operation. Using CSV files would have been a simpler approach from an implementation standpoint, but would have required a further transform and load step when the data was to be queried. The use of a centralised database for metrics gathering does, however, introduce a bottleneck. InfluxDB can deal with a large volume of input requests, but for large topologies and/or topologies with a low metric bucket period the metrics reporting rate could be very high. This is unlikely to be an issue for the test cases used in this research, however the system could be made more robust by placing a message broker (such as Apache Kafka) in front of the database and using a secondary processing step to extract messages off the broker and load these into InfluxDB. This would smooth out load spikes and allow the system to be resilient to slow writes causing incoming requests to be dropped.

[4] https://www.influxdata.com/time-series-platform/

C.2.2 Custom metrics

The metrics gathering code described in the following sections can be seen at the link below, and a copy is available in the digital media attached to this document:

https://github.com/tomncooper/stormtracer

InfluxDB metrics consumer

In order for Storm metrics to be sent to an InfluxDB instance, a custom metrics consumer implementation is required so that Storm knows how to format the metrics and where to send them. Storm's metrics API[5] provides several pre-built consumers, including ones which log metrics to files (this was the CSV option described earlier) and a consumer which will forward metrics to a HTTP end point which would allow a separate server (an example implementation of which is also provided by Storm) to receive and process the metrics. It also provides interfaces to allow the creation of your own custom consumer.

InfluxDB provides several different language clients, including Java. This provides access to a batch writing InfluxDB client that can minimise the network communication each Storm metric consumer has to make. The behaviour of the metrics consumer implementation is important to the overall performance of the Storm topology. As the Storm documentation states:

> "Please keep in mind that `MetricsConsumerBolt` is just a kind of bolt, so [the] whole throughput of the topology will go down when registered metrics consumers cannot keep up [with] handling [the] incoming metrics... One idea to avoid this is [by] making your implementation of Metrics Consumer as non-blocking [as possible.]"

All the executors within the topology will be connected to the metrics consumer bolts via a shuffle grouped connection. The consumer executors will be scheduled in the topology physical plan in the same way as the executors of the user defined bolts. Large volumes of metrics — either from a large number of executors, low metric bucket period and/or a consumer that is slow to process metrics — could lead to the metrics consumer bolt being overloaded. It is therefore important to design a consumer implementation that processes metrics as efficiently as possible and also ensure that the parallelism of the consumer is set to an appropriate level.

Setting the parallelism of the consumers is a modelling task in itself. The overall arrival rate into the metric consumers will be approximately equal to the number of executors and Ackers in the topology multiplied by one over the configured metric bucket period. Unfortunately, despite the metrics consumers being based on the standard bolt implemen-

[5] https://storm.apache.org/releases/1.2.2/Metrics.html

tation, Storm does not provide metrics for the processing time of the metrics consumers themselves so modelling their performance is not possible. However, as metrics processing is not required to meet a performance target and is only required to keep up with the metric production by the executors, setting the parallelism for the metrics consumer to match the number of worker processes is usually sufficient.

The InfluxDB metric consumer implementation we created takes advantage of the InfluxDB Java clients multi-threaded batch writing functionality to process all the metrics from a single bucket period, for all tasks on a given executor, into one batch and send that to the InfluxDB server. Once the batch is created and issued, a separate thread deals with the sending of the batch, freeing the consumer to work on the next batch.

Storm-Tracer metrics manager

The Storm metrics API allows users to register custom metrics which will be reported to the configured metrics consumer along with the default metrics. Storm provides interfaces for several common metrics types such as count or latency metrics. Below, we detail several custom metrics which are required to aid in the performance modelling process. Registering metrics requires several setup steps so, in order to reduce the load on topology creators, we created a Storm-Tracer metrics manager class that would automatically name, register and perform the other setup steps, as well as handle the logic for updating the metric. All that needs to be supplied in the bolt's `execute` method is the input tuple (which contains source task information).

Task-to-task transfer counts

As discussed in section 4.4, the execute count metrics used to calculate the routing probabilities only provide information on the incoming stream name and the source component. This is not an issue for shuffle grouped connections as these are load balanced and therefore the expected routing probabilities can be inferred from the number of downstream executors. However, to calculate the routing probabilities for fields grouped (key based) connections we need the task-to-task transfer count.

We have created a custom Storm metric which tracks the task-to-task transfers. The Storm-Tracer metrics manager mentioned above provides methods to log a transfer, which extracts the source task identifier from the tuple and updates its internal counts. Once every metrics bucket period these counts are reported to the metrics consumer and sent to the TSDB.

C.2.3 Cluster metrics

Storm provides a rich set of metrics by default, however these do not include the resource usage on the worker node. In order to access this information a real time monitor (RTM) program was created that collects metrics on the worker nodes and reports these to the TSDB. The code for the worker node monitoring program can be seen at the address below as well as in the digital media attached to this document:

https://github.com/tomncooper/tracer_rtm

The RTM is a self contained executable that can be configured with the TSDB connection information and a reporting period. Once every reporting period, which is 1 minute by default, the RTM will aggregate the metrics it monitors and issue them to the TSDB.

Early on in this research project, there was an intention to look at modelling resource usage as well as topology performance. As a result the RTM includes not only the metrics described below, but also the ability to extract central processing unit (CPU) load and random access memory (RAM) usage information[6] about each of the worker processes running on the worker node as well as some information about the executor threads within the worker process.

Transfer latencies

As discussed in section 4.11, accounting for the delay due to sending tuples across the network is an important factor in the modelling process. In order to be able to estimate the likely network delay we need to be able to measure the existing network delay between the worker nodes.

To do this the RTM uses the `fping`[8] program to issue Internet Control Message Protocol (ICMP) packets to a list of active worker node addresses. `fping` allows multiple packets to be sent to each host address and calculates a statistical summary once all have returned. This a prudent approach, as relying on a small number of measures leaves the measurements open to bias due to latency spikes. It also offers a significant usability advantage over the standard `ping` program as it asynchronously sends packets to the host addresses.

Each RTM instance sends a periodic *heartbeat* message to the TSDB. When the metric reporting period activates the RTM polls the TSDB for the unique set of host addresses active within a configurable period. This list of hosts is then used with the `fping` program and the round trip time statistics are parsed from the output of that program. These statistics are then issued to the TSDB and the RTM waits for the next reporting period to activate.

[6] This is done via the psutil[7] Python library.
[8] https://fping.org/

C.3 Interacting with Nimbus

Many aspects of the Storm-Tracer modelling system require interacting with the Nimbus node of the Storm cluster. As discussed in section 2.3.1, Nimbus is the central control node of the cluster and is where topologies are sent to be deployed and where changes to topology configurations are issued. Other than the Java based command line client used to issue and configure topologies, Storm does not provide Nimbus clients that can be used with external programs. However, the Nimbus server is based on Apache Thrift[9] which is a language agnostic system for creating services. It allows you to define your service end points, client methods and the messages that will pass between them in a definition file. You can then use the Thrift compiler to create program implementations in various languages. The Storm Thrift definition file provides definitions for a Nimbus client interface which is used by the Java command line client to issue commands to the Storm cluster. Using this definition file and the Python `thriftpy2`[10] library allows the modelling system to access all the functionality of the Nimbus server natively. This allows not only the automation of rebalance commands, used in the data gathering for the validation process described in section 5.2.1, but also access to information on the topologies currently running on the Storm cluster.

C.4 Topology Structure Storage and Analysis

One of the key features the modelling system must provide is the ability to query the proposed and currently running physical plans of the topologies it is modelling. The initial implementation of the modelling system used an in-memory representation of the topology physical plan in the form of a custom Python class, which wrapped a nested map structure representing the nodes and edges of the plan and contained methods for performing queries against the plan. Initially this was a reasonable approach, as the modelling system development started by using the logical plan as the basis for the tuple flow through the topology. As a better understanding of the operation of Storm's internal systems was developed and the tuple flow plan emerged as the more accurate representation, the old map based implementation became unwieldy. Performing queries to identify paths that included the worker process nodes, as well as incorporating worker nodes, meant changing many of the methods used to access the map.

In addition to this, the in-memory map based representation could only be constructed from a running topology via the Nimbus client (see section C.3). This meant that it would be difficult to compare and investigate multiple topology configurations simultaneously. In order to save copies of the map based topologies the class instance would have to be

[9] https://thrift.apache.org/
[10] https://github.com/Thriftpy/thriftpy2

serialised to disk and retrieve later, these multiple topology files would not be portable between different languages and machines and would require additional effort to organise and catalog them.

It became clear that storing these, now quite complex, directed graphs (DiGs) would require a more specialised system. Several options were considered, including using a specialised graph library such as Python's NetworkX[11] or Java's JGraphT[12] to provide the tuple flow plan storage. However, whilst these options possessed many graph analysis algorithms and easy setup, they had limited query flexibility and storing and querying across multiple graphs was limited or not possible. Eventually it was decided to utilise a dedicated graph database to store the tuple flow plans for currently running and proposed topology configurations.

C.4.1 Graph database

Graph databases are specifically designed for storage and querying of highly connected data. Examples of their use include analysing friendship networks on social media sites like Facebook (Ching et al., 2015) or Twitter (Gupta et al., 2013), to identifying relevant information related to a current search term like in Google's Knowledge Graph feature (Singhal, 2012).

Using a separate database to store the tuple flow plans allows the modelling system to abstract away the topology structure storage and query aspects of its operation. It also means that many different tuple flow plans for a given running topology could be stored and linked to their operation period which could then be used in conjunction with the TSDB to perform modelling on a particular topology configuration with ease. This would be useful for validation of the modelling approach but would also allow future systems to compare a proposed tuple flow plan to the most similar recorded topology tuple flow plan and use metrics from that period to make its predictions.

There are many graph databases available, however as with other systems used in this research, options were limited to open-source systems with a large user base. Neo4j[13] is one of the most popular graph databases, is open source, has a mature codebase and a multitude of client and extension libraries. Neo4j uses the labeled property graph model where nodes can have one or more *labels* (such as *Person* or *executor*) which allow them to be easily indexed and grouped. Nodes can also have multiple properties (essentially key/value pairs). Nodes are linked (in one or both directions) by relationships, that have a single label or type (such as *Knows* or *LogicallyConnectedTo*) and can also have multiple properties.

[11] https://networkx.github.io/
[12] https://jgrapht.org/
[13] https://neo4j.com/product/

C.4.2 Topology graph structure

For the tuple flow plan, the worker nodes, worker processes and executors can be represented as nodes, with appropriate labels for their respective types. Executors have properties such as component name and the range of task values they contain, worker processes have host name and port properties and worker nodes contain host name and IP address information. It was decided that the individual tasks did not need to be represented as separate nodes, as they are not dealt with individually in queries and are only ever interacted with via their containing executor. Separate node labels for spouts and bolts were not deemed necessary as this aspect could be added as a property to each executor node.

In order to distinguish one topology from another all elements of a particular tuple flow plan are linked to a root Topology node with a *BelongsTo* relationship:

```
CREATE (ex:Executor {component: "BoltA", start_task: 12, end_task: 15}),
       (ex)-[:BelongsTo]->(:Topology {id: "Test1"})
```

The above code is an example of the *Cypher*[14] query language that Neo4j uses to create, manipulate and query graphs. Nodes are represented in parentheses, relationships use square brackets with hyphens and greater than or less than signs to indicate directionality, the labels for each element are prefaced by a colon.

Linking all elements to a root topology node means that they can easily be retrieved using a query like the one below, which retrieves all the bolt executors for a specific topology:

```
MATCH (t:Topology {id: "Test1"})<-[:BelongsTo]-(ex:Executor {type: "bolt"})
RETURN ex
```

Physical plan

The physical plan of the topology can be represented by linking the various Storm element nodes with an *isWithin* relationship. For example:

```
MATCH (ex:Executor {component: "BoltA", start_task: 12, end_task: 15}),
      (wp:WorkerProcess {hostname: "Worker1", port: 6700}),
      (wn:WorkerNode {hostname: "Worker1", ip_address: "57.178.12.17")
CREATE (ex)-[:isWithin]->(wp)-[:isWithin]->(wn)
```

The physical plan can then be queried by using these relationships to limit the search to only the relevant parts of the graph. For example, the IP address of the worker node that a given executor is in, so that the network latency can be retrieved from the TSDB, can be found using the following query:

```
MATCH (ex:Executor {component: "BoltA", start_task: 12, end_task: 15}),
```

[14] https://neo4j.com/developer/cypher-query-language/

```
      (ex)-[:isWithin]->(:WorkerProcess)-[:isWithin]->(wn:WorkerNode)
RETURN wn.ip_address
```

Logical plan

The logical plan of the topology can be represented by connecting the executors of the topology with a *LogicallyConnected* relationship. This relationship can also contain the stream name, type of grouping that exists between the executors and whether the connection is local, inter-local or remote:

```
MATCH (spout:Executor {component: "SpoutA", start_task: 88,
                       end_task: 89, type: "spout"}),
      (exa:Executor {component: "BoltA", start_task: 12, end_task: 15}),
      (exb:Executor {component: "BoltB", start_task: 27, end_task: 30}),

CREATE (spout)-[:LogicallyConnected {stream: "sentences",
                                     grouping: "shuffle"
                                     type: "Local"}]->
       (exa)-[:LogicallyConnected {stream: "words",
                                   grouping: "fields"
                                   type: "remote"}]->(exb)
```

The logical plan can then be queried using these relationships. For example, finding all the downstream executors from a source, along with their connecting stream name and groupings, can be done with the following query:

```
MATCH (source:Executor {component: "BoltA", start_task: 12, end_task: 15}),
      (source)-[lc:LogicallyConnected]->(destination:Executor)
RETURN destination, lc.stream, lc.grouping
```

If you wanted to retrieve the query plan of the topology, this can be done from the logical plan by returning distinct component names along with the distinct stream and grouping information.

Tuple flow plan

The tuple flow plan of the topology can be represented by connecting the logically connected executors with a *PhysicallyConnected* relationship via the intermediate worker process hosting the source executor (which can be found using a nested query):

```
MATCH (source:Executor)-[lc:LogicallyConnected]->(destination:Executor)
WHERE lc.type != "local"
WITH source, lc, destination
```

```
MATCH (source)-[:isWithin]->(wp:WorkerProcess)
CREATE (source)-[:PhysicallyConnected]->
       (wp)-[:PhysicallyConnected)->(destination)
```

The paths tuples follow through the topology can now be found by restricting searches to physically connected relationships.

C.4.3 Constructing the topology graphs

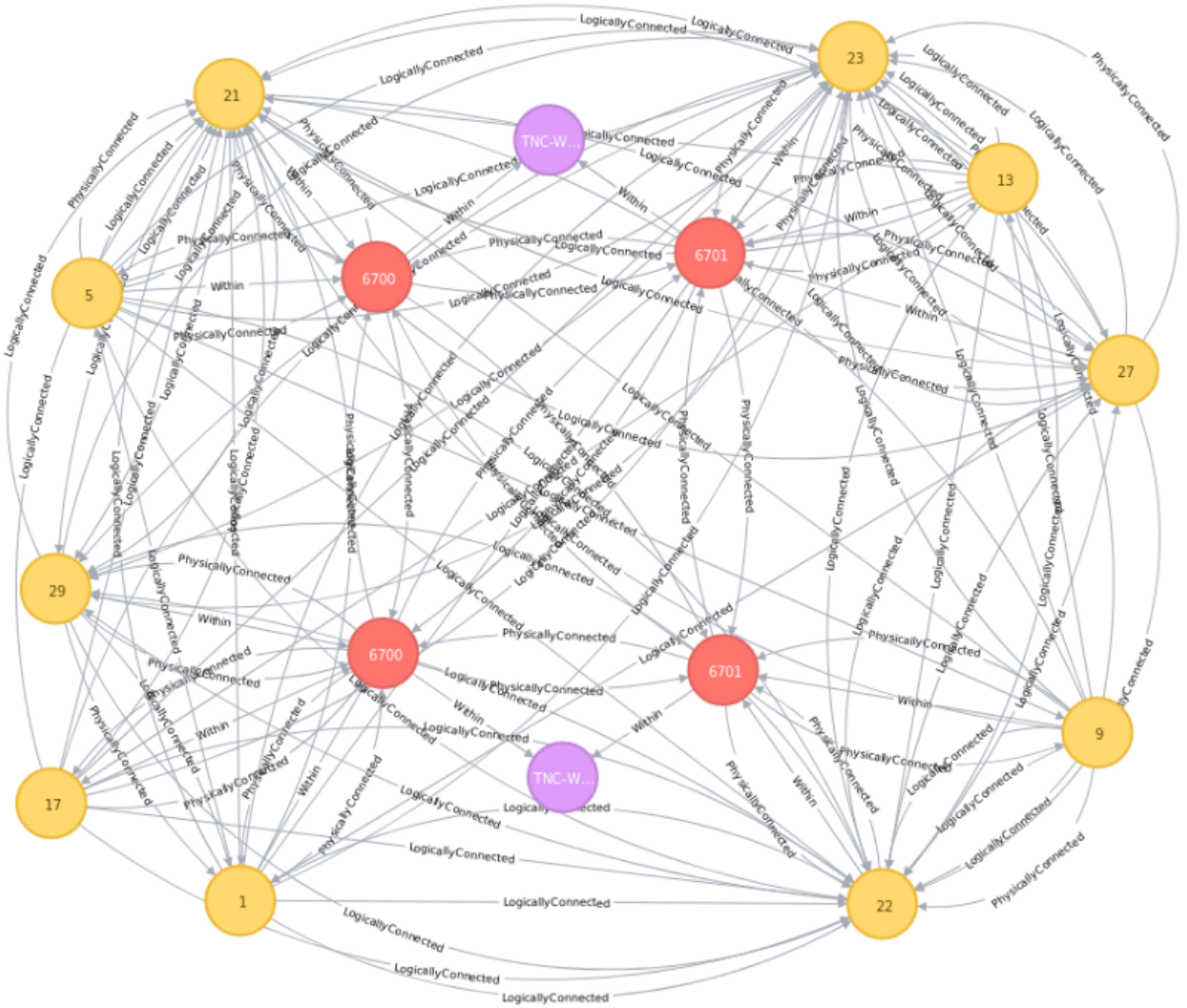

Figure C.2: The tuple flow plan as represented in the Neo4j graph database.

The Cypher queries shown above are simplified examples. Cypher provides much more efficient ways to bulk create the graph structures described above. These can be seen in the `graph` module of the Storm-Tracer code base. However, the basic process involves first extracting the physical plan representation from the Nimbus API (see section C.3) and then adding all the worker node, worker process and executor nodes to the graph. Once this is done, bulk create statements create the various relationships starting with the physical plan so that properties like whether a logical connection is remote or not can be inferred by looking at whether executors are on the same worker process and/or worker

node. Figure C.2 shows what a simple topology looks like when fully constructed with all the relationships described above. This is from Neo4j's built in visualisation tool and shows how even reasonably simple topologies can quickly become difficult to interpret[15]. This highlights the advantage of abstracting this complexity to the graph database, rather than managing it ourselves.

C.5 Modelling Implementation

The author would like to make clear, before going into the details of the system, that they make no claim to this system being a production ready or performant implementation of a DSPS modelling system. Nor is this an example of software engineering best practice[16]. This is research code that grew organically as various modelling approaches were attempted and refined. The author would ask that the reader keeps that in mind when reviewing the code described below. That being said, chapter D details a DSPS performance modelling system that was, as part of the author's internship at the social media company Twitter (the creators of Apache Storm and Heron), built from the ground up to be used in production environments and implemented many of the lessons learned whilst creating the Storm-Tracer system.

The Storm-Tracer modelling system is written in Python. This language was chosen for its ease of use and large eco-system of third party libraries including data manipulation (`pandas`[17]) and scientific (`scipy`[18]) libraries as well as ready availability of clients for the various external systems mentioned above. Storm-Tracer is divided into several modules[19] that deal with different aspects of the modelling process and these are broken down into more detail in the sections below.

C.5.1 Metrics

The metrics module contains classes and functions which can interface with the TSDB and extract time series or statistical summary information for the various performance metrics forwarded by the Storm metrics consumer.

[15] We refer to this as the topology "hairball" plan.
[16] Indeed, there could be a convincing argument made for the reverse being true.
[17] https://pandas.pydata.org/
[18] https://www.scipy.org/
[19] These modules can be seen in the Storm-Tracer repository available on the digital media attached to this document. The modules consist of folders containing Python source files which hold the various functions.

C.5.2 Graph

The graph module contains functions for interfacing with the Neo4j graph database. There are methods for constructing graphs of currently running topologies from the data provided by the Nimbus server and creating graphs of proposed physical plans issued by the Storm scheduler (via the API module).

C.5.3 API

The API module contains classes and functions for receiving proposed physical plans from the custom scheduler implementation. The code for this scheduler can be seen in the repository for the metrics gathering system described in section C.2.2. The scheduler simply creates a new physical plan using a round-robin approach used by Storm's default scheduler, but instead of deploying it will package the plan up as a protocol buffer[20] (a technology similar to Thrift that is used by the Heron DSPS as its internal message passing system) and send it to an end point associated with the modelling service. The code in the API module converts the protocol buffer physical plan into a Python object which is used by the Graph module in creating graphs of the proposed physical plans.

C.5.4 Storm

The storm module contains the Nimbus client interface code described in section C.3. This also contains a copy of Storm's Thrift definition file which is used in conjunction with the `thriftpy2` library to access the Nimbus services.

C.5.5 Modelling

The modelling module contains methods which implement the modelling approaches laid out in chapter 4, including the routing probability, I/O ratio, arrival rate, service time and incoming tuple list size estimations.

Queue simulation

Both the full user logic thread (ULT) simulation and simplified simulation methods, described in section 4.2.1, are implemented. The full system simulator uses a Python class to encapsulate the state described in section B.2 and also provide methods for updating the state in line with algorithms 1, 2 and 3. A main simulation method then wraps this class and calls the appropriate state update method according to which random event has occurred. The simple ULT simulator has a reduced state and so is implemented as a single method.

[20] https://developers.google.com/protocol-buffers/

The full simulator was used during model development, however simulating each executor in the topology is a significant bottleneck in the modelling pipeline. As each simulator is independent from the other, multi-processing allows the set of executors to be shared between the available cores on whatever machine is running the modelling code. Whilst this does offer a speed up proportional to the number of cores, the individual simulations themselves can take a significant amount of time. Therefore, in order to reduce the simulation time to as short as possible, the simplified simulator was reimplemented in Cython[21]. The Cython language is a superset of the Python language that additionally supports calling C functions and declaring C types. This allows the compiler to generate very efficient C code from Cython code which is also accessible natively from Python code. This meant the C version of the simulator could be easily integrated into the rest of the Python based modelling pipeline.

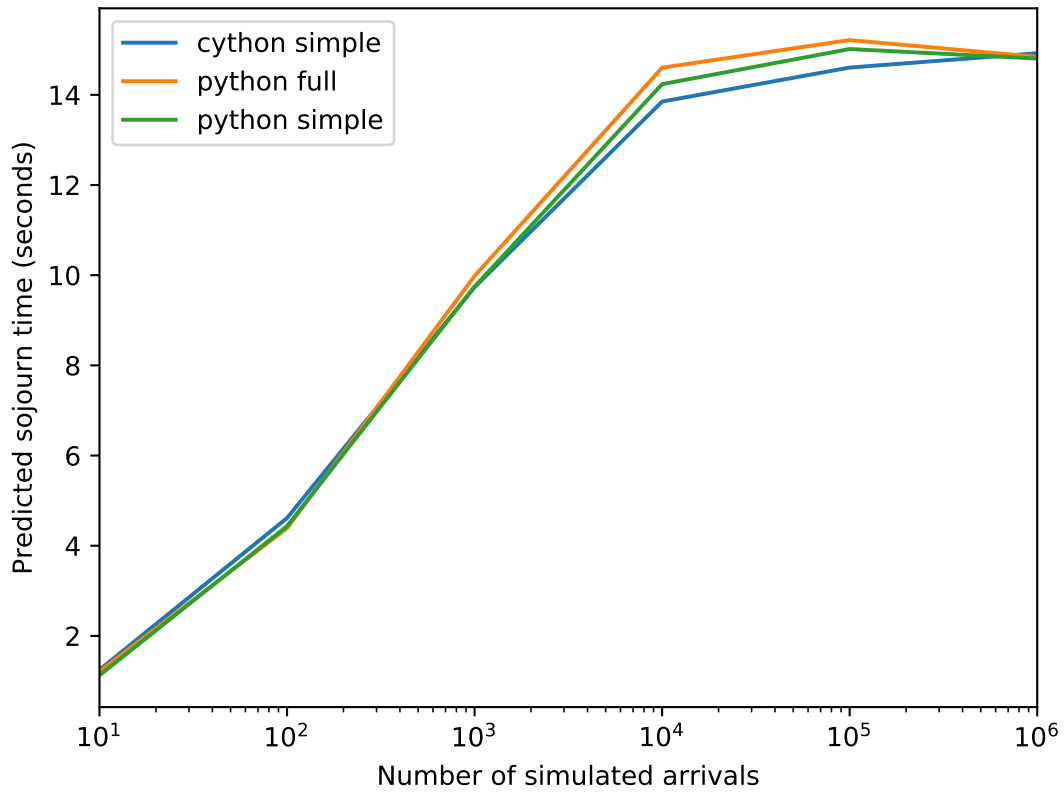

Figure C.3: Comparison of the predicted sojoun time for the three ULT simulator implementations.

In order to validate that both the Python and Cython based simplified simulators were producing results consistent with the full simulator, several comparisons were performed using the same modelling parameters. The accuracy of the simulator depends greatly on the number of simulated arrivals, due to the random nature of the event generation within

[21] https://cython.org/

the simulator and the need for sufficient arrivals to trigger certain internal behaviour. Figure C.3 shows the output of all three simulator implementations with fixed parameters and varying numbers of simulated arrivals. This plot clearly shows that all three simulators produce consistent results and that after approximately 1 million simulated arrivals they all begin to converge on a constant predicted value. The accuracy of these simulated results is discussed in chapter 5. The processing time for each of the simulators is shown in figure C.4 and shows how, despite the fact that these simulators deliver similar results, the time taken to deliver them is significantly different. The Cython based simulator is an order of magnitude faster than the pure Python implementation and two orders of magnitude faster (0.4 sec compared to 23.5) than the full system simulator.

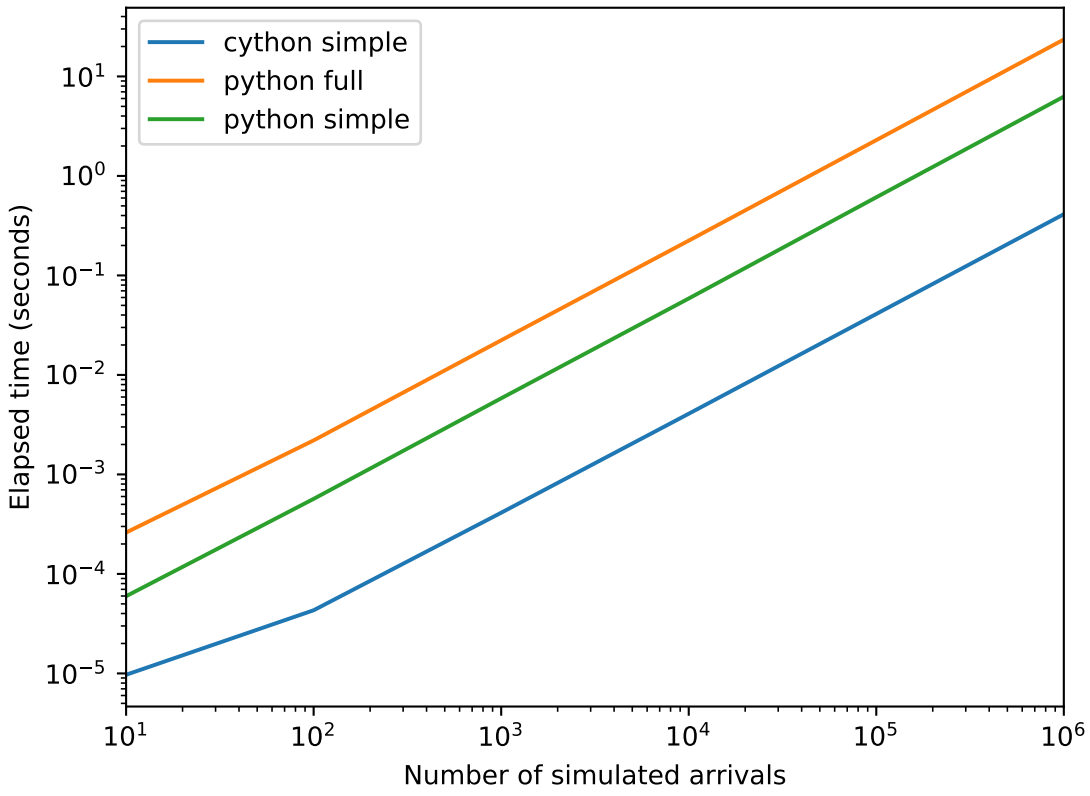

Figure C.4: Comparison of the average elapsed time to complete the number of simulated arrivals for the three ULT simulator implementations.

Given the testing shown in the figures above, the Cython based simplified ULT simulator, with 1 million simulated arrivals, was used in the evaluations described in chapter 5.

Performance

The performance module contains the code which pulls together all the other elements of the modelling code into single functions which are able to predict the end-to-end latency of a proposed physical plan. It also contains variations of these methods for predicting

the end-to-end latency of a currently running topology physical plan using a different incoming workload.

Appendix D

Caladrius

D.1 Background

In the summer of 2016 the author attended the Distributed Event Based Systems (DEBS)[1] conference in Los Angeles, California. They where there to present their research project as part of the conference's Doctoral Symposium (Cooper, 2016).

The keynote speaker at DEBS that year was Karthik Ramasamy[2], who at the time was the Engineering Manager for Twitter's Real Time Compute team, which handled their stream processing systems. His talk[3] focused on Twitter's replacement of Apache Storm with a new system called Heron[4]. At the end of his talk he listed some continuing challenges in stream processing engineering, one of which was scaling the Heron topologies to the correct size to meet the expected incoming workload. Their engineers were spending days and even weeks tweaking their production topologies. This issue, as outlined in section 1.2, could be addressed by the approach we were proposing at the conference (Cooper, 2016). After several months of correspondence with Mr Ramasamy and other members of the Twitter Real Time Compute team, the author secured a three month internship at Twitter's headquarters in San Francisco.

The aim of this internship was to see if the approaches outlined in chapter 4 for Apache Storm could be used to create a performance modelling system for Heron topologies. The result of this work was *Caladrius*, an open source framework for performance modelling of distributed stream processing systems (DSPSs). Section D.2 gives a brief overview of how Heron differs from Apache Storm. Section D.3 describes how the differences in operation between the two systems affect the performance modelling approach. Section D.4 describes the internal layout and operation of the Caladrius system and section D.5 summarises the

[1] https://www.ics.uci.edu/~debs2016/
[2] https://www.linkedin.com/in/kramasamy/
[3] https://www.ics.uci.edu/~debs2016/keynote-speakers.html#ramasamy
[4] http://heronstreaming.io

outcomes and further work on this project.

D.2 Heron Architecture

The Heron DSPS was developed by Twitter as a replacement for Apache Storm, which had served for many years as the low latency processing aspect of their Lambda Architecture based data processing system (with Apache Hadoop providing the slower, more accurate batch processing aspect). Heron was designed to address several shortcomings in Apache Storm (Kulkarni et al., 2015), namely:

- The highly multi-threaded nature of Storm's architecture, which makes it very hard to debug issues and identify which element is the source of a given error. For example, an exception in a single task (see section 2.4.2) will cause its host executor (see section 2.4.1) and worker process (see section 2.4.3) to crash along with all the other executors it hosts. It is also very hard to identify which task within an executor is the source of a particular error.
- The multi-threaded nature of Storm means that isolating the resource (CPU, RAM, etc.) usage of a particular task is very difficult.
- Resource allocation can only be performed at the worker process level and is homogeneous across all worker processes assigned to a topology. This often leads to over-provisioning of resources as some worker processes will be assigned resources to cope with the worst case parallelism (being assigned a lot of resource intensive executors), when they do not require them.
- Storm's Nimbus control server represents a single point of failure for the system. Without this node topologies cannot be started, stopped or altered.
- Lack of support for back-pressure, which is a mechanism for telling upstream executors to stop or slow the flow of tuples to overloaded downstream executors.
- Worker processes with a large number of executors can suffer high contention for the worker process transfer queue (WPTQ).

Heron was designed to address the above issues by moving away from the highly nested model (tasks → executors → worker processes) used by Storm. It uses a more isolated structure, where each executor is within its own operating system (OS) process (referred to as an *instance*) and only a small number of these instances are hosted within a container (the equivalent of Storm's worker process). The advantage to this is that the instances provide much better isolation for the purposes of debugging and gauging resource usage. The use of containers to host the instances also allows finer grained control of the allocation of resources. Each container runs a *Stream Manager* process which is responsible for routing all the traffic into and out of the instances on that container. Figure D.1 shows an illustration of a simple two container Heron cluster. The arrows

indicate meta-data/configuration messages and the red indicate tuple transfers. Heron clusters are designed to be run on modern container orchestration systems (such as Apache Mesos[5] or Kubernetes[6]) and the typical approach is to use a small number of instances per container so as not to cause issues with queue contention at the Stream Managers.

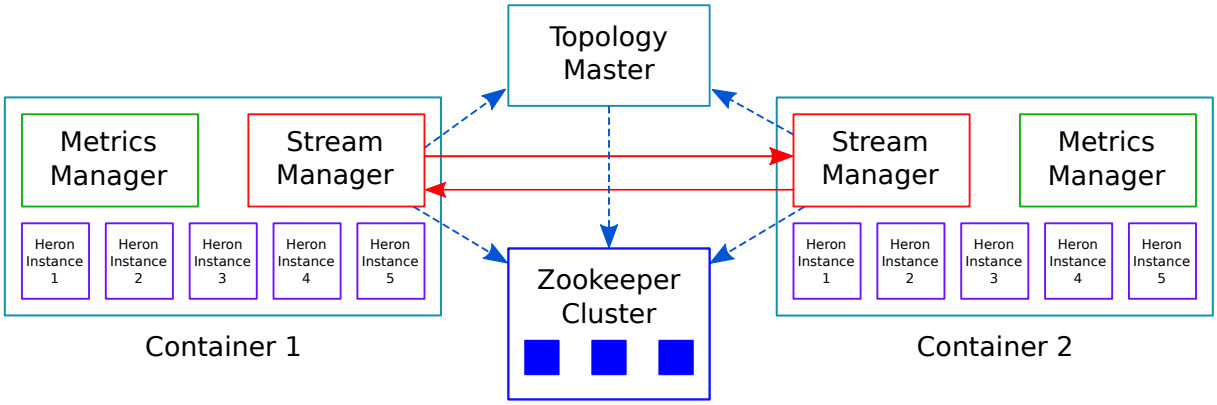

Figure D.1: A two container Heron cluster.

D.2.1 Differences to Storm

Heron is backwards compatible with Storm topologies and at a high level many of the plan types that a topology can be represented as are equivalent. For example, the query plan and logical plan are the same as for Apache Storm topologies, as is the topology configuration (with the number of worker processes replaced with the number of assigned containers). The difference comes at the physical plan level and this also has implications for the tuple flow plan.

Physical connection types

In Heron **every** tuple issued by an instance must pass through the stream manager's queue and this means there is no direct message passing (local physical connection) between instances. All tuples must be serialised to be sent to the stream manager and de-serialised when arriving at the instances. This means there there are only inter-local and remote physical connections within Heron topology tuple flow plans.

Fields groupings and state

One of the motivations when designing Heron was to simplify the physical elements of the system and improve the scaling performance of the topologies. To this end, the developers removed Storm's state space partitioning via tasks (see section 2.4.2) and simply numbered each Heron instance when a topology is first deployed and would renumber them whenever

[5] http://mesos.apache.org/
[6] https://kubernetes.io/

a rebalance (called a topology *update* in Heron) occurred. These instance numbers were used, as in Storm, to route tuples with the same field values to the same Heron instances.

The removal of the tasks meant that there was no longer a maximum parallelism limit (imposed in Storm by the tasks-per-component setting) for Heron topology components, which was seen as an advantage by Heron's developers. However, as the instance numbers are not fixed and can change after a rebalance, this means that the number of destination instances for a fields grouped connection can change. The implication being that tuples with a certain field value in a source physical plan cannot be guaranteed to be routed to the same numbered instance in a proposed physical plan, as the identifier of that instance may have changed.

This design decision also meant that recovering state after a rebalance or failure — something trivial in Storm, as all state can be indexed by the persistent task ID — was much harder. At the time of the author's internship this was an area of active study by the Heron developers and there was a suggestion of re-introducing Storm-style task based state partitioning.

Back-pressure

A key design feature of Heron is its back-pressure system, which is built into the stream managers that handle communication between the containers that host the Heron instances. When the queues attached to the Heron instances reach a high watermark, they issue a message to their host stream manager that they are overloaded. The host stream manager will then analyse the topology to identify which spouts are feeding this particular instance and tell them to stop issuing tuples. Once the overloaded Heron instance's queue drops below a low watermark the host stream manager will tell the spout to begin issuing tuples again.

D.3 Modelling Heron

Initially, the goal of the internship was to directly apply the prediction methods developed for Apache Storm (see chapter 4) to Heron. The aim was to create a prototype service which was able to intercept a proposed topology physical plan (called *packing plans* in Heron) and model its expected performance.

However, as section D.2.1 makes apparent, there are significant differences between how Heron and Storm handle the partitioning of state and the routing of tuples for fields grouped connections. These differences mean that the routing probability prediction methods, described in section 4.5, are not applicable. This is because the routing probabilities calculated for the instances in a source physical plan cannot be used to predict those in a

proposed physical plan.

At the start of the internship there was some discussion of helping to implement Storm-style state partitioning for Heron. However, it was decided that this would take longer than the internship period to complete. Therefore the focus of the project was changed. Instead of predicting the performance of proposed physical plans, the aim would be to create a system that could predict the future incoming workload into a running topology and use this workload to see if it would trigger back-pressure.

The triggering of back-pressure was a significant concern to the users of Heron within Twitter, as it would often require manual intervention to rectify. Therefore, having a system that could warn of issues ahead of time, allowing users to configure the topology to prevent them, was seen as a valuable contribution.

D.3.1 Incoming workload

The advantages to predicting incoming workload for DSPS were laid out in section 1.3.3. In section 4.3 we described how performing this aspect of the modelling was difficult for proposed physical plan and would require large amounts of historical data in order to build accurate models. For our work with Apache Storm we left the prediction of incoming workload to future research (see section 6.3.2). However, for Heron we were only concerned with modelling currently running topologies, so predicting the resulting output due to changes in the parallelism of the spouts was not required. Furthermore, Twitter had an extensive metrics database, including very large production topologies that had been running for many months. This provided a wealth of data to aid in the workload prediction process.

In order to speed up the creation of a viable prototype system, an *off the shelf* load prediction package was used. Facebook's open source *Prophet*[7] software is able to perform workload predictions using a generalised additive model approach to take account of both trends and seasonality in time series data (Taylor & Letham, 2018). This was well suited to the seasonal workloads that Twitter experienced, where users tended to tweet more in the morning and evening.

For each topology, a Prophet model was trained on the emit-count time series data for each spout instance. Typically, several days of data were used in order to capture as much seasonal variation as possible. A forecast of the output from each spout instance, for several different time horizons, could then be made each with its own measure of uncertainty in the prediction.

[7]See: https://facebook.github.io/prophet/

D.3.2 Arrival rates

In order to estimate the effect of a predicted incoming workload on a topology's instances, a similar process to the arrival rate prediction method described in section 4.8 was used. Therefore, predictions of the stream routing probabilities (SRPs) and input to output (I/O) ratios of each instance were required.

Although the current design of Heron prevented the prediction of the routing probabilities for proposed physical plans, those for currently running topologies could be calculated by adding custom metrics to the topologies components that would provide instance-to-instance transfer counts (the equivalent of the task to task routing probabilities (TRPs) in Storm). Alternatively, the routing probabilities could be calculated by using a process similar to the estimated task output proportion (ETOP) calculations discussed in section 4.5. However this second method, whilst allowing unmodified topology to be used, would only work with topologies that did not have consecutive fields grouped connections.

The instance I/O ratios were calculated in a similar manner to that described in section 4.6. For each output stream of a given instance, a least squares regression approach was used to create a set of coefficients for the input streams to that instance.

Once the routing probabilities and I/O ratios were calculated, a breadth-first traversal of the topology's logical plan was performed and the predicted output from each spout propagated according to the calculated routing probability and I/O ratio coefficients. This resulted in a predicted arrival rate at each instance in the topology.

D.3.3 Back-pressure prediction

Once an arrival rate for each instance of the topology was calculated it could be compared to the service rate of each instance. If any instance had an arrival rate higher than its service rate, then the Caladrius could raise an alert that back-pressure could be expected.

The service rates, in this case, were calculated by using the median of the service time (based on the distribution taken over the same time period used for the workload forecasting). As with the prediction of service times for Storm topologies, described in section 4.9, this estimation of the service time was subject to the issues of a multi-processing environment. However, the isolation of the instances within their own OS processes and the fact that this was not a prediction of service time under a different topology configuration made this less of a factor.

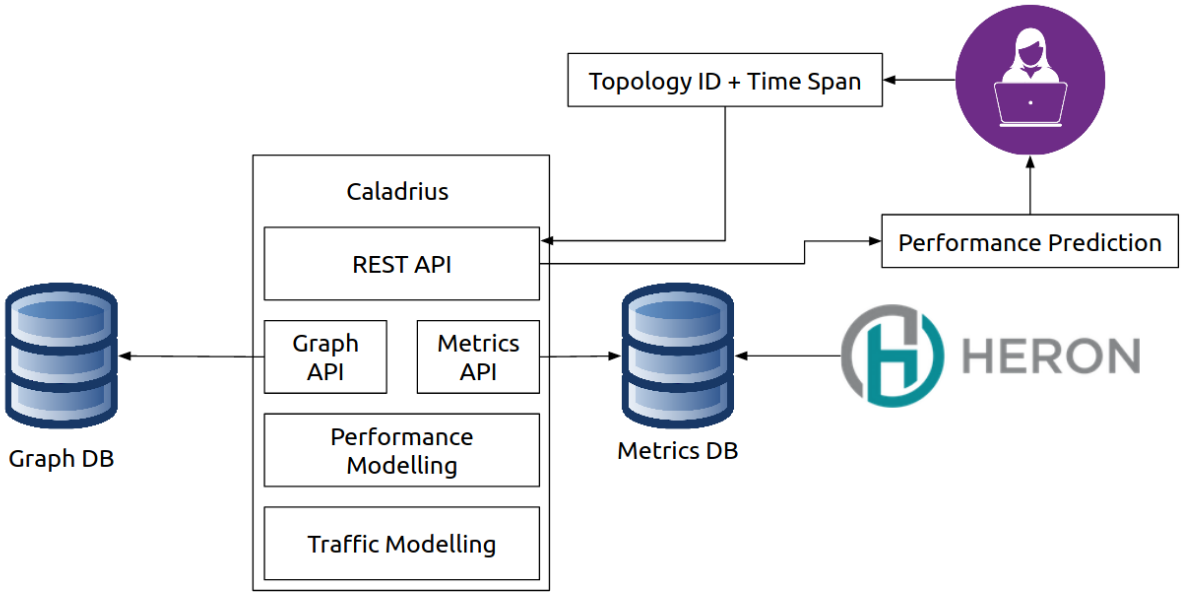

Figure D.2: The Caladrius modelling system.

D.4 Caladrius Implementation

Caladrius is designed to be a general performance modelling framework for DSPSs. It provides several services useful for this task including an interface for a graph database, used to store topology structure information (logical plans, physical plans and tuple flow plans), and interfaces to metrics databases. It provides base model classes for incoming workload (traffic) and topology performance predictions.

Caladrius provides a REST API by which a user can request a performance prediction for a given topology running on a their cluster. Caladrius then handles the modelling asynchronously and stores the result in a local file or database. Figure D.2 shows a diagram of the Caladrius system. In its current form it provides modelling and metrics interface classes for Heron topologies, however modelling classes for other DSPSs can be configured and the REST endpoints provided.

The code for Caladrius was open sourced by Twitter at the end of the author's internship and is currently hosted at: https://github.com/twitter/caladrius

D.5 Outcomes

Caladrius' performance was validated against several production topologies running on Twitter's internal Heron cluster. The workload predictions were validated over several time horizons, and custom test topologies were used to evaluate the back-pressure predictions. These custom topologies had instances with very low service rates and under testing they used artificially high arrival rates to ensure back-pressure occurred.

Unfortunately, due to the workload traces and topology structures being considered as commercially sensitive information, the results of this evaluation could not be released outside of Twitter. However, a brief summary was permitted.

Regarding workload prediction, those topologies which were attached to human-facing systems (those reacting to tweets directly) showed good prediction accuracy due to their strong seasonality. However, those topologies which were attached to other automated systems showed poorer results, as these types of machine-to-machine communications showed workload patterns that were more "bursty" in nature. This bursty pattern was poorly suited to the modelling approach employed by the Prophet package.

Where the workload predictions were good, the back-pressure predictions showed high accuracy and a viable prototype alert service was demonstrated for the custom test topologies.

D.5.1 Further development

After the author finished their internship Twitter was keen to continue the development of Caladrius. Another intern, a PhD student researching DSPSs, used Caladrius as a base for an investigation of through-put prediction for Heron topologies. They, the author and several Heron developers from inside Twitter collaborated on a research paper detailing this work (Kalim et al., 2019).

The through-put prediction research significantly altered the direction of Caladrius' development and a fork of the codebase (called Magpie), from the end of the author's internship (without any of the later code changes), can be seen at: https://github.com/tomncooper/magpie

It is the author's intention to eventually implement model classes and workload predictions, in Magpie, for Apache Storm using the methods laid out in chapter 4.

Appendix E

Experimental Configurations

The settings for each of the topologies used in the performance modelling evaluation, described in chapter 5, are shown below. The code for these topologies can be seen in the *StormTimer* repository available on the digital media attached to this document or at:

`https://github.com/tomncooper/StormTimer`

E.1 Fields to Fields

E.1.1 Experimental steps

Table E.1: Parallelism configuration for each step of the fields to fields test topology.

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Kafka Spout	1	2	4	8	16
JSON Bolt A	1	2	4	8	16
JSON Bolt B	1	2	4	8	16
Sender Bolt	1	2	4	8	16

E.1.2 Experiments

F2F-1

- Worker nodes: 2
- Worker processes: 4
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 16

F2F-2

- Worker nodes: 4
- Worker processes: 8
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 16

E.2 Multiplier

E.2.1 Experimental steps

Table E.2: Parallelism configuration for each step of the multiplier test topology.

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Kafka Spout	1	2	4	8	16
Multiplier JSON Bolt	1	2	4	8	16
Sender Bolt	1	2	4	8	16

E.2.2 Experiments

Multiplier-1

- Worker nodes: 2
- Worker processes: 4
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 16
- Multiplier settings: 10

E.3 Windowed

E.3.1 Experimental steps

Table E.3: Parallelism configuration for each step of the windowed test topology.

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Kafka Spout	1	2	4	8	16
Windowed JSON Bolt	1	2	4	8	16

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Sender Bolt	1	2	4	8	16

E.3.2 Experiments

Windowed-1

- Worker nodes: 2
- Worker processes: 4
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 16
- Window length (tuples): 10

E.4 All-in-one

E.4.1 Experimental steps

Table E.4: Parallelism configuration for each step of the all-in-one test topology.

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Kafka Spout	1	2	4	8	16
Multipler JSON Bolt	1	2	4	8	16
Windowed JSON Bolt	1	2	4	8	16
Sender Bolt	1	2	4	8	16

E.4.2 Experiments

All-In-One-1

- Worker nodes: 3
- Worker processes: 6
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 50
- Multiplier settings (min, max, mean, std): (1, 20, 10, 1.0)
- Window length (tuples): 10

E.5 Join and split

E.5.1 Experimental steps

Table E.5: Parallelism configuration for each step of the join-split test topology.

Component	Step 0	Step 1	Step 2	Step 3	Step 4
Kafka Spout	1	2	4	8	16
JSON Bolt	1	2	4	8	16
Multipler JSON Bolt	1	2	4	8	16
Join-Split Bolt	1	2	4	8	16
Sender Bolt	1	2	4	8	16

E.5.2 Experiments

Fish-1

- Worker nodes: 4
- Worker processes: 8
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 50
- Multiplier settings (min, max, mean, std): (1, 20, 10, 1.0)
- Join window length (tuples): 100
- Join Split Bolt setting: one tuple out per window
- I/O Ratio estimation bucket length (secs): 4

Fish-2

- Worker nodes: 4
- Worker processes: 8
- Tasks per component: 16
- Metric bucket period (secs): 2
- Message input rate (mps): 50
- Multiplier settings (min, max, mean, std): (1, 20, 10, 1.0)
- Join window length (secs): 2
- Join Split Bolt setting: Stream 3 if input stream counts are both even or both odd Stream 4 otherwise.
- I/O Ratio estimation bucket length (secs): 4