SATISFYING SERVICE LEVEL OBJECTIVES IN STREAM PROCESSING SYSTEMS

BY

FARIA KALIM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Indranil Gupta, Chair
Professor Klara Nahrstedt
Assistant Professor Tianyin Xu
Dr. Ganesh Ananthanarayanan

Abstract

An increasing number of real-world applications today consume massive amounts of data in real-time to produce up to date results. These applications include social media sites that show top trends and recent comments, streaming video analytics that identify traffic patterns and movement, and jobs that process ad pipelines. This has led to the proliferation of stream processing systems that process such data to produce real-time results. As these applications must produce results quickly, users often wish to impose performance requirements on the stream processing jobs, in the form of service level objectives (SLOs) that include producing results within a specified deadline or producing results at a certain throughput. For example, an application that identifies traffic accidents can have tight latency SLOs as paramedics may need to be informed, where given a video sequence, results should be produced within a second. A social media site could have a throughput SLO where top trends should be updated with all received input per minute.

Satisfying job SLOs is a hard problem that requires tuning various deployment parameters of these jobs. This problem is made more complex by challenges such as 1) job input rates that are highly variable across time e.g., more traffic can be expected during the day than at night, 2) transparent components in the jobs' deployed structure that the job developer is unaware of, as they only understand the application-level business logic of the job, and 3) different deployment environments per job e.g., on a cloud infrastructure vs. on a local cluster. In order to handle such challenges and ensure that SLOs are always met, developers often over-allocate resources to jobs, thus wasting resources.

In this thesis, we show that *SLO satisfaction can be achieved by resolving (i.e., preventing or mitigating) bottlenecks in key components of a job's deployed structure*. Bottlenecks occur when tasks in a job do not have sufficient allocation of resources (CPU, memory or disk), or when the job tasks are assigned to machines in a way that does not preserve locality and causes unnecessary message passing over the network, or when there are an insufficient number of tasks to process job input. We have built three systems that tackle the challenges of satisfying SLOs of stream processing jobs that face a combination of these bottlenecks in various environments.

We have developed Henge, a system that achieves SLO satisfaction of stream processing jobs deployed on a multi-tenant cluster of resources. As the input rates of jobs change dynamically, Henge makes cluster-level resource allocation decisions to continually meet jobs' SLOs in spite of limited cluster resources. Second, we have developed Meezan, a system that aims to remove the burden of finding the ideal resource allocation of jobs deployed on commercial cloud platforms, in terms of performance and cost, for new users of stream processing. When a user submits their job

to Meezan, it provides them with a spectrum of throughput SLOs for their jobs, where the most performant choice is associated with the highest resource usage and consequently cost, and vice versa. Finally, we have built Caladrius in collaboration with Twitter that enables users to model and predict how input rates of jobs may change in the future. This allows Caladrius to preemptively scale a job out when it anticipates high workloads to prevent SLO misses. Henge is built atop Apache Storm [7], while Meezan and Caladrius are integrated with Apache Heron [112].

Acknowledgments

I would like to thank my advisor, Professor Indranil Gupta, for his unfailing support and guidance. I have benefited enormously from his immense knowledge and constant motivation during the course of my Ph.D. study. He has been a mentor both professionally and personally, and has given me opportunities that I could not have ever had without his help. I am incredibly grateful for the privilege of having been his student.

Additionally, I would like to thank my committee members, Prof. Klara Nahrstedt, Prof. Tianyin Xu and Dr. Ganesh Ananthanarayanan, for providing insightful feedback that helped shape my thesis. I have always looked up to Klara as the role model of an impactful female researcher in STEM. Meeting Tianyin was a turning point during grad school: I always found his optimism and energy uplifting, and am going to miss his encouragement outside school. Ganesh's work has always set the bar for me, for what cutting-edge research in systems and networking should look like, and I have tried to meet it as best as I could.

I have also had the chance to meet great collaborators during my journey. Asser Tantawi at IBM Research emphasized the importance of building systems on strong theoretical grounds. The Real-Time Compute team at Twitter (especially Huijun Wu, Ning Wang, Neng Lu and Maosong Fu) gave me great insight into streaming systems and were a massive help with Caladrius. I would like to express my sincere gratitude to Lalith Suresh, who was my internship mentor at VMware Research, and gave me the opportunity to work on incredibly exciting research. I will always be inspired by his kindness towards me, his meticulous approach towards research, and his high standards for system building and emphasis on reproducibility.

I have had the privilege to make amazing friends in Urbana-Champaign. I will always be inspired by Farah Hariri and Humna Shahid, who have set an example for what great strength of character and selflessness looks like. I am indebted to Sangeetha Abdu Jyothi for her buoyant positivity, advice and help, in and out of grad school. Sneha Krishna Kumaran and Pooja Malik have been close friends through thick and thin. I am also thankful to Huda Ibeid, who is marvelously personable and broke many barriers for me. I am incredibly grateful to serendipity for giving me the chance to meet these amazing women in grad school and I will treasure their friendship forever. A special thanks to Mainak Ghosh, Rui Yang, Shadi Abdollahian Noghabi, Jayasi Mehar, and Mayank Bhatt for their kindness and support during my lows. I would also like to thank Le Xu, Beomyeol Jeon, Shegufta Ahsan Bakht, Safa Messaoud and Assma Boughoula for all the wonderful times we have spent together.

I am grateful to the Computer Science Department at University of Illinois, Urbana Champaign,

for giving me the opportunity to pursue a doctorate at this amazing institution. Many thanks to the staff of the department, including but not limited to: Viveka Perera Kudaligama, Samantha Smith, Kathy Runck, Kara MacGregor, Mary Beth Kelley, Maggie Metzger Chappell and Samantha Hendon for their help and support at countless times during my Ph.D. process.

I am thankful to the National Science Foundation, Microsoft, and Sohaib and Sara Abbasi for funding my work. I will always look up to Sohaib Abbasi as a role model, who generously gave back to his country by establishing a fellowship for Pakistani students pursuing higher education. He is an inspiration and I hope I can demonstrate a generosity equal to his in my career.

I would like to extend my gratitude to Dr. Usman Ilyas Dr. Tahir Azim, and Dr. Aamir Shafi, who were my mentors at my undergraduate alma mater, at National University of Science and Technology, Pakistan. My journey would not have started without their support.

Finally and most importantly, I would like to thank my family for their unconditional love and support. My parents' faith in me gives me the courage to do things I could otherwise not even contemplate. They are exemplary people who have fought many difficult battles in their life, have served their country well and have made countless sacrifices for their children. I hope that my efforts make them proud. Umar Kalim is a pillar of support, guidance and empathy, who I will always need. Umairah Kalim has always been a source of warmth, encouragement and strength, and Zubair Khan is the invaluable glue that holds our family together. I dedicate this dissertation to them.

To my family.

Table of Contents

Chapter 1: Introduction

Many uses cases of large-scale analytics today involve processing massive volumes of continuously-produced data. For example, Twitter needs to support functions such as finding patterns in user tweets and counting top trending hashtags over time. Uber must run business-critical calculations such as finding surge prices for different localities when the number of riders exceeds the number of drivers [42]. Zillow needs to provide near-real-time home prices to customers [44] and Netflix needs to find applications communicating in real-time so it can consequently colocate them to improve user experience [33].

The high variability across use cases has led to the creation of many open-source stream processing systems and commercial offerings that have gained massive popularity in a short time. For example, Twitter uses Apache Heron [112], Uber uses Apache Flink [62] and both Netflix and Zillow employ Amazon Kinesis [2], a commercial offering from Amazon. Some corporations have created their in-house stream processing solutions such as Turbine at Facebook [124], and Millwheel [47] and later Dataflow [48] at Google. The popularity of stream processing has increased so much that the streaming analytics market is expected to grow to $35.5 billion by 2024 from $10.3 billion in 2019 [122].

As stream processing systems have gained popularity, they have begun to differentiate themselves in both design and use cases. They began as systems that processed textual data ([7], [45], [47]), but with the proliferation of smart phones and cameras, the processing of video and images has also become an important use case ([99], [6], [172]). The market for processing of video streams is expected to grow to $7.5 billion by 2022, from $3.3 billion in 2017 [123]. Additionally, as storing streams of records became increasingly important, alongside allowing multiple users to consume the same stream, traditional publish-subscribe systems evolved to support some streaming functionality (such as in Kafka Streams [29]). In this thesis, we focus on stream processing applications that process textual data (e.g. sensor readings, and rider and driver locations in ride-share apps etc.) as it arrives.

Once deployed and stable, stream processing jobs continue to process incoming data as long as their developer has use for its results; hence, jobs are usually long-running. Thus, it is essential to minimize the amount of resources they are deployed on, to reduce capital and operational expenses (Capex and Opex).

Users of stream processing jobs generally expect that the jobs will provide well-defined performance guarantees. One of the easiest ways to express these guarantees is in terms of high-level service level objectives (SLOs) [38]. For example, a revenue-critical application for social media websites may be one that constructs an accurate, real-time count of ad-clicks per minute for their advertisers. Similarly, Uber may want to calculate surge pricing or match a driver to a rider within

20 seconds. These are examples of latency sensitive applications. Applications with throughput goals include LinkedIn's pipeline [6], where trillions of events are processed per day, updates are batched together and sent to users according to a user-defined frequency e.g., once per day.

Currently, tuning stream processing jobs to allow them to satisfy their SLOs requires a great deal of manual effort, and expert understanding of how they are deployed. As an example, we find that even today, many Github issues for Apache Heron [112] (a popular stream processing system) are queries for the system's developers that are related to finding out the optimal resource allocations for various scenarios [15–26]. Although there is a great deal of preexisting work on automatically finding the best resource allocation for batch processing jobs [87–89,95,140,157,158], the long-running nature of stream processing jobs means that unlike batch processing jobs, resources made available by finished upstream tasks in a job cannot be reused by downstream tasks. Therefore, the job developer must reason about the entire structure of the job when determining its required resource allocation. In addition, changing input rates of streaming jobs over time imply that a resource allocation may work very well for a job at a certain time of day but may be completely insufficient at another.

Today, developers of the stream processing system deal with these challenges by observing low-level monitoring information such as queue sizes, and CPU and memory load to ascertain which parts of the job are bottlenecked, and scale those out gradually until performance goals are met [27]. This is not a straightforward process – once upstream bottlenecks are resolved, downstream operators may bottleneck. Fully resolving all bottlenecks requires looking at the entire job structure. In addition, this problem becomes more complex as the deployment environment of jobs changes: jobs can be running in a shared, multi-tenant environment, on separate virtual machines on privately-owned infrastructure or on virtual machines on popular cloud offerings such as Amazon EC2 [3] or Microsoft Azure [30].

Our thesis is driven by the vision that *the deployer of each job should be able to clearly specify their performance expectations, or intents, as SLOs to the system, and it is responsibility of the underlying engine and system to meet these objectives.* This alleviates the developer's burden of monitoring and adjusting their job. Modern open-source stream processing systems like Storm [41] are very primitive and do not admit intents of any kind. We posit that new users should not have to grapple with complex metrics such as queue sizes and load.

As stream processing becomes prevalent, it is important to allow jobs to be deployed easily, thus reducing the load on cluster operators that manage the jobs, and reducing the barrier to entry for novice users. These users include a wide range of people from students working in research labs with very limited resources, to experienced software engineers who work alongside the developers of these systems (e.g. for Apache Heron at Twitter), but do not have experience with running the framework themselves and thus have difficulty optimizing jobs. This has lead to the creation of

system-specific trouble-shooting guides [27] that offer advice on how to deploy jobs correctly and tune them to achieve optimal performance. Referring to these guides repeatedly to tune jobs can be an arduous process, especially if the guides are out of date.

Another difficulty in accurately ascertaining the resource requirements of a stream processing job is that some of its components are transparent to the developer. In order to make job development easier, developers are asked to only provide the logical computation the job must perform on each piece of data. The developers are then asked to allocate resources to the job as a whole: thus, they may not allocate sufficient resources for the components that are transparent to them. These include components that perform orchestration e.g., message brokers that communicate messages between operators and monitoring processes that communicate job health metrics to the developer. Usually, developers get around this problem by over-allocating resources to jobs [101] and hope that all goes well. This leads to unnecessarily high operational costs (Opex) [43, 125].

Furthermore, translating a performance goal into a resource specification is made challenging by the number of variables in a data center environment, all of which have an impact on performance. The heterogeneity of available hardware, varying protocol versions used on the network stack, and variation in input rates of jobs due to external, unpredictable events are just a few of these challenges. However, despite all of these variables, all jobs present clear information about the resources they need more of during execution, through bottlenecks. Thus, we propose the central hypothesis of this thesis: *SLO satisfaction in stream processing jobs can be achieved by preventing and mitigating bottlenecks in key components of their deployed structure. These components include those that are transparent to developers and those that are not.*

In order to handle the many variables that can cause variation in performance, previous approaches have applied machine learning (ML) techniques to derive SLO satisfying resource allocations for batch jobs [49, 158]. We argue that correctly identifying bottlenecks in jobs allows us to ascertain the resources they need more of to achieve their SLOs. Each component has a maximum processing rate. Once that is reached, the component is bottlenecked and cannot keep up with a higher input rate, causing input to queue. Hence, bottlenecks need to be resolved to ensure that SLOs are satisfied for all inputs. Additionally, awareness of possible bottlenecks in a job allows us to correctly ascertain the amount of resources the job requires to satisfy its SLO. This allows us to derive SLO-satisfying job deployments in explainable ways, unlike ML based approaches. Our SLO-satisfying job deployments also prevent over-allocation of resources, leading to lower operational costs.

Bottlenecks can be pre-emptively predicted or they can be detected during execution. Pre-emptively forecasting and removing bottlenecks is useful in cases where workloads are very predictable, or have few unexpected events. Usually however, workloads consist of a baseline rate that has a predictable pattern, with some unexpected events. Data center schedulers always require an online reactive component to handle such unexpected loads; however, predictive approaches can

System	Problem Setting	Approach to SLO Satisfaction	Bottlenecks Addressed	Streaming Input
Henge	Multi-Tenant, Limited Resources in DCs	Bottleneck Mitigation	Operators	Text
Caladrius	Unlimited Resources in DCs	Bottleneck Prevention	Operators	Text
Meezan	Commerical Cloud Offerings (e.g., Azure & AWS)	Bottleneck Prevention	Operators & Message Brokers	Text

Figure 1.1: Systems developed in thesis, their respective problem settings and approaches used for deriving effective solutions

be helpful to the extent that they allow us to avoid regularly occurring bottlenecks. Thus, we note that both approaches when utilized together minimize the potential for missed SLOs. In this thesis, we design systems that implement each approach respectively, and can be used to complement each other.

We describe the contributions of this thesis in the next section.

1.1 THESIS CONTRIBUTIONS

The contributions of this thesis (summarized in Figure 1.1) include SLO-satisfying deployments of stream processing jobs that primarily process text-based input, in three different cluster scheduling cases, which cover the majority of deployment use cases within data center environments. These cases are described in detail below:

1. Henge: Intent-driven Multi-Tenant Stream Processing

We built Henge, an online scheduler that provides intent-based multi-tenancy in modern distributed stream processing systems. This means that everyone in an organization can now submit their stream processing jobs to a single, shared, consolidated cluster. Henge allows each job to specify its own performance intent as a Service Level Objective (SLO) that captures latency or throughput SLOs. In such a cluster, the Henge scheduler behaves reactively: it detects bottlenecks during job execution and adapts job configuration continually to mitigate them, so that job SLOs are met in spite of limited cluster resources, and under dynamically varying workloads. SLOs are soft and are based on utility functions. Henge's overall goal is to maximize the total system utility achieved by all jobs deployed on the cluster.

Henge is integrated into Apache Storm. Our experiments with real world workloads show that Henge converges quickly to maximum system utility when the cluster has sufficient resources and to high system utility when resources are constrained.

2. Meezan: Stream Processing as a Service

Meezan is a system that allows novice users to deploy stream processing jobs easily on commercial cloud offerings. Given a user's job, Meezan profiles it to understand its performance at larger scales and with increased input. In light of this information, it presents the user with a spectrum of possible job deployments where the cheapest (/most expensive) options would provide the minimum (/maximum) level of guaranteed throughput performance. With each deployment option, Meezan guarantees that bottlenecks will be prevented as long as the input rate remains constant. Each deployment option ensures that the job has sufficient resources to maintain the promised throughput SLO. This way, Meezan prevents bottlenecks from occurring.

We have integrated Meezan into Apache Heron. Our experiments with real-world clusters and workloads show that Meezan creates job deployments that scale linearly in terms of size and cost in order to scale job throughput and minimizes resource fragmentation. It is able to reduce cost of deployment by up to an order of magnitude, as compared a version of the default Heron scheduler that is modified to support job scheduling for cloud platforms with heterogeneous VM types.

3. Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems

Given the varying job workloads that characterize stream processing, stream processing systems need to be tuned and adjusted to maintain performance targets in the face of variation in incoming traffic. Current auto-scaling systems adopt a series of trials to approach a job's expected performance due to a lack of performance modelling tools. We find that general traffic trends in most jobs lend themselves well to prediction. Based on this premise, we built a system called Caladrius that forecasts the future traffic load of a stream processing job and predicts its processing performance after a proposed change to the parallelism of its operators.

We have integrated Caladrius into Apache Heron. Real world experimental results show that Caladrius is able to estimate a job's throughput performance and CPU load under a given scaling configuration.

Within all three environments, fully understanding the deployment structure of jobs and the different components that can present bottlenecks is essential for creating SLO-satisfying deployments.

1.2 THESIS ORGANIZATION

Chapters 2-3 describe each of our contributions in detail. We conclude with pertinent future directions in chapter 5.

Chapter 2: Henge: Intent-Driven Multi-Tenant Stream Processing

This chapter presents Henge, a system that supports intent-based multi-tenancy in modern stream processing applications. Henge supports multi-tenancy as a first-class citizen: everyone inside an organization can now submit their stream processing jobs to a single, shared, consolidated cluster. Additionally, Henge allows each tenant (job) to specify its own intents (i.e., requirements) as a Service Level Objective (SLO) that captures latency and/or throughput. In a multi-tenant cluster, the Henge scheduler adapts continually to meet jobs' SLOs in spite of limited cluster resources, and under dynamic input workloads. SLOs are soft and are based on utility functions. Henge continually tracks SLO satisfaction, and when jobs miss their SLOs, it wisely navigates the state space to perform resource allocations in real time, maximizing total system utility achieved by all jobs in the system. Henge is integrated in Apache Storm and the thesis presents experimental results, using both production topologies and real datasets.

2.1 INTRODUCTION

Modern stream processing systems process continuously-arriving data streams in real time, ranging from Web data to social network streams. For instance, several companies use Apache Storm [7] (e.g., Weather Channel, Alibaba, Baidu, WebMD, etc.), Twitter uses Heron [112], LinkedIn relies on Samza [6] and others use Apache Flink [4]. These systems provide high-throughput and low-latency processing of streaming data from advertisement pipelines (Yahoo! Inc. uses Storm for this), social network posts (LinkedIn, Twitter), and geospatial data (Twitter), etc.

While stream processing systems for clusters have been around for decades [46, 77], modern stream processing systems have scant support for *intent-based multi-tenancy*. We describe these two terms. First, multi-tenancy allows multiple jobs to share a single consolidated cluster. This capability is lacking in stream processing systems today–as a result, many companies (e.g., Yahoo!) over-provision the stream processing cluster and then physically apportion it among tenants (often based on team priority). Besides higher cost, this entails manual administration of multiple clusters and caps on allocation by the sysadmin, and manual monitoring of job behavior by each deployer.

Multi-tenancy is attractive as it reduces acquisition costs and allows sysadmins to only manage a single consolidated cluster. Thus, this approach reduces capital expenses (Capex) and operational expenses (Opex), lowers total cost of ownership (TCO), increases resource utilization, and allows jobs to elastically scale based on needs. Multi-tenancy has been explored for areas such as key-value stores [147], storage systems [159], batch processing [156], and others [121], yet it remains a vital need in modern stream processing systems.

Second, we believe the deployer of each job should be able to clearly specify their performance

expectations as an intent to the system, and it is the underlying engine's responsibility to meet this intent. This alleviates the developer's burden of monitoring and adjusting their job. Modern open-source stream processing systems like Storm [41] are very primitive and do not admit intents of any kind.

Our approach is to allow each job in a multi-tenant environment to specify its intent as a Service Level Objective (SLO). Then, Henge is responsible for translating these intents to resource configurations that allow the jobs to satisfy their SLOs. The metrics in an SLO should be *user-facing*, i.e., understandable and settable by lay users such as a deployer who is not intimately familiar with the innards of the system. For instance, SLO metrics can capture latency and throughput expectations. SLOs do not include internal metrics like queue lengths or CPU utilization which can vary depending on the software, cluster, and job mix (however, these latter metrics can be monitored and used internally by the scheduler for self-adaptation). It is simpler for lay users to not have to grapple with such complex metrics.

Business	**Use Case**	**SLO Type**
The Weather Channel	Monitoring natural disasters in real-time	Latency e.g., a tuple must be processed within 30 seconds
	Processing collected data for forecasts	Throughput e.g, processing data as fast as it can be read
WebMD	Monitoring blogs to provide real-time updates	Latency e.g., provide updates within 10 mins
	Search Indexing	Throughput e.g., index all new sites at the rate they're found
E-Commerce Websites	Counting ad-clicks	Latency e.g., click count should be updated every second
	Alipay uses Storm to process 6 TB logs per day	Throughput e.g., process logs at the rate of generation

Table 2.1: Stream Processing Use Cases and Possible SLO Types.

While there are myriad ways to specify SLOs (including potentially declarative languages paralleling SQL), this work is best seen as one contributing *mechanisms* that are pivotal to build a truly intent-based distributed system for stream processing. In spite of their simplicity, our latency and throughput SLOs are immediately useful. Time-sensitive jobs (e.g., those related to an ongoing ad campaign) are latency-sensitive and can specify latency SLOs, while longer running jobs (e.g., sentiment analysis of trending topics) typically have throughput SLOs. Table 2.1 shows several real stream processing applications [40], and the latency or throughput SLOs they may require. Support for a multi-tenant cluster with SLOs eliminates the need for over-provisioning. Instead of the de

Schedulers	Job Type	Adaptive	Reservation-Based	SLOs
Mesos [90]	General	✗	✓(CPU, Mem, Disk, Ports)	✗
YARN [156]	General	✗	✓(CPU, Mem, Disk)	✗
Rayon [66]	Batch	✓	✓(Resources across time)	✓
Henge	**Stream**	✓	**✗ (User-facing SLOs)**	✓

Table 2.2: Henge vs. Existing Schedulers.

facto style today of statically partitioning a cluster for jobs, consolidation makes the cluster shared, more effective, and cost-efficient.

As Table 2.2 shows, most existing schedulers use reservation-based approaches to specify intents: besides not being user-facing, these are very hard to estimate even for a job with a static workload [101], let alone the dynamic workloads in streaming applications.

This thesis presents Henge, a system consisting of the first scheduler to support both multi-tenancy and per-job intents (SLOs) for modern stream processing engines. In a cluster of limited resources, Henge continually adapts to meet jobs' SLOs in spite of other competing SLOs, both under natural system fluctuations, and under input rate changes due to diurnal patterns or sudden spikes. As our goal is to satisfy the SLOs of all jobs on the cluster, Henge must deal with the challenge of allocating resources to jobs continually and wisely.

Henge is implemented in Apache Storm, one of the most popular modern open-source stream processing system. Our experimental evaluation uses real-world workloads: Yahoo! production Storm topologies, and Twitter datasets. The evaluation shows that while satisfying SLOs, Henge prevents non-performing topologies from hogging cluster resources. It scales well with cluster size and jobs, and is tolerant to failures.

2.2 CONTRIBUTIONS

This chapter makes the following contributions:

1. We present the design of Henge and its state machine that manages resource allocation on the cluster.

2. We define a new throughput SLO metric called "juice" and present an algorithm to calculate it.

3. We define the structure of SLOs using utility functions.

4. We present implementation details of Henge's integration into Apache Storm.

5. We present evaluation of Henge using production topologies from Yahoo! and real-world workloads e.g., diurnal workloads, spikes in input rate and workloads generated from Twitter traces. We also evaluate Henge with topologies that have different kinds of SLOs e.g., topologies with hybrid SLOs, and tail latency SLOs. Additionally, we measure Henge's fault-tolerance, and scalability with respect to both an increasing number of topologies and cluster size.

This chapter is organized as follows.

- Section 2.3 presents a summary of Henge goals and design.
- Section 2.5 discusses core Henge design: SLOs and utilities (Section 2.5.1), operator congestion (Section 2.5.2), and the state machine (Section 2.6).
- Section 2.7 describes juice and its calculation.
- Section 2.8 describes how Henge is implemented and works as a module in Apache Storm.
- Section 2.9 presents evaluation results.
- Section 2.10 discusses related work in the areas of elastic stream processing systems, cluster scheduling and SLAs/SLOs in other areas.

2.3 HENGE SUMMARY

This section briefly describes the key ideas behind Henge's goals and design.

Juice: As input rates can vary over time, it is infeasible for a throughput SLO to merely specify a desired absolute output rate value. For instance, it is very common for stream processing jobs to have diurnal workloads, with higher input rates during the day when most users are awake. In addition, sharp spikes in workloads can also occur e.g., when a lot of tweets are generated discussing an interesting event on Twitter. Therefore, setting an absolute value as a throughput SLO is very difficult: should it be set according to the highest possible workload or the average workload? We resolve this dilemma by defining a new *input rate-independent* metric for throughput SLOs called *juice*. In addition to being independent of input rate, juice is also independent of the operations a topology performs and the structure of the topology. We show how Henge calculates juice for arbitrary topologies (section 2.7). This makes the task of setting a throughput SLO very simple for the users.

Juice lies in the interval [0, 1] and captures the ratio of processing rate to input rate–a value of 1.0 is ideal and implies that the rate of incoming tuples equals rate of tuples being processed by the job.

Throughput SLOs can then contain a minimum threshold for juice, making the SLO independent of input rate. We consider processing rate instead of output rate as this generalizes to cases where tuples may be filtered (thus they affect results but are never outputted themselves).

SLOs: A job's SLO can capture either latency or juice (or a combination of both). The SLO contains: a) a threshold (min-juice or max-latency), and b) a *utility function*, inspired by soft real-time systems [110]. The utility function maps current achieved performance (latency or juice) to a value which represents the benefit to the job, even if it does not meet its SLO threshold. The function thus captures the developer intent that a job attains full "utility" if its SLO threshold is met and partial benefit if not. Henge supports monotonic utility functions: the closer the job is to its SLO threshold, the higher its achieved maximum possible utility. (Section 2.5.1).

State Space Exploration: At its core, Henge decides wisely how to change resource allocations of jobs (or rather of their basic units, operators) using a new *state machine* approach (Section 2.6). Moving resources in a live cluster is challenging. It entails a state space exploration where every step has both: 1) a significant realization cost, because moving resources takes time and affects jobs, and 2) a convergence cost, since the system needs a while to converge to steady state after a step. Our state machine is unique as it is *online* in nature: it takes one step at a time, evaluates its effect, and then moves on. This is a good match for unpredictable and dynamic scenarios such as modern stream processing clusters.

The primary actions in our state machine are: 1) Reconfiguration (give resources to jobs missing SLO), 2) Reduction (take resources away from overprovisioned jobs satisfying SLO), and 3) Reversion (give up an exploration path and revert to past high utility configuration). Henge takes these actions wisely. Jobs are given more resources as a function of the amount of congestion they face. Highly intrusive actions like reduction are minimized in number and frequency. Small marginal gains in a job's utility lead to it being precluded from reconfigurations in the near future.

Maximizing System Utility: Design decisions in Henge are aimed at converging each job quickly to its maximum achievable utility in a minimal number of steps. Henge attempts to maximize total achieved utility summed across all jobs. It does so by finding SLO-missing topologies, then their congested operators, and gives the operators thread resources according to their congestion levels. Our approach creates a weak form of Pareto efficiency [161]; in a system where jobs compete for resources, Henge transfers resources among jobs only if this will cause the system's utility to rise.

Henge's technique attempts to satisfy all jobs' SLOs when the cluster is relatively less packed. When the cluster becomes packed, jobs with higher priority automatically get more resources as this leads to higher maximum utility for the organization.

Preventing Resource Hogging: Topologies with stringent SLOs may try to take over all the resources of the cluster. To mitigate this, Henge prefers giving resources to those topologies that: a) are farthest from their SLOs, and b) continue to show utility improvements due to recent Henge actions. This spreads resource allocation across all wanting jobs and prevents starvation and resource hogging.

2.4 BACKGROUND

This section presents relevant background information about stream processing topologies, particularly those belonging to Apache Storm [7].

A stream processing job can be logically interpreted as a *topology*, i.e., a directed acyclic graph of *operators* (the Storm term is “bolt"). We use the terms job and topology interchangeably in this thesis. An operator is a logical processing unit that applies user-defined functions on a stream of *tuples*. The edges between operators represent the dataflow between the computation components. Source operators (called spouts) pull input tuples while sink operators spew output tuples. Spouts typically pull in tuples from sources such as publish-subscribe systems e.g., Kafka. The sum of output rates of sinks in a topology is its *output rate*, while the sum of all spout rates is the *input rate*. Each operator is parallelized via multiple *tasks*. Fig. 2.1 shows a topology with one spout and one sink.

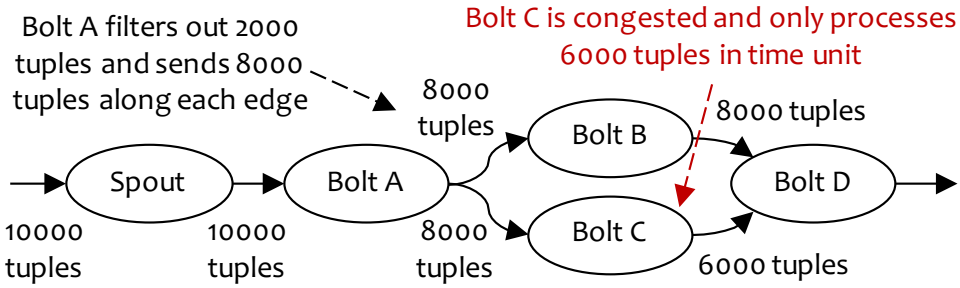


Figure 2.1: Sample Storm topology. Showing tuples processed per unit time. Edge labels indicate number of tuples sent out by the parent operator to the child. (Congestion described in section 2.7.)

We consider long-running stream processing topologies with a continuous operator model. Storm topologies usually runs on a distributed cluster. Users can submit their jobs to the master process which is called Nimbus. This process distributes and coordinates the execution of the topology. A topology is actually run on one or more worker processes. A worker node may have one or more worker processes but each each worker is mapped only to a single topology. Each worker runs on a JVM and instantiates *executors* (threads), which run tasks specific to one operator. Therefore, tasks provide parallelism for bolts and spouts and executors provide parallelism for the whole

topology. An operator processes streaming data one tuple at a time and forwards the tuples to the next operators in the topology. Systems that follow such a model include Apache Storm [154], Heron [112], Flink [4] and Samza [6].

Each worker node also runs a Supervisor process that communicates with Nimbus. A separate Zookeeper [8] installation is used to maintain the state of the cluster. Figure 2.2 shows the high-level architecture of Apache Storm.

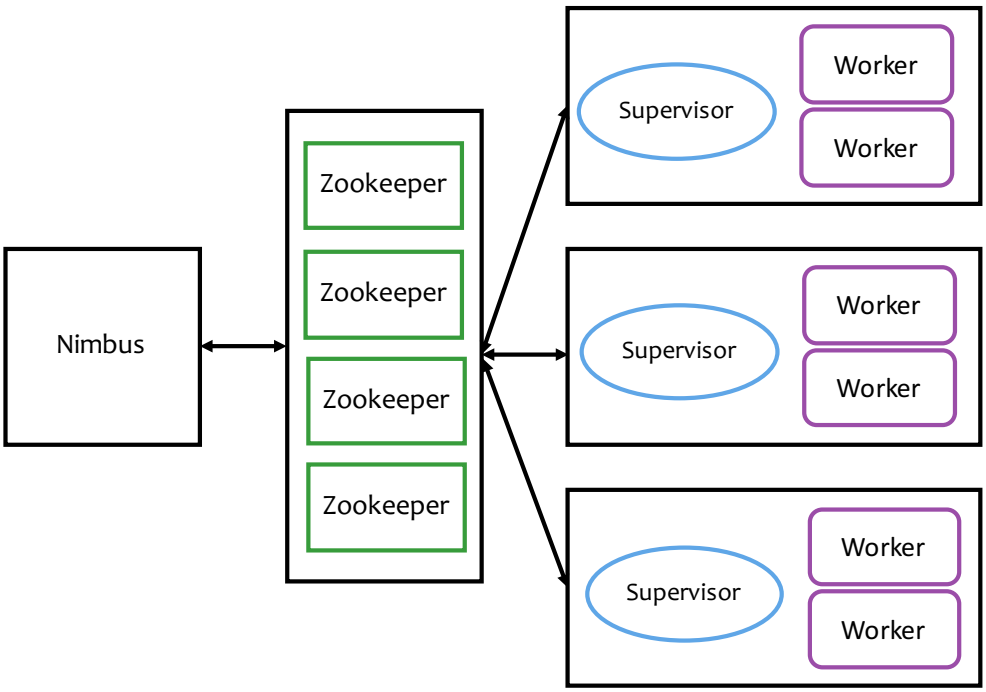


Figure 2.2: High Level Storm Architecture

Definitions: The latency of a tuple is the time between it entering the topology from the source, to producing an output result on any sink. A *topology's latency* is then the average of tuple latencies, measured over a period of time. A *topology's throughput* is the number of tuples it processes per unit time.

A *Service Level Objective (SLO)* [38] is a customer topology's requirement, in terms of latency and/or throughput. Some examples of latency-sensitive jobs include applications that perform real-time analytics or real-time natural language processing, provide moderation services for chat rooms, count bid requests, or calculate real-time trade quantities in stock markets. Examples of throughput-sensitive application include jobs that perform incremental checkpointing, count online visitors, or perform sentiment analysis.

2.5 SYSTEM DESIGN

This section describes Henge's utility functions (Section 2.5.1), congestion metric (Section 2.5.2), and its state machine (Section 2.6).

2.5.1 SLOs and Utility Functions

Each topology's SLO contains: a) an SLO threshold (min-juice or max-latency), and b) a utility function. The utility function maps the current performance metrics of the job (i.e. its SLO metric) to a *current* utility value. This approach abstracts away the type of SLO metric each topology has, and allows the scheduler to compare utilities across jobs.

Currently, Henge supports both latency and throughput metrics in the SLO. Latency was defined in Section 2.4.

For throughput, we use a new SLO metric called juice which we define concretely later in Section 2.7 (for the current section, an abstract throughput metric suffices).

When the SLO threshold cannot be satisfied, the job still desires *some* level of performance close to the threshold. Hence, utility functions must be monotonic–for a job with a latency SLO, the utility function must be monotonically non-increasing as latency rises, while for a job with a throughput SLO, it must be monotonically non-decreasing as throughput rises.

Each utility function has a *maximum utility* value, achieved only when the SLO threshold is met e.g., a job with an SLO threshold of 100 ms would achieve its maximum utility only if its current latency is below 100 ms. As latency grows above 100 ms, utility can fall or plateau but can never rise.

The maximum utility value is based on job priority. For example, in Fig. 2.3a, topology T2 has twice the priority of T1, and thus has twice the maximum utility (20 vs. 10).

Given these requirments, Henge is able to allow a wide variety of shapes for its utility functions including: linear, piece-wise linear, step function (allowed because utilities are monotonically non-increasing instead of monotonically decreasing), lognormal, etc. Utility functions do not need to be continuous. All in all, this offers users flexibility in shaping utility functions according to individual needs.

The concrete utility functions used in our Henge implementation are *knee* functions, depicted in Fig. 2.3. A knee function has two pieces: a plateau beyond the SLO threshold, and a sub-SLO part for when the job does not meet the threshold. Concretely, the achieved utility for jobs with throughput and latency SLOs respectively, are:

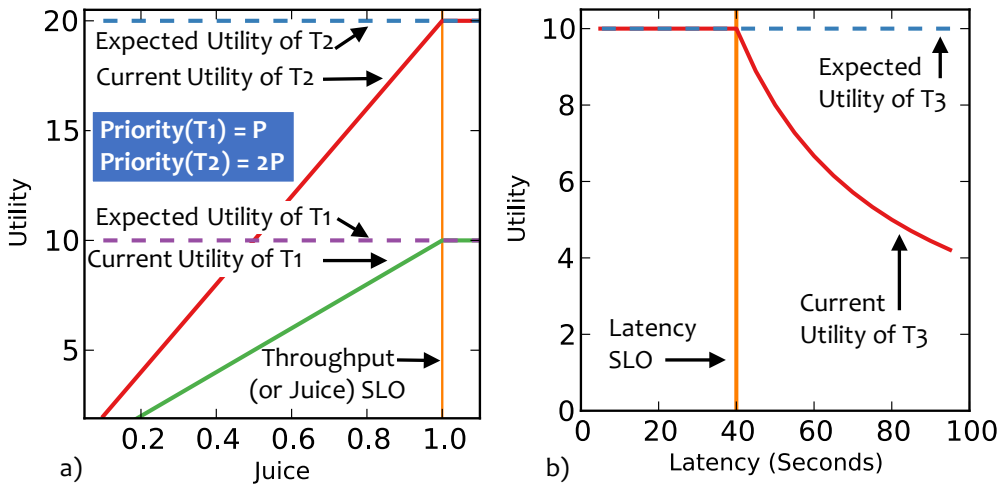


Figure 2.3: Knee Utility functions. (a) Throughput SLO utility, (b) Latency SLO utility.

$$\frac{\texttt{Current Utility}}{Job\ Max\ Utility} = min(1, \frac{Current\ Throughput\ Metric}{SLO\ Throughput\ Threshold}) \tag{2.1}$$

$$\frac{\texttt{Current Utility}}{Job\ Max\ Utility} = min(1, \frac{SLO\ Latency\ Threshold}{Current\ Latency}) \tag{2.2}$$

The sub-SLO is the last term inside “min".

For throughput SLOs, the sub-SLO is linear and arises from the origin point. For latency SLOs, the sub-SLO is hyperbolic ($y \propto \frac{1}{x}$), allowing increasingly smaller utilities as latencies rise. Fig. 2.3 shows a throughput SLO (Fig. 2.3a) vs. latency SLO (Fig. 2.3b).

We envision Henge to be used internally inside companies, hence job priorities are set in a consensual way (e.g., by upper management). The utility function approach is also amenable to use in contracts like Service Level Agreements (SLAs), however these are beyond the scope of this chapter.

2.5.2 Operator Congestion Metric

A topology misses its SLOs when some of its operators become *congested*, i.e., have insufficient resources. To detect congestion our implementation uses a metric called operator *capacity* [39]. However, Henge can also use other existing congestion metrics, e.g., input queue sizes or ETP [167].

Operator capacity captures the fraction of time that an operator spends processing tuples during a

time unit. Its values lie in the range $[0.0, 1.0]$. If an executor's capacity is near 1.0, then it is close to being congested.

Consider an executor E that runs several (parallel) tasks of a topology operator. Its capacity is calculated as:

$$Capacity_E = \frac{Executed\,Tuples_E \times Execute\,Latency_E}{Unit\,Time} \tag{2.3}$$

where $Unit\,Time$ is a time window. The numerator multiplies the number of tuples executed in this window and their average execution latency to calculate the total time spent in executing those tuples. The operator capacity is then the maximum capacity across all executors containing it.

Henge considers an operator to be congested if its capacity is above the threshold of 0.3. This increases the pool of possibilities, as more operators become candidates for receiving resources (described next).

2.6 HENGE STATE MACHINE

The state machine (shown in Fig. 2.4) considers all jobs in the cluster as a whole and wisely decides how many resources to give to congested jobs in the cluster and when to stop. The state machine is for the entire cluster, not per job.

The cluster is in the Converged state if and only if either: a) all topologies have reached their maximum utility (i.e., satisfy their respective SLO thresholds), or b) Henge recognizes that no further actions will improve the performance of any topology, and thus it has reverted to the last best configuration. All other states are Not Converged.

To move among these two states, Henge uses three actions: Reconfiguration, Reduction, and Reversion.

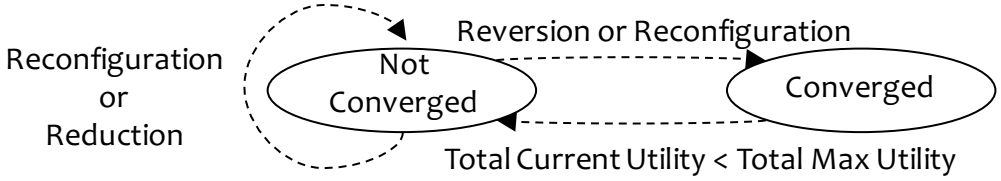


Figure 2.4: Henge's State Machine for the Cluster.

2.6.1 Reconfiguration

In the Not Converged state, a Reconfiguration gives resources to topologies missing their SLO. Reconfigurations occur in *rounds* which are periodic intervals (currently 10 s apart). In each round, Henge first sorts all topologies missing their SLOs, in descending order of their maximum utility, with ties broken by preferring lower current utility. It then picks the head of this sorted queue to allocate resources to.

This greedy strategy works best to maximize cluster utility.

Within this selected topology, the intuition is to increase each congested operator's resources by an amount proportional to its respective congestion. Henge uses the capacity metric (Section 2.5.2, eq. 2.3) to discover all congested operators in this chosen topology, i.e., operator capacity > 0.3. It allocates each congested operator an extra number of threads based on the following equation:

$$\left(\frac{Current\ Operator\ Capacity}{Capacity\ Threshold} - 1\right) \times 10 \tag{2.4}$$

Henge deploys this configuration change to a single topology on the cluster, and waits for the measured utilities to quiesce (this typically takes a minute or so in our configurations). No further actions are taken in the interim. It then measures the total cluster utility again, and if it improved, Henge continues its operations in further rounds, in the Not Converged State. If this total utility reaches the maximum value (the sum of maximum utilities of all topologies), then Henge cautiously continues monitoring the recently configured topologies for a while (4 subsequent rounds in our setting). If they all stabilize, Henge moves the cluster to the Converged state.

A topology may improve only marginally after being given more resources in a reconfiguration, e.g., utility increases $< 5\%$. In such a case, Henge retains the reconfiguration change but skips this particular topology in the near future rounds. This is because the topology may have plateaued in terms of marginal benefit from getting more threads. Since the cluster is dynamic, this black-listing of a topology is not permanent but is allowed to expire after a while (1 hour in our settings), after which the topology will again be a candidate for reconfiguration.

As reconfigurations are exploratory steps in the state space search, total system utility may decrease after a step. Henge employs two actions called Reduction and Reversion to handle such cases.

2.6.2 Reduction

If a Reconfiguration causes total system utility to drop, the next action is either a Reduction or a Reversion. Henge performs Reduction if and only if three conditions are true: (a) the cluster is

congested (we detail below what this means), (b) there is at least one SLO-satisfying topology, and (c) there is no past history of a Reduction action.

First, CPU load is defined as the number of processes that are running or runnable on a machine [10]. A machine's load should be $\leq$ number of available cores, ensuring maximum utilization and no over-subscription. As a result, Henge considers a machine congested if its CPU load exceeds its number of cores. Henge considers a *cluster congested* when it has a majority of its machines congested.

If a Reconfiguration drops utility and results in a congested cluster, Henge executes Reduction to reduce congestion. For all topologies *meeting* their SLOs, it finds all their un-congested operators (except spouts) and reduces their parallelism level by a large amount (80% in our settings). If this results in SLO misses, such topologies will be considered in future reconfiguration rounds. To minimize intrusion, Henge limits Reduction to once per topology; this is reset if external factors change (input rate, set of jobs, etc.). Akin to backoff mechanisms [92], massive reduction is the only way to free up a lot of resources at once, so that future reconfigurations may have a positive effect. Reducing threads also decreases their context switching overhead.

Right after a reduction, if the next reconfiguration drops cluster utility again while keeping the cluster congested (measured using CPU load), Henge recognizes that performing another reduction would be futile. This is a typical "lockout" case, and Henge resolves it by performing Reversion.

2.6.3 Reversion

If a Reconfiguration drops utility and a Reduction is not possible (meaning that at least one of the conditions (a)-(c) in Section 2.6.2 is not true), Henge performs Reversion.

Henge sorts through its history of Reconfigurations and picks the one that maximized system utility. It moves the system back to this past configuration by resetting the resource allocations of all jobs to values in this past configuration and moves to the Converged state. Here, Henge essentially concludes that it is impossible to further optimize cluster utility, given this workload. Henge maintains this configuration until changes like further SLO violations occur, which necessitate reconfigurations.

If a large enough drop ($> 5\%$) in utility occurs in this Converged state (e.g., due to new jobs, or input rate changes), Henge infers that as reconfigurations cannot be a cause of this drop, the workload of topologies must have changed. As all past actions no longer apply to this change in behavior, Henge forgets all history of past actions and moves to the Not Converged state. This means that in future reversions, forgotten states will not be available. This reset allows Henge to start its state space search afresh.

2.6.4 Discussion

Online vs. Offline State Space Search: Henge prefers an online state space search. In fact, our early attempt at designing Henge was to perform offline state space exploration (e.g., through simulated annealing), by measuring SLO metrics (latency, throughput) and using analytical models to predict their relation to resources allocated to the job.

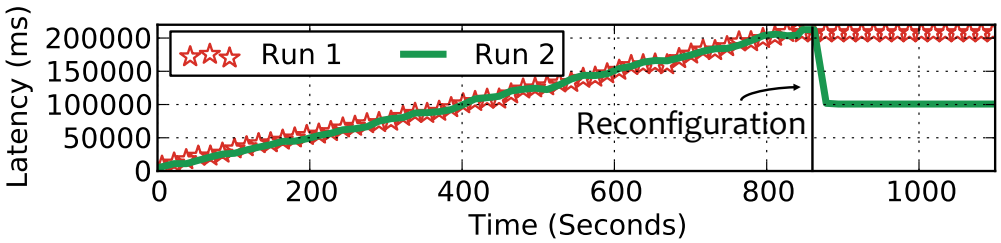


Figure 2.5: Unpredictability in Modern Stream Processing Engines: Two runs of the same topology (on 10 machines) being given the same extra computational resources (28 threads, i.e., executors) at 910 s, react differently.

The offline approach turned out to be impractical. Analysis and prediction is complex and often turns out to be inaccurate for stream processing systems, which are very dynamic in nature. (This phenomenon has also been observed in other distributed scheduling domains, e.g., see [59,101,133].) We show an example in Fig. 2.5. The figure shows two runs of the same Storm job on 10 machines. In both runs we gave the job equal additional thread resources (28 threads) at t=910 s. Latency drops to a lower value in run 2, but only stabilizes in run 1.

This is due to differing CPU resource consumptions across the runs. More generally, we find that natural fluctuations occur commonly in an application's throughput and latency; left to itself an application's performance changes and degrades gradually over time. We observed this for all our actions: reconfiguration, reduction, and reversion. Thus, we concluded that online state space exploration would be more practical.

Statefulness, Memory Bottlenecks: The common case among topologies is stateless operators that are CPU-bound, and our exposition so far is thus focused. Nevertheless, Henge gracefully handles stateful operators and memory-pressured nodes (evaluated in Sections 2.9.3, 2.9.5).

Dynamically Added Jobs: Henge accepts new jobs deployed on the multi-tenant cluster as long as existing jobs are able to satisfy their SLOs. However, if the set of jobs the cluster starts with do not satisfy their SLOs, Henge employs admission control. A possible future direction may be that if existing jobs on the cluster do not all satisfy their SLOs, Henge selects the maximum subset of

jobs from the existing and newly added jobs that can be deployed on the cluster while providing maximum utility.

2.7 JUICE: DEFINITION AND ALGORITHM

This section describes the motivation for Juice and how it is calculated.

As described in section 2.1, a throughput metric (for use in throughput SLOs) should be designed in a way that is independent of input rate. Henge uses a new metric called *juice*. Juice defines what fraction of the input data is being processed by the topology per unit time. It lies in the interval [0, 1], and a value of 1.0 means all the input data that arrived in the last time unit has been processed. Thus, the user can set throughput requirements as a percentage of the input rate (Section 2.5.1), and Henge subsequently attempts to maintain this even as input rates change.

Any algorithm that calculates juice should satisfy three requirements:
1. Code Independence: It should be independent of the operators' code, and should be calculate-able by only considering the number of tuples generated by operators.
2. Rate Independence: It should abstract away throughput SLO requirements in a way that is independent of absolute input rate.
3. Topology Independence: It should be independent of the shape and structure of the topology.

Juice Intuition: Overall, juice is formulated to reflect the *global* processing efficiency of a topology. We define per-operator contribution to juice as the proportion of input passed in originally *from the source* that the operator processed in a given time window. This reflects the impact of that operator *and* its upstream operators, on this input. The juice of a topology is then the normalized sum of juice values of all its sinks.

Juice Calculation: Henge calculates juice in configurable windows of time (unit time). We define *source input* as the tuples that arrive at the input operator in a unit of time. For each operator o in a topology that has n parents, we define T_o^i as the total number of tuples sent out from its i^{th} parent per time unit, and E_o^i as the number of tuples that operator o executed (per time unit), from those received from parent i.

The per-operator contribution to juice, J_o^s, is the proportion of source input *sent* from source s that operator o received and processed. Given that J_i^s is the juice of o's i^{th} parent, then J_o^s is:

$$J_o^s = \sum_{i=1}^{n} \left(J_i^s \times \frac{E_o^i}{T_o^i} \right) \tag{2.5}$$

A spout s has no parents, and its juice: $J_s = \frac{E_s}{T_s} = 1.0$.

In eq. 2.5, the fraction $\frac{E_o^i}{T_o^i}$ reflects the proportion of tuples an operator received from its parents, and processed successfully. If no tuples waiting in queues, this fraction is equal to 1.0. By multiplying this value with the parent's juice we accumulate through the topology the effect of all upstream operators.

We make two important observations. In the term $\frac{E_o^i}{T_o^i}$, it is critical to take the denominator as the number of tuples *sent* by a parent rather than received at the operator. This allows juice: a) to account for data splitting at the parent (fork in the DAG), and b) to be reduced by tuples dropped by the network. The numerator is the number of *processed* tuples rather than the number of output tuples – this allows juice to generalize to operator types whose processing may drop tuples (e.g., filter).

Given all operator juice values, a topology's juice can be calculated by normalizing w.r.t. number of sources:

$$\frac{\sum_{Sinks\ s_i,\ Sources\ s_j} (J_{s_i}^{s_j})}{Total\ Number\ of\ Sources} \tag{2.6}$$

If no tuples are lost in the system, the numerator sum is equal to the number of sources. To ensure that juice stays below 1.0, we normalize the sum with the number of sources.

Example 2.1: Consider Fig. 2.1 in Section 2.4. $J_A^s = 1 \times \frac{10K}{10K} = 1$ and $J_B^s = J_A^s \times \frac{8K}{16K} = 0.5$. B has a T_B^A of 16K and not 8K, since B only receives half the tuples that were sent out by operator A, and its per-operator juice should be in context of only this half (and not all source input).

The value of $J_B^s = 0.5$ indicates that B processed only half the tuples sent out by parent A. This occurred as the parent's output was split among children. (If (alternately) B and C were sinks (if D were absent from the topology), then their juice values would sum up to the topology's juice.). D has two parents: B and C. C is only able to process 6K as it is congested. Thus, $J_C^s = J_A^s \times \frac{6K}{16K} = 0.375$. T_D^C thus becomes 6K. Hence, $J_D^C = 0.375 \times \frac{6K}{6K} = 0.375$. J_D^B is simply $0.5 \times \frac{8K}{8K} = 0.5$. We sum the two and obtain $J_D^s = 0.375 + 0.5 = 0.875$. It is less than 1.0 as C was unable to process all tuples due to congestion.

Example 2.2 (Topology Juice with Split and Merge):

In Fig. 2.6, we show how our approach generalizes to: a) multiple sources (spout 1 & 2), and b) operators splitting output (E to B and F) and c) operators with multiple input streams (A and E to B). Bolt A has a juice value of 0.5 as it can only process half the tuples spout 1 sent it. Bolt D has a

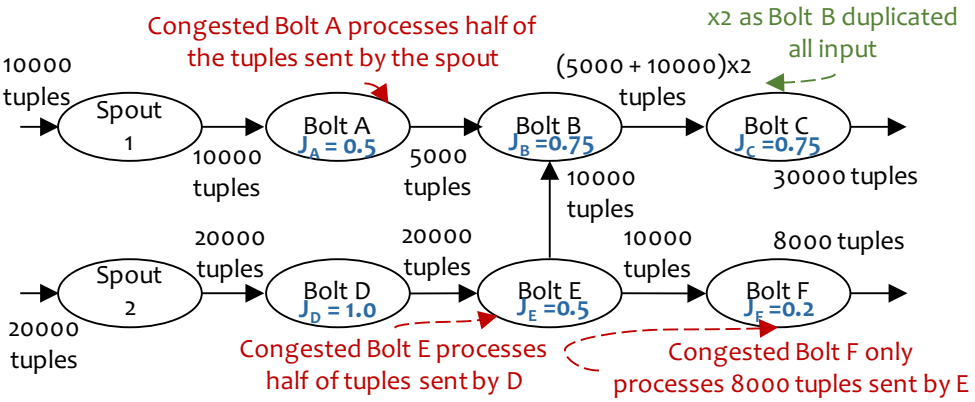


Figure 2.6: Juice Calculation in a Split and Merge Topology.

juice value of 1.0. 50% of the tuples from D to E are unprocessed due to congestion at E. E passes its tuples on to B and F: both of them get half of the total tuples it sends out. Therefore, B has juice of 0.25 from E and 0.5 from A ($0.25 + 0.5 = 0.75$). 20% of the tuples E sent F are unprocessed at F as it is congested, so F has a juice value of $0.25 \times 0.8 = 0.2$. C processes as many tuples as B sent it, so it has the same juice as B (0.75). The juice of the topology is the sum of the juices of the two sinks, normalized by the number of sources. Thus, the topology's juice is $\frac{0.2+0.75}{2} = 0.475$.

Some Observations: First, while our description used unit time, our implementation calculates juice using a sliding window of 1 minute, collecting data in sub-windows of length 10 s. This needs only loose time synchronization across nodes (which may cause juice values to momentarily exceed 1, but does not affect our logic). Second, eq. 2.6 treats all processed tuples equally–instead, a weighted sum could be used to capture the higher importance of some sinks (e.g., sinks feeding into a dashboard). Third, processing guarantees (exactly, at least, at most once) are orthogonal to the juice metric.

Our experiments use the non-acked version of Storm (at most once semantics), but Henge also works with the acked version of Storm (at least once semantics). At least once semantics entails that if any tuples fail (i.e., they have to be reprocessed), we should proportionally reduce juice to only reflect the amount of tuples acked. We do so in the following manner:

$$J_{Final} = J_o^s \times \frac{Total\ No.\ of\ Tuples\ Acked}{Total\ No.\ of\ Tuples\ Sent\ by\ All\ Spouts} \tag{2.7}$$

This allows the juice metric to reflect only the tuples that have been processed and provide value to the final result.

2.8 IMPLEMENTATION

This section describes how Henge is built and is integrated into Apache Storm [7].

Henge involves 3800 lines of Java code. It is an implementation of the predefined IScheduler interface. The scheduler runs as part of the Storm Nimbus daemon, and is invoked by Nimbus periodically every 10 seconds. The developer can specify which scheduler to use in a configuration file that is provided to Nimbus. Further changes were made to Storm Config, allowing users to set topology SLOs and utility functions while writing topologies.

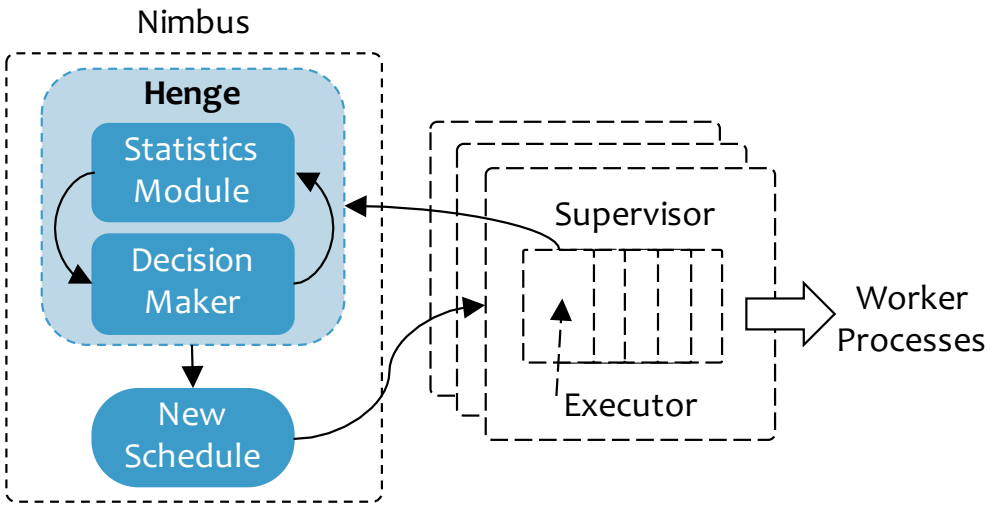


Figure 2.7: Henge Implementation: *Architecture in Apache Storm.*

Henge's architecture is shown in Fig. 2.7. The Decision Maker implements the Henge state machine of Section 2.6. The Statistics Module continuously calculates cluster and per-topology metrics such as the number of tuples processed by each task of an operator per topology, the end-to-end latency of tuples, and the CPU load per node. This information is used to produce useful metrics such as juice and utility, which are passed to the Decision Maker. The Decision Maker runs the state machine, and sends commands to Nimbus to implement actions.

The Statistics Module also tracks historical performance and configuration of topologies whenever a reconfiguration is performed by the Decision Maker, so that reversion can be performed.

2.9 EVALUATION

This section presents the evaluation of Henge with a variety of workloads, topologies, and SLOs. We answer the following necessary experimental questions:

1. Is juice truly independent of input rate, topology structure and operations? i.e., Is juice a good abstraction for throughput?

2. How well does Henge perform as compared to finely-tuned manual configurations and vanilla Storm?

3. Is Henge able to maximize cluster utility for a variety of workloads e.g., diurnal workloads, spikes in input rate and production workloads?

4. Is Henge scalable and fault-tolerant?

Experimental Setup: By default, our experiments used the Emulab cluster [160], with machines (2.4 GHz, 12 GB RAM) running Ubuntu 12.04 LTS, connected via a 1 Gbps connection. Another machine runs Zookeeper [8] and Nimbus. Workers (Java processes running executors) are allotted to each of our 10 machines (we evaluate scalability later).

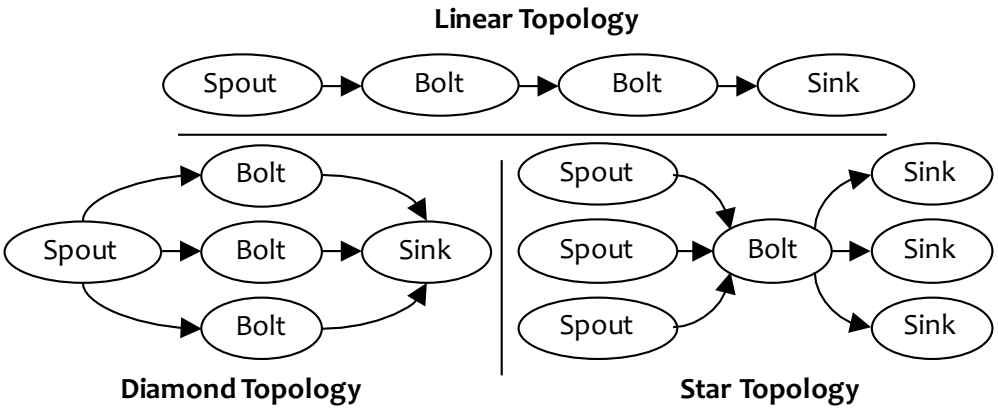


Figure 2.8: Three Microbenchmark Topologies.

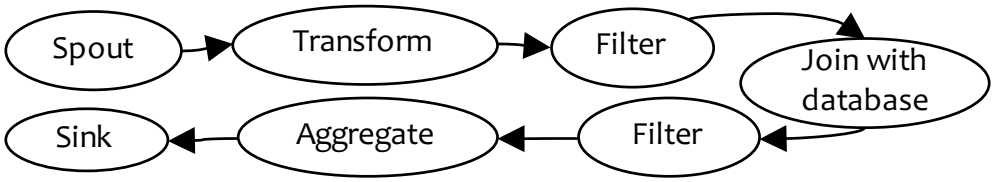


Figure 2.9: PageLoad Topology from Yahoo!.

Topologies: For evaluation, we use both: a) micro-topologies that are possible sub-parts of larger topologies [167], shown in Fig. 2.8; and b) a production topology from Yahoo! Inc. [167]–this topology is called “PageLoad" (Fig. 2.9). Operators are the ones that are most commonly used in production: filtering, transformation, and aggregation. In each experimental run, we initially

allow topologies to run for 900 s without interference (to stabilize and to observe their performance with vanilla Storm), and then enable Henge to take actions. All topology SLOs use the knee utility function of Section 2.5.1. Hence, below we use "SLO" as a shorthand for the SLO threshold.

2.9.1 Juice as a Performance Indicator

Juice is an indicator of queue size: Fig. 2.10 shows the inverse correlation between topology juice and queue size at the most congested operator of a PageLoad topology. Queues buffer incoming data for operator executors, and longer queues imply slower execution rate and higher latencies. Initially queue lengths are high and erratic–juice captures this by staying well below 1. At the reconfiguration point (910 s) the operator is given more executors, and juice converges to 1 as queue lengths fall, stabilizing by 1000 s.

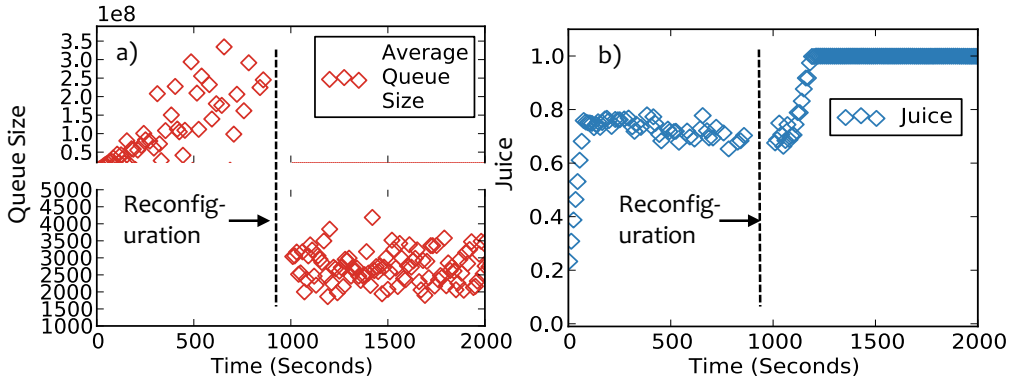


Figure 2.10: Juice vs. Queue Size: Inverse Relationship.

Juice is independent of operations and input rate: In Fig. 2.11, we run 5 PageLoad topologies on one cluster, and show data for one of them. Initially juice stabilizes to around 1.0, near t=1000 s (values above 1 are due to synchronization errors, but they don't affect our logic). PageLoad filters tuples, thus output rate is $<$ input rate–however, juice is 1.0 as it shows that all input tuples are being processed.

Then at 4000 s, we triple the input rate to all tenant topologies. Notice that juice stays 1.0. Due to natural fluctuations, at 4338 s, PageLoad's juice drops to 0.992. This triggers reconfigurations (vertical lines) from Henge, stabilizing the system by 5734 s, maximizing cluster utility.

2.9.2 Henge Policy and Scheduling

Impact of Initial Configuration:

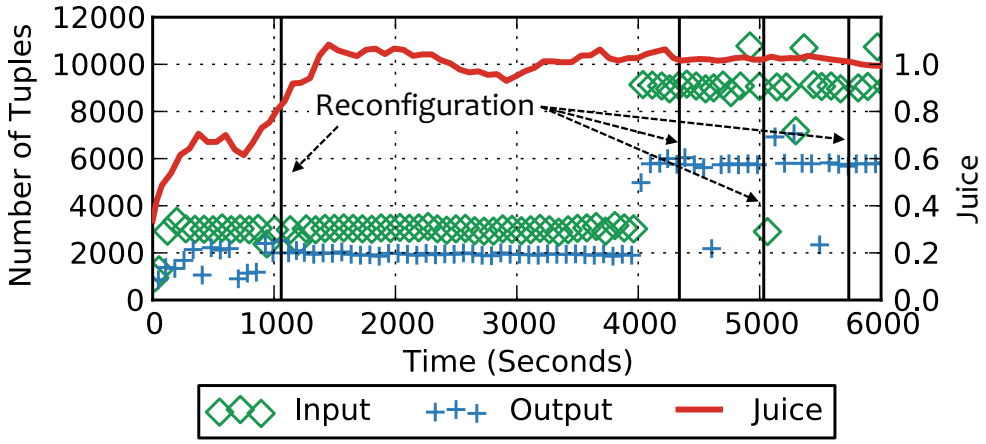


Figure 2.11: Juice is Rate-Independent: Input rate is increased by 3 × at 4000 s, but juice does not change. When juice falls to 0.992 at 4338 s, Henge stabilizes it to 1.0 by 5734 s.

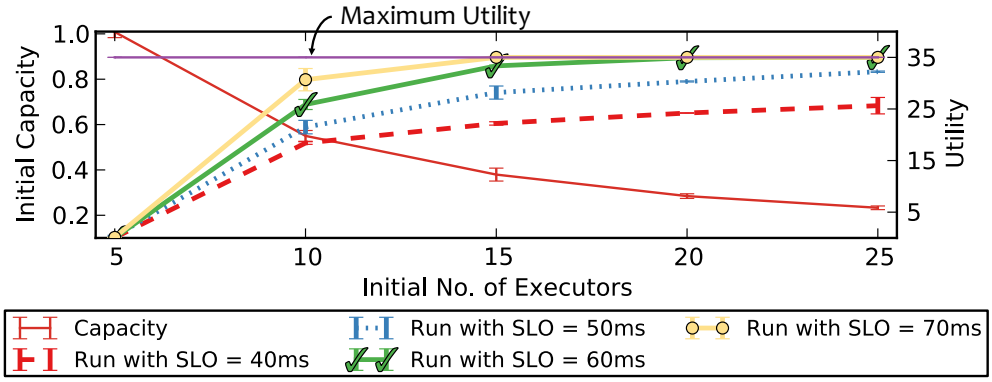


Figure 2.12: Performance vs. Resources in Apache Storm: The x-axis shows initial parallelism of one intermediate operator in a linear topology. Left y-axis shows initial capacity of the operator. Right y-axis shows stable utility reached without using Henge.

State Space: Fig. 2.12 illustrates the state space that Henge needs to navigate. These are runs *without* involving Henge. We vary the initial number of executors for an intermediate operator. Fewer initial executors (5, 10) lead to a high capacity (indicating congestion: Section 2.5.2) and consequently the topology is unable to achieve its SLO. From the plot, the more stringent the SLO, the greater the number of executors needed to reach max utility. Except very stringent jobs SLOs (40, 50 ms) all others can meet their SLO.

Henge In Action: Now, we put Henge into action on Fig. 2.12's topology and initial state, with max utility 35. Fig. 2.13 shows the effect of varying: a) initial number of executors (5 to 25), b) latency SLO (40 ms to 60 ms), and c) input rate. We plot converged utility, rounds needed, and executors assigned.

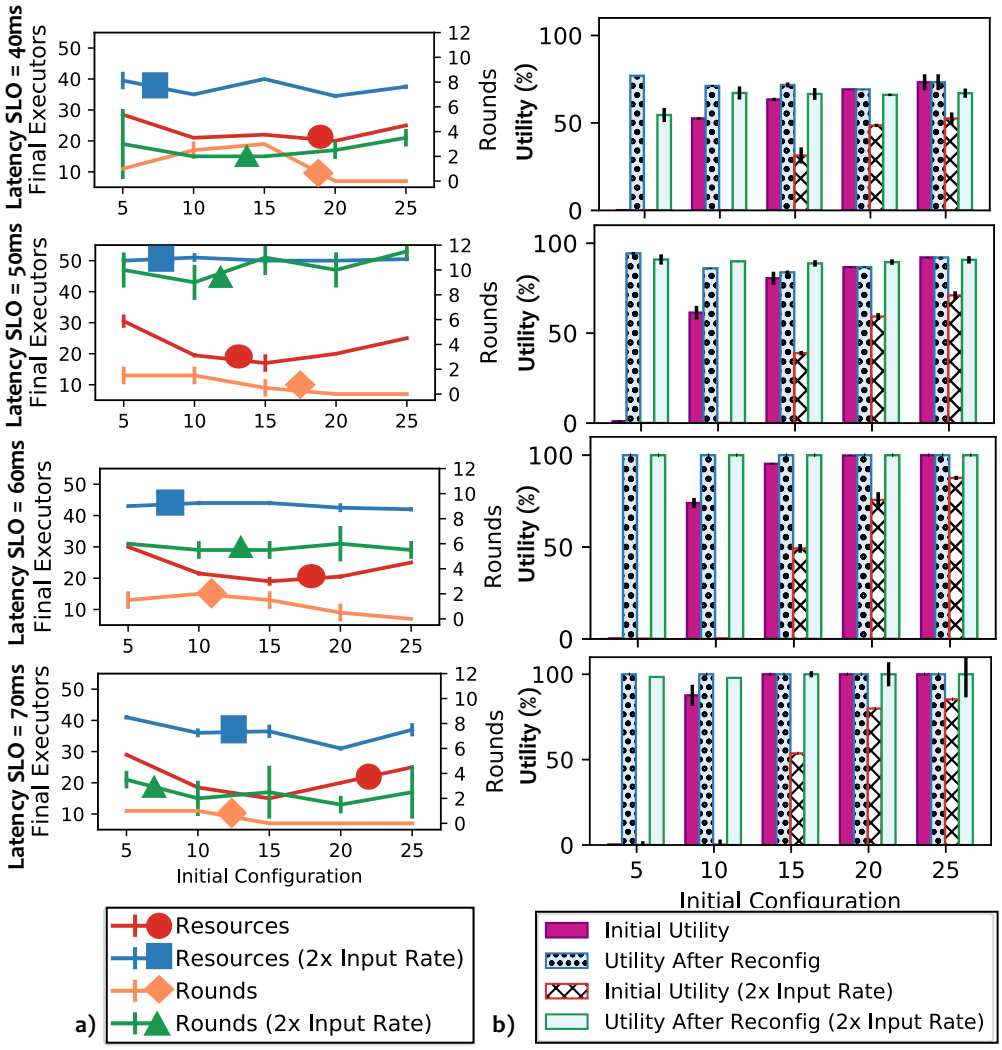


Figure 2.13: Effect of Henge on Figure 2.12's initial configurations: SLOs become more stringent from bottom to top. We also explore a 2 × higher input rate. a) Left y-axis shows final parallelism level Henge assigned to each operator. Right y-axis shows number of rounds required to reach said parallelism level. b) Utility values achieved before and after Henge.

We observe that generally, Henge gives more resources to topologies with more stringent SLOs and higher input rates. For instance, for a congested operator initially assigned 10 executors in a 70 ms SLO topology, Henge reconfigures it to have an average of 18 executors, all in a single round. On the other hand, for a stricter 60 ms SLO it assigns 21 executors in two rounds. When we double the input rate of these two topologies, the former is assigned 36 executors in two rounds and the latter is assigned 44, in 5 rounds.

Henge convergence is fast. In Fig. 2.13a, convergence occurs within 2 rounds for a topology with a 60 ms SLO. Convergence time increases for stringent SLOs and higher input rates. With the 2 × higher input rate convergence time is 12 rounds for stringent SLOs of 50 ms, vs. 7 rounds for 60 ms.

Henge always reaches max utility (Fig. 2.13b) unless the SLO is unachievable (40, 50 ms SLOs). Since Henge aims to be minimally invasive, we do not explore operator migration (but we could use them orthogonally [129, 130, 136]). With an SLO of 40 ms, Henge actually performs fewer reconfigurations and allocates less resources than with a laxer SLO of 50 ms. This is because the 40 ms topology gets black-listed earlier than the 50 ms topology (Section 2.6.3: recall this occurs if utility improves $< 5\%$ in a round).

Overall, by black-listing topologies with overly stringent SLOs and satisfying other topologies, Henge meets its goal of preventing resource hogging (Section 2.3).

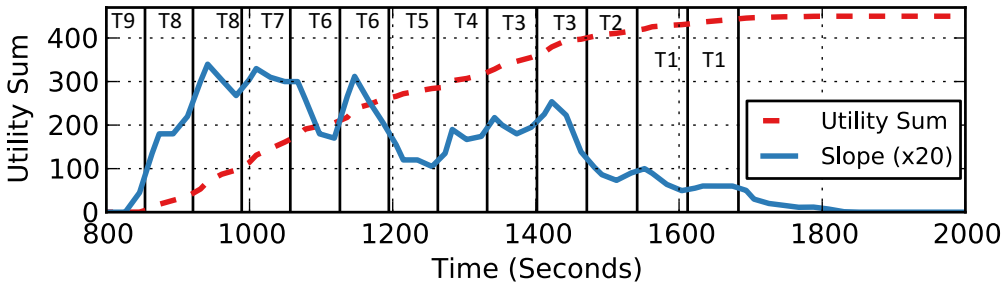


Figure 2.14: Maximizing Cluster Utility: Red (dotted) line is total system utility. Blue (solid) line is magnified slope of the red line. Vertical lines are reconfigurations annotated by the job touched. Henge reconfigures higher max-utility jobs first, leading to faster increase in system utility.

Meeting SLOs:

Maximizing Cluster Utility: To maximize total cluster utility, Henge greedily prefers to reconfigure those topologies first which have a higher max achievable utility (among those missing their SLOs). In Fig. 2.14, we run 9 PageLoad topologies on a cluster, with max utility values ranging from 10 to 90 in steps of 10. The SLO threshold for all topologies is 60 ms. Henge first picks T9 (highest max utility of 90), leading to a sharp increase in total cluster utility at 950 s. Thereafter, it continues in this greedy way. We observe some latent peaks when topologies reconfigured in the past stabilize to their max utility. For instance, at 1425 s we observe a sharp increase in the slope (solid) line as T4 (reconfigured at 1331 s) reaches its SLO threshold. All topologies meet their SLO within 15 minutes (900 s to 1800 s).

Hybrid SLOs: We evaluate a hybrid SLO that has separate thresholds for latency and juice, and two corresponding utility functions (Section 2.5.1) with identical max utility values. The job's utility is then the average of these two utilities.

Fig. 2.15 shows 10 (identical) PageLoad topologies with hybrid SLOs running on a cluster of 10 machines. Each topology has SLO thresholds of: juice 1.0, and latency 70 ms. The max utility value

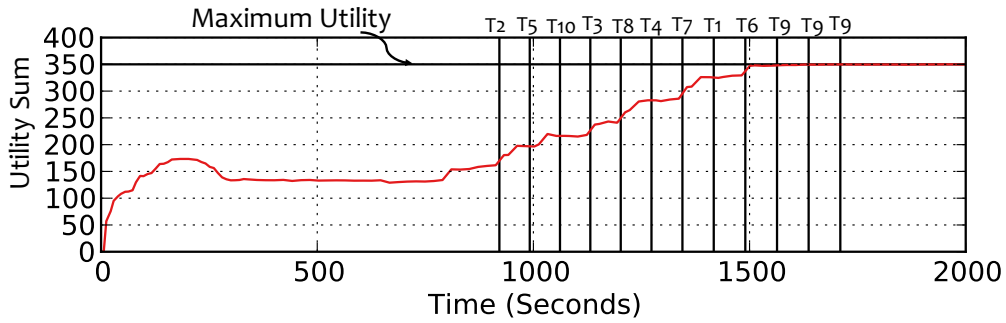


Figure 2.15: Hybrid SLOs: Henge Reconfiguration.

of each topology is 35. Henge only takes about 13 minutes (t=920 s to t=1710 s) to reconfigure all topologies successfully to meet their SLOs. 9 out of 10 topologies required a single reconfiguration, and one (T9) required 3 reconfigurations.

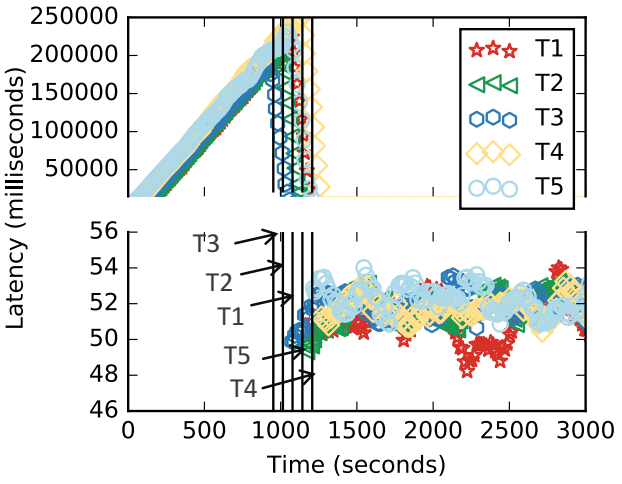
Tail Latencies: Henge can also admit SLOs expressed as tail latencies (e.g., 95th percentile, or 99th percentile). Utility functions are then expressed in terms of tail latency and the state machine remains unchanged. Fig.2.16 depicts a case where five PageLoad topologies are unable to meet their SLO. All are latency-sensitive with utilities of 35, and threshold of 80 ms. Henge performs one reconfiguration for each topology allowing all topologies to meet their SLO. The figure also shows the 99th percentile tail latency values. for the scenario. As every reconfiguration removes congestion from operators, we can see that the tail latencies improve as well.

Handling Dynamic Workloads:

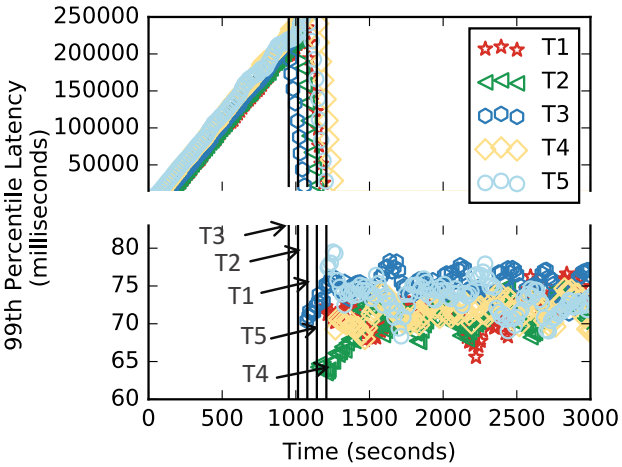
A. Spikes in Workload: Fig. 2.17 shows the effect of a workload spike in Henge. Two different PageLoad topologies A and B are subjected to input spikes. B's workload spikes by 2 $\times$, starting from 3600 s. The spike lasts until 7200 s when A's spike (also 2 $\times$) begins. Each topology's SLO is 80 ms with max utility is 35. Fig. 2.17 shows that: i) output rates keep up for both topologies both during and after the spikes, and ii) the utility stays maxed-out during the spikes. In effect, Henge completely hides the effect of the input rate spike from the user.

B. Diurnal Workloads: Diurnal workloads are common for stream processing in production, e.g., in e-commerce websites [70] and social media [128]. We generated a diurnal workload based on the distribution of the SDSC-HTTP [36] and EPA-HTTP traces [13], injecting them into PageLoad topologies.

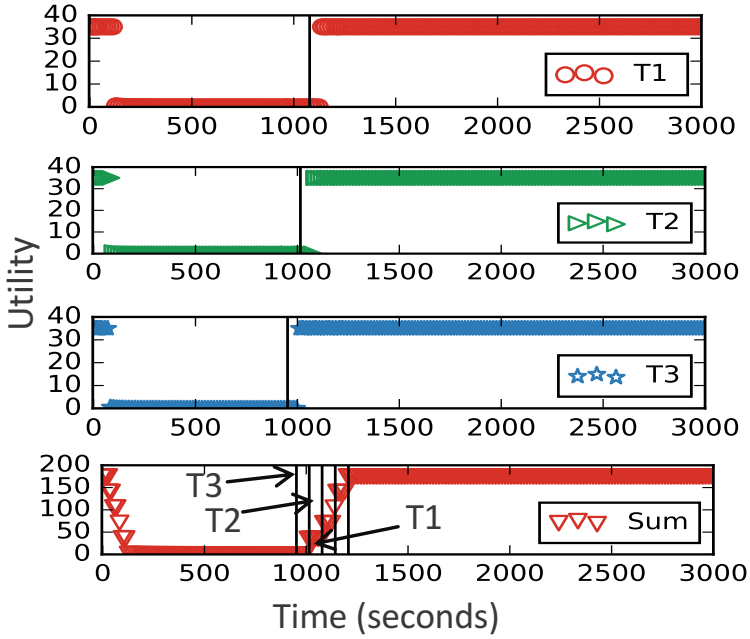
5 topologies are run with the SDSC-HTTP trace and concurrently, 5 other topologies are run with



(a) The latencies of the topologies in ms as reconfiguration occurs.



(b) The latencies of the topologies at the 99^{th} percentile



(c) Individual utilities of the topologies.

Figure 2.16: Tail Latencies: 5 PageLoad topologies run on a cluster.

the EPA-HTTP trace. All 10 topologies have max-utility=10 (max achievable cluster utility=350), and a latency SLO of 60 ms.

Fig. 2.18 shows the results of running 48 hours of the trace (each hour mapped to 10 min intervals). In Fig. 2.18a workloads increase from hour 7 of the day, reach their peak by hour $13\frac{1}{3}$, and then start to fall. Within the first half of Day 1, Henge successfully reconfigures all 10 topologies, reaching by hour 15 a cluster utility that is 89% of the max.

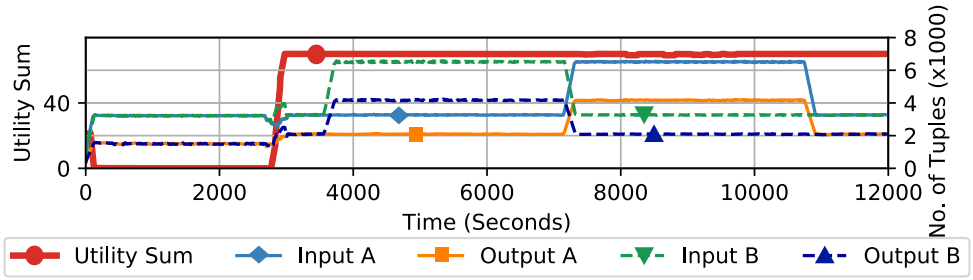


Figure 2.17: Spikes in Workload: Left y-axis shows total cluster utility (max possible is $35 \times 2 = 70$). Right y-axis shows the variation in workload as time progresses.

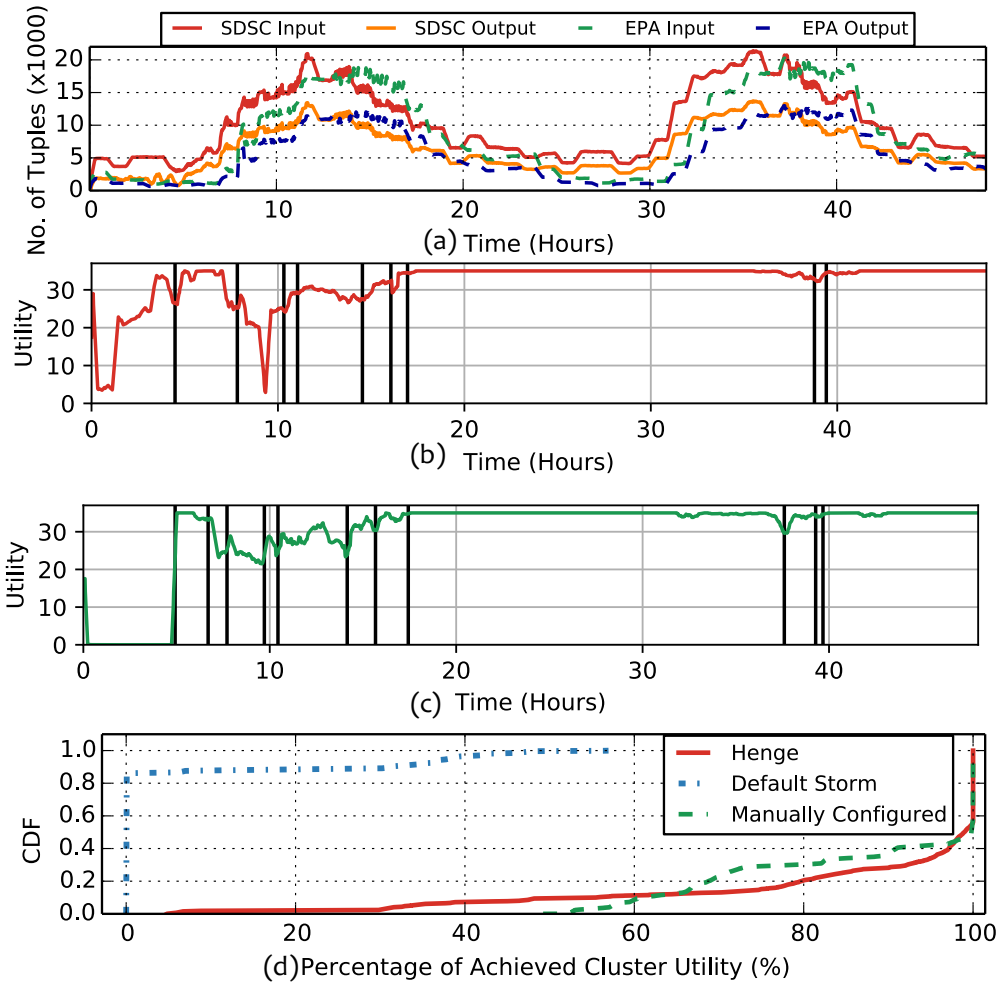


Figure 2.18: Diurnal Workload: a) Input and output rates over time, for two different diurnal workloads. b) Utility of a topology (reconfigured by Henge at runtime) with the EPA workload, c) Utility of a topology (reconfigured by Henge at runtime) with the SDSC workload, d) CDF of SLO satisfaction for Henge, default Storm, and manually configured. Henge adapts during the first cycle and fewer reconfigurations are required thereafter.

Fig. 2.18b shows a topology running the EPA workload and Fig. 2.18c a topology running the SDSC workload. Observe how Henge reconfigurations from hour 8 to 16 adapt to the fast changing workload. This also results in fewer SLO violations during the second peak (hours 32 to 40). Thus, even without adding resources, Henge tackles diurnal workloads. Fig. 2.18d shows the CDF of SLO satisfation for the three systems. Default Storm performs poorly, giving 0.0006% SLO satisfaction at the median, and 30.9% at the 90th percentile. (This means that 90% of the time, default Storm provided a total of at most 30.9% of the cluster's max achievable utility.) Henge yields 74.9% , 99.1%, and 100% SLO satisfaction at the 15th, 50th, and 90th percentiles respectively.

Henge is also better than manual configurations. We manually configured all topologies to meet their SLOs at median load.

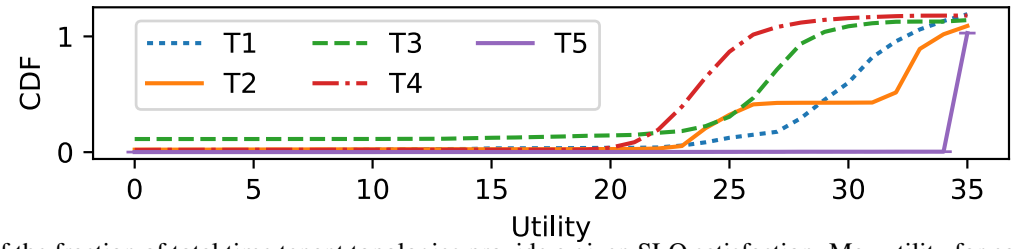
These provide 66.5%, 99.8% and 100% SLO satisfaction at the 15th, 50th and 90th percentiles respectively. Henge is better than manual configurations from the 15th to 45th percentile, and comparable from then onwards. Henge has an average of 88.12% SLO satisfaction rate, whereas default Storm and manually configured topologies provide an average of 4.56% and 87.77% respectively. Thus, Henge provides 19.3 $\times$ better SLO satisfaction than default Storm, and performs better than manual configuration.

Production Workloads: We configured the sizes of 5 PageLoad topologies based on data from a Yahoo! Storm production cluster and Twitter datasets [51], shown in Table 2.3. We use 20 nodes each with 14 worker processes. For each topology, we inject an input rate proportional to its number of workers. T1-T4 run sentiment analysis on Twitter workloads from 2011 [51]. T5 processes logs at a constant rate. Each topology has a latency SLO threshold of 60 ms and max utility value of 35.

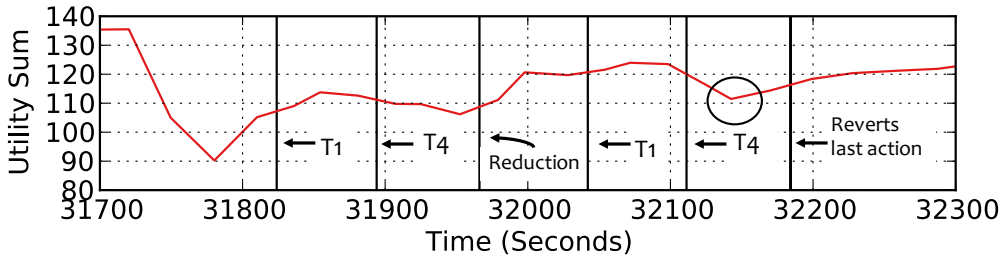
Job	Workload	Workers	Tasks
T1	Analysis (Egypt Unrest)	234	1729
T2	Analysis (London Riots)	31	459
T3	Analysis (Tsunami in Japan)	8	100
T4	Analysis (Hurricane Irene)	2	34
T5	Processing Topology	1	18

Table 2.3: Job and Workload Distributions in Experiments: Derived from Yahoo! production clusters, using Twitter Datatsets for T1-T4. (Experiments in Figure 2.19.)

This is an extremely constrained cluster where not all SLOs can be met. Yet, Henge maximizes cluster utility. Fig. 2.19a shows the CDF of the fraction of time each topology provided a given utility (including the initial 900 s where Henge is held back). T5 shows the most improvement (at the 5th percentile, it has 100% SLO satisfaction), whereas T4 shows the worst performance (at the



(a) CDF of the fraction of total time tenant topologies provide a given SLO satisfaction. Max utility for each topology is 35.



(b) Reconfiguration at 32111 s causes a drop in total system utility. Henge reverts the configuration of all tenants to that of 32042 s. Vertical lines show Henge actions for particular jobs.

Figure 2.19: Henge on Production Workloads.

median, its utility is 24, which is 68.57% of 35). The median SLO satisfaction for T1-T3 ranges from 27.0 to 32.3 (77.3% and 92.2% respectively).

Reversion: Fig. 2.19b depicts Henge's reversion. At 31710 s, the system utility drops due to natural system fluctuations. This forces Henge to perform reconfigurations for two topologies (T1, T4). Since system utility continues to drop, Henge is forced to reduce a topology (T5, which satisfies its SLO before and after reduction. As utility improves at 32042 s, Henge proceeds to reconfigure other topologies. However, the last reconfiguration causes another drop in utility (at 32150 s). Henge reverts to the configuration that had the highest utility (at 32090 s). After this point, total cluster utility stabilizes at 120 (68.6% of max utility). Thus, even under scenarios where Henge is unable to reach the max system utility it behaves gracefully, does not thrash, and converges quickly.

Reacting to Natural Fluctuations: Natural fluctuations occur in the cluster due to load variation that arises from interfering processes, disk IO, page swaps, etc. Fig. 2.20 shows such a scenario. We run 8 PageLoad topologies, 7 of which have an SLO of 70 ms, and the 8th SLO is 10 ms. Henge resolves congestion initially and stabilizes the cluster by 1000 s. At 21800 s, CPU load increases sharply due to OS behaviors (beyond Henge's control). Seeing the significant drop in cluster utility, Henge reduces two of the topologies (from among those meeting their SLOs). This allows other topologies to recover within 20 minutes (by 23000 s). Henge converges the system to the same total utility as before the CPU spike.

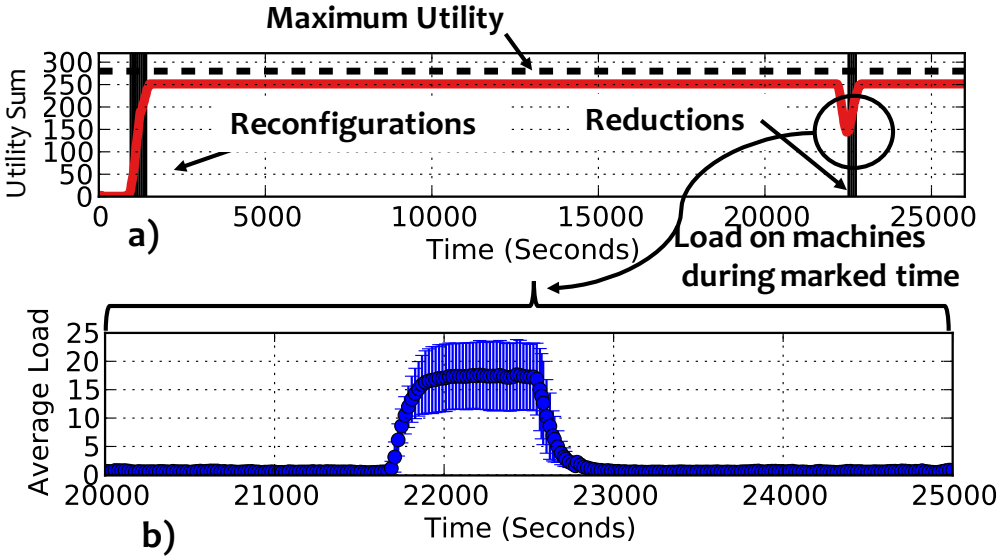


Figure 2.20: Handling CPU Load Spikes: a) Total cluster utility. b) Average CPU load on machines in the CPU spike interval.

2.9.3 Stateful Topologies

Job Type	Avg. Reconfig. Rounds (Stdev)	Average Convergence Time (Stdev)
Stateful	5.5 (0.58)	1358.7355 (58.1s)
Stateless	4 (0.82)	1134.2235 (210.5s)

Table 2.4: Stateful Topologies: Convergence Rounds and Times for a cluster with Stateful and Stateless Topologies.

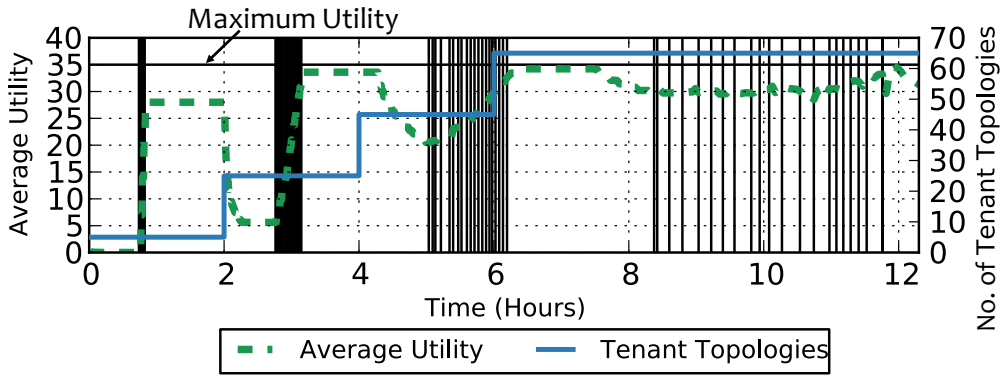
Henge handles stateful topologies gracefully, alongside stateless ones. We ran four WordCount topologies with identical workload and configuration as T2 in Table 2.3. Two of these topologies periodically checkpoint state to Redis (making them stateful) and have 240 ms latency SLOs. The other two topologies do not persist state in an external store and have lower SLOs of 60 ms. Initially, none of the four meet their SLOs. Table 2.4 shows results after convergence. Stateful topologies take 1.5 extra reconfigurations to converge to their SLO, and only 19.8% more reconfiguration time. This difference is due to external state checkpointing and recovery mechanisms, orthogonal to Henge.

2.9.4 Scalability and Fault-tolerance

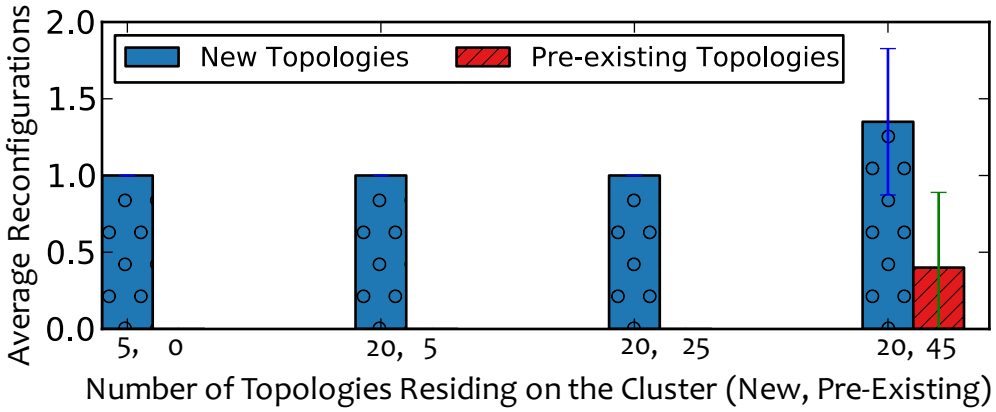
We vary number of jobs and nodes, and inject failures.

Scalability:

Increasing the Number of Topologies: Fig. 2.21 stresses Henge by overloading the cluster with topologies over time. We start with a cluster of 5 PageLoad topologies, each with a latency SLO of 70 ms, and max utility value of 35. Every 2 hours, we add 20 more PageLoad topologies.



(a) Green (dotted) line is average job utility. Blue (solid) line is number of job on cluster. Vertical black lines are reconfigurations.



(b) Average number of reconfigurations that must take place when new topologies are added to the cluster.

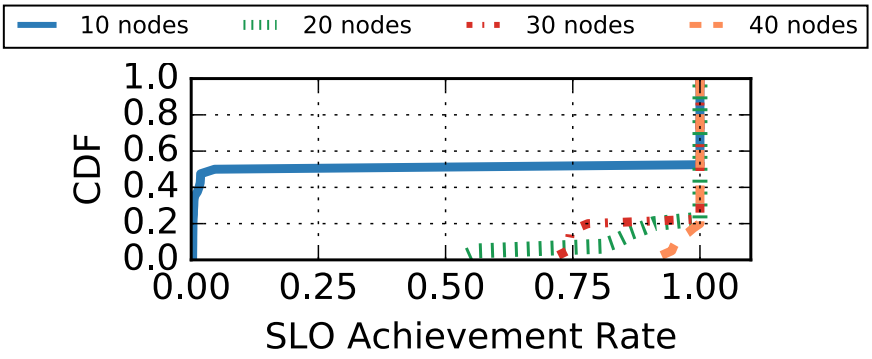
Figure 2.21: Scalability w.r.t. No. of Topologies: Cluster has 5 tenants. 20 tenants are added every 2 hours until the 8 hour mark.

Henge stabilizes better when there are more topologies. In the first 2 hours, Fig. 2.21a shows that the average utility of the topologies is below the max, because Henge has less state space to maneuver with fewer topologies. 20 new tenant topologies at the 2 hour mark cause a large drop in average utility but also open up the state space more–Henge quickly improves system utility to the max value. At the 4 hour and 6 hour marks, more topologies arrive. Henge stabilizes to max utility in both cases.

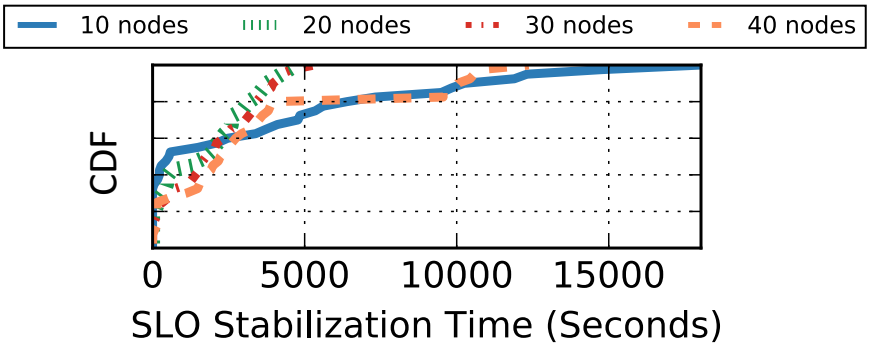
Topologies arriving at the 8 hour mark cause contention. In Fig. 2.21a, the average system utility drops not only due to the performance of the new tenants, but also because the pre-existing tenants

are hurt. Henge converges both types of topologies, requiring fewer reconfiguration rounds for the pre-existing topologies (Fig. 2.21b).

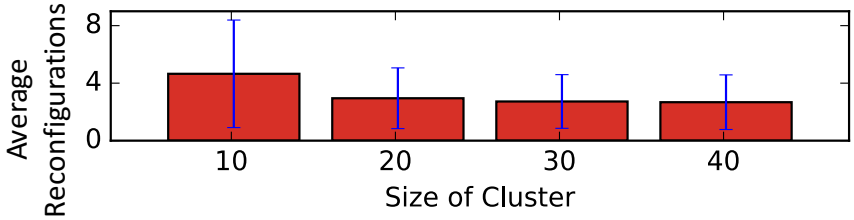
Increasing Cluster Size: In Fig. 2.22 we run 40 topologies on clusters ranging from 10 to 40 nodes. The machines have two 2.4 GHz 64-bit 8-core processors, 64 GB RAM, and a 10 Gbps network. 20 topologies are PageLoad with latency SLOs of 80 ms and max utility 35. Among the rest, 8 are diamond topologies, 6 are star topologies and 6 are linear topologies, with juice SLOs of 1.0 and max utility 5.



(a) CDF of jobs according to fraction of SLO thresholds reached.



(b) CDF of convergence time.



(c) No. of reconfigurations until convergence.

Figure 2.22: Scalability w.r.t. No. of Machines. 40 jobs run on cluster sizes increasing from 10 to 40 nodes.

From Fig. 2.22a, Henge is able to provide SLO satisfaction for 40% of the tenants even in an overloaded cluster of 10 nodes. As expected, in large clusters Henge has more room to maneuver

and meets more SLOs. This is because CPUs saturate slower in larger clusters. In an overloaded cluster of 10 nodes, topologies at the 5th percentile are able to achieve only 0.3% of their max utility. On the other hand, in clusters with 20, 30, and 40 machines, 5th percentile SLO satisfactions are higher: 56.4%, 74.0% and 94.5% respectively.

Fig. 2.22b shows the time taken for topologies to converge to their highest utility. Interestingly, while the 10 node cluster has a longer tail than 20 or 30 nodes, it converges faster at the median (537.2 seconds). Topologies at the tail of both the 10 and 40 node clusters take a longer time to converge. This is because in the 10 node cluster, greater reconfiguration is required per topology as there is more resource contention (see Fig. 2.22c). At 40 nodes, collecting cluster information from Nimbus daemons leads to a bottleneck. This can be alleviated by decentralized data gathering (beyond our scope).

Fig. 2.22c shows that the number of reconfigurations needed to converge is at most 2 $\times$ higher when resources are limited and does not otherwise vary with cluster size. Overall, Henge's performance generally improves with cluster size, and overheads are scale-independent.

Fault Tolerance: Henge reacts gracefully to failures. In Fig. 2.23, we run 9 topologies each with 70 ms SLO and 35 max utility. We introduce a failure at the worst possible time: in the midst of Henge's reconfiguration operations, at 1020 s. This severs communication between Henge and all the worker nodes; and Henge's Statistics module is unable to obtain fresh information about jobs. We observe that Henge reacts conservatively by avoiding reconfiguration in the absence of data. At 1380 s, when communication is restored, Henge collects performance data for the next 5 minutes (until 1680 s) and then proceeds with reconfigurations as usual, meeting all SLOs.

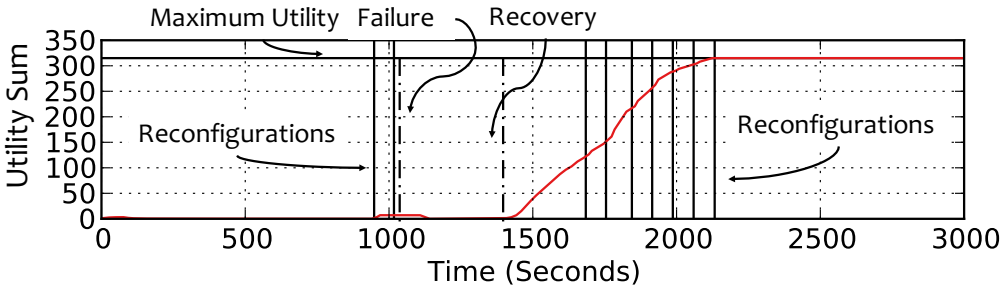


Figure 2.23: Fault-tolerance: Failure occurs at t=1020s, and recovers at t=1380 s. Henge makes no wrong decisions due to the failure, and immediately converges to the max system utility after recovery.

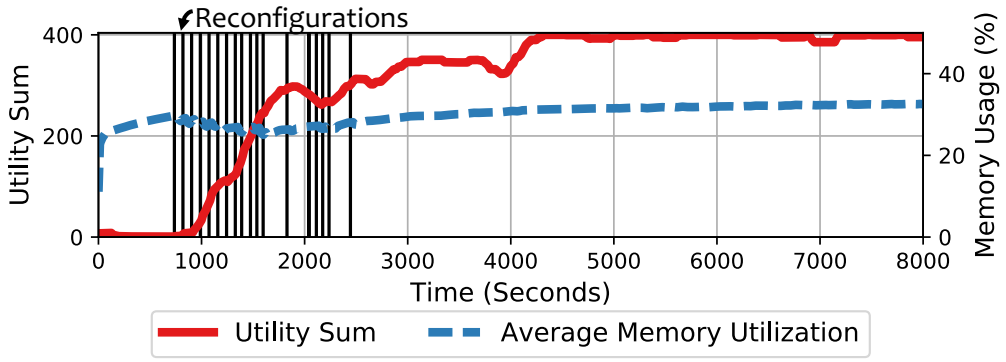


Figure 2.24: Memory Utilization: 8 jobs with joins and 30 s tuple retention.

2.9.5 Memory Utilization

Fig. 2.24 shows a Henge cluster with 8 memory-intensive topologies. Each topology has a max utility value of 50 and a latency SLO of a 100 ms. These topologies have join operations, and tuples are retained for 30 s, creating memory pressure at some cluster nodes. As the figure shows, Henge reconfigures memory-bound topologies quickly to reach total max utility of 400 by 2444s, and keeps average memory usage below 36%. Critically, the memory utilization (blue dotted line) plateaus in the converged state, showing that Henge is able to handle memory-bound topologies gracefully.

2.10 RELATED WORK

This section presents relevant related work in the areas of elastic stream processing, cluter scheduling, multi-tenant resource management and SLA/SLO satisfaction in other areas.

2.10.1 Elastic Stream Processing Systems

Traditional query-based stream processing systems such as Aurora [45] and Borealis [46] provide load-balancing [149] but not first-class multi-tenant elasticity. Modern general-purpose stream processing systems [4–7, 35, 47, 112, 127] do not natively handle adaptive elasticity. Ongoing work [12] on Spark Streaming [168] allows scaling but does not apply to resource-limited multi-tenant clusters. SEEP [63] describes a single approach for scaling out stateful operators and recovering from failures based on check-pointing but does not discuss scale-in or resource distribution in the presence of multiple tenants. Similarly, TimeStream [139] describes resilient substitution as a single method for fault recovery and resource reconfiguration without addressing multi-tenancy.

Resource-aware elasticity in stream processing [57, 64, 76, 106, 108, 135, 145] assumes infinite

resources that the tenant can scale out to. [52, 78, 93, 114, 119, 146, 163] propose resource provisioning but not multi-tenancy. Some works have focused on balancing load [129, 130], optimal operator placement [84, 105, 136] and scaling out strategies [85, 86] in stream processing systems. These approaches can be used in complement with Henge in various ways. [84–86] look at single-job elasticity, but not multi-tenancy.

Themis' [107] SIC metric is similar to juice, but the two systems have different goals. Themis measures the weight of each tuple to find those that can be dropped to reduce load, while Henge uses juice to support SLOs.

Dhalion [75] describes a system that provides the capabilities of self-regulation to stream processing systems, namely Heron. It only supports throughput SLOs and uses triggers such as backpressure to detect whether a single topology is failing and then carries out dynamic resource provisioning to resolve the issue. However, backpressure is a very coarse-grained measure of SLO satisfaction – by the time backpressure is triggered, a latency SLO in place by the user may already be violated. In addition, backpressure completely stops tuples flowing in the topology, meaning that resources are consumed for a topology even though it is doing no work. Henge simply uses SLO violations as a trigger for corrective measures and is able to satisfy both latency and throughput SLOs in a multi-tenant environment.

2.10.2 Multi-tenant Resource Management Systems

Resource schedulers like YARN [156] and Mesos [90] can be run under stream processing systems, and manually tuned [65].

Since the job internals are not exposed to the scheduler (jobs run on containers) it is impossible to make fine-grained decisions for stream processing jobs in an automated fashion.

2.10.3 Cluster Scheduling

Some systems propose scheduling solutions to address resource fairness and SLO achievement [71, 72, 80, 121, 141, 147]. VM-based scaling approaches [109] do not map directly and efficiently to expressive frameworks like stream processing systems. Among multi-tenant stream processing systems, Chronostream [164] achieves elasticity through migration across nodes. It does not support SLOs.

2.10.4 Video Stream Processing Systems

Henge is not built for video stream processing systems specifically, where techniques are needed to reduce the high cost of vision processing. For example, some existing schedulers for video stream

processing focus on the characteristics of the video stream to reduce the amount of resources used per job as much as possible. VideoStorm is a system that processes video analytics queries on live video streams [172]. VideoStorm's offline profiler generates a query resource-quality profile, while its online scheduler allocates resources to queries to maximize performance on quality and lag, as opposed to the commonly used fair sharing of resources in clusters.

Applications use video data in varying ways. One challenge is applying deep neural networks to video data, which requires massive computational resources. Chameleon [99] is a controller that dynamically picks the best configurations for existing NN-based video analytics pipelines. Chameleon utilizes the fact that underlying characteristics of video streams (e.g., the velocity and size) that affect the best configuration have enough temporal and spatial correlation to allow the search cost of finding the best configuration to be amortized over time and across multiple video feeds.

Deployments of video streaming have also been explored in wide-area scenarios, where it faces the challenges of variable WAN bandwidth. AWStream [170] is a stream processing system that achieves low latency and high accuracy in this setting through three insights: (i) it integrates application adaptation as a first-class abstraction in the stream processing model (ii) with a combination of offline and online profiling, it automatically learns an accurate profile that models accuracy and bandwidth trade-off, and (iii) at runtime, it adjusts the application data rate to match the available bandwidth while maximizing the achievable accuracy.

2.10.5 SLAs/SLOs in Other Areas

SLAs/SLOs have been explored in other areas. Pileus [152] is a geo-distributed storage system that supports multi-level SLA requirements dealing with latency and consistency. Tuba [55] builds on Pileus and performs reconfiguration to adapt to changing workloads. SPANStore [165] is a geo-replicated storage service that automates trading off cost vs. latency, while being consistent and fault-tolerant.

E-store [148] re-distributes hot and cold data chunks across nodes in a cluster if load exceeds a threshold. Cake [159] supports latency and throughput SLOs in multi-tenant storage settings.

To the best of our knowledge, Henge is the first work that combines elastic resource management for stream processing systems in a multi-tenant environment.

2.11 CONCLUSION

We presented Henge, a system for intent-driven (SLO-based) multi-tenant stream processing. Henge provides SLO satisfaction for topologies (jobs) with latency and/or throughput requirements.

To make throughput SLOs independent of input rate and topology structure, Henge uses a new relative throughput metric called juice. In a cluster, when jobs miss their SLO, Henge uses three kinds of actions (reconfiguration, reversion or reduction) to improve the sum utility achieved by all jobs throughout the cluster. Our experiments with real Yahoo! topologies and Twitter datasets have shown that in multi-tenant settings with a mix of SLOs, Henge: i) converges quickly to max system utility when resources suffice; ii) converges quickly to a high system utility when the cluster is constrained; iii) gracefully handles dynamic workloads, both abrupt (spikes, natural fluctuations) and gradual (diurnal patterns, Twitter datasets); iv) scales gracefully with cluster size and number of jobs; and v) is failure tolerant.

Chapter 3: Caladrius: A Performance Modelling Service for Distributed Stream Processing Systems

Real-time processing has become increasingly important in recent years and has led to the development of a multitude of stream processing systems. Given the varying job workloads that characterize stream processing, these systems need to be tuned and adjusted to match incoming traffic.

Current scaling systems adopt a series of trials to approach a job's expected performance due to a lack of performance modeling tools. In this chapter, we show that general traffic trends in most jobs lend themselves well to prediction. Based on this premise, we built a system called Caladrius that forecasts future traffic load of a stream processing job and predicts the job's processing performance after scaling. Experimental results show that Caladrius is able to estimate a job's throughput performance and CPU load under a given scaling configuration.

3.1 INTRODUCTION

Many use cases of the deluge of data that is pouring into organizations today require real-time processing. Common examples of such use cases include internal monitoring jobs that allow engineers to react to service failures before they cascade, jobs that process ad-click rates, and services that identify trending conversations in social networks, etc.

Many distributed stream processing systems (DSPSs) have been developed to cater to this rising demand, that provide high-throughput and low-latency processing of streaming data. For instance, Twitter uses Apache Heron [112], LinkedIn relies on Apache Samza [6] and others use Apache Flink [62]. Usually, DSPSs run stream processing jobs (or topologies) as a directed acyclic graph (DAG) of operators that perform user-defined computation on incoming data (also called tuples).

These systems are generally well-equipped to self-tune their configuration parameters and achieve self-stabilization against unexpected load variations. For example, Dhalion [75] allows DSPSs to monitor their jobs, recognize symptoms of failures and implement necessary solutions. Most commonly, Dhalion scales out job operators to stabilize their performance.

In addition to Dhalion, there are many examples from the research community of attempts to create automatic scaling systems for DSPSs. These examples usually consist of schedulers whose goal is to minimize certain criteria, such as the network distance between operators that communicate large tuples or very high volumes of tuples, or to ensure that no worker nodes are overloaded by operators that require a lot of processing resources [78, 84, 85]. While the new job deployments these schedulers produce may be improvements over the original ones, none of these systems assess whether these deployments are actually capable of meeting a performance target

or service level objective (SLO). A job's "deployment" refers to it's packing plan, defined as a mapping of operator instances to runnable containers (described in Section 3.2.)

This lack of performance prediction and evaluation is problematic: it requires the user (or an automated system) to run the new job deployment, wait for it to stabilize and for normal operation to resume, possibly wait for high traffic to arrive and then analyze the metrics to see if the required performance has been met. Depending on the complexity of the job and the traffic profile, it may take weeks for a production job to be scaled to the correct configuration.

A performance modelling system that can provide the following benefits is necessary to handle these challenges:

Faster tuning iterations during deployment Auto-scaling systems use performance metrics from physical deployments to make scaling decisions that allow jobs to meet their performance goals. Performance modelling systems that are able to evaluate a proposed deployment's performance can eliminate the need for physical job executions, thus making each iteration faster. Of course, any modelling system is subject to errors so some re-deployment may be required, however the frequency and thus, the length of the tuning process can be significantly reduced.

Improved scheduler selection A modelling system would allow several different proposed job deployments to be assessed in parallel. This means that schedulers optimized for different criteria can be compared simultaneously, which helps achieve the best performance without prior knowledge of these different schedulers.

Enabling preemptive scaling A modelling system can accept predictions of future workloads (e.g. defined by tuple arrival rate) and trigger preemptive scaling if it finds that a future workload would overwhelm the current job deployment.

Caladrius[1] is a performance modelling service for DSPSs.

The aim of this service is to predict potential job performance under varying traffic loads and/or deployments, thus 1) reducing the time required to tune a topology's configuration for a given incoming traffic load. This significantly shortens the loop of 'plan –> deploy –> stabilize –> analyze' that is currently required to tune a topology; 2) enabling preemptive scaling before disruptive workloads arrive. Caladrius can be easily extended to provide other analyses of topology layouts and performance.

Caladrius provides a framework to analyze, model and predict various aspects of DSPSs (such as Apache Heron [112] and Storm [154]) and focuses on two key areas:

[1]This is the Roman name for the legend of the healing bird that takes sickness into itself

- *Traffic:* The prediction of the incoming workload of a stream processing topology. Caladrius provides interfaces for accessing metrics databases and methods that analyze traffic entering a topology and predict future traffic levels.
- *System Performance:* The prediction of a topology's performance under a given traffic load and deployment plan. This can be broken down into two scenarios:
  - ⋆ *Under varying traffic load:* The prediction of how a topology will perform under different or future traffic levels if it keeps its current deployment.
  - ⋆ *Using a different deployment plan:* The prediction of how a topology will perform under current traffic levels if its layout is changed.

1. Motivated by challenges that users face, we introduce the notion of DSPSs modeling to enable fast job tuning and preemptive scaling, and discuss its properties.
2. We present Caladrius, the first performance modelling and evaluation tool for DSPSs (Section 3.3, 3.4). Caladrius has a modular and extensible architecture and has been implemented on top of Apache Heron.
3. We validate the proposed models and present one use case for performance assessment and prediction for stream processing topologies (Section 3.5).

We discuss related work in Section 3.6. Finally, we draw conclusions in Section 3.7. In the next section, we start by presenting job scaling and scheduling background (Section 3.2).

3.2 BACKGROUND

This section presents a brief overview of DSPSs as well as related concepts and terminologies, particularly those belonging to Apache Heron [112], on which Caladrius is built. Terms are formatted as *italics* in this section and the rest of the chapter aligns to the term definitions here.

3.2.1 Topology Components

A stream processing *topology* can be logically represented as a DAG of *components* or *operators*. A component is a logical processing unit, defined by the developer, that applies user-defined functions on a stream of input data, called *tuples*. The edges between components represent the dataflow between the computational units. Source components are called *spouts* in Heron terminology, and they pull tuples into the topology, typically from sources such as pub-sub systems like Kafka [111]. Tuples are processed by downstream components called *bolts*.

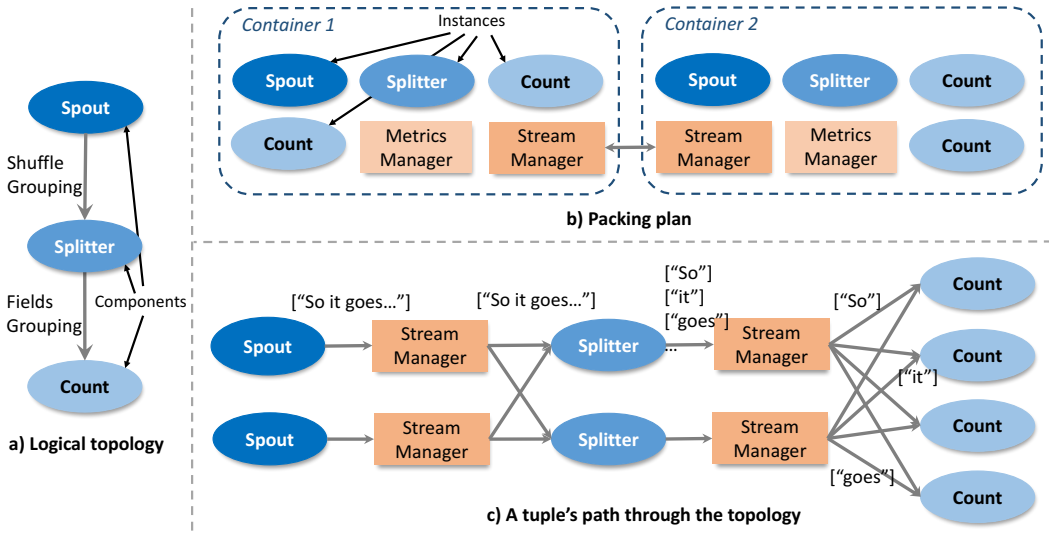


Figure 3.1: Sentence-Word-Count Topology a) A logical DAG of the topology as the developer writes it. b) A physical representation of the topology when it is run. c) A possible path an input might take through the topology.

We define three kinds of throughputs: 1) *source throughput*, the throughput that the source pub-sub system provides, waiting to be processed by the topology; 2) *spout throughput*, the throughput processed by spouts which allows data to enter the topology; 3) *output throughput*, the sum output throughput of all sink bolts i.e. the job's final output throughput. We use the terms *throughput* and *traffic* interchangeably in this chapter.

A job can be saturated due to limited resources where its processing throughput cannot catch up with the pub-sub system's source throughput. Then, data accumulates in the source pub-sub system waiting to be fetched. For example, we can have a topology where Kafka generates *source throughput* = 5 million tuples in a minute, the topology's spouts try their best to read from Kafka with *spout throughput* = 3 million tuples, and the whole topology generates *output throughput* = 10 million tuples. In this example, Kafka accumulates $5 - 3 = 2$ million tuples.

3.2.2 Topology Instances

The developer specifies how many parallel *instances* there should be for each component: this is called the component's *parallelism level*. In other words, instances perform the same processing logic on different input data. All instances of the same component are of same resource configuration. The developer also specifies how tuples from each component's instances should be partitioned amongst the downstream component's instances. These partitioning methods are called *grouping*. The most common grouping is called *shuffle grouping*, where data can be partitioned randomly across downstream instances. *Field grouping* chooses the downstream instance based on the hash of

the data in a tuple whereas *all grouping* replicates the entire stream to all the downstream instances.

3.2.3 Implementation of DSPSs

We consider long-running stream processing topologies with a continuous operator model. A *topology master* (*tmaster*) is responsible for managing the topology throughout its existence and provides a single point of contact for discovering the status of the topology. Each topology is run on a set of containers using an external scheduler, e.g. Twitter uses Aurora [45] for this purpose. Each container consists of

- one or more *Heron-Instances* (*instances*),
- a *metrics manager* and
- a *stream manager* (*stmgr*),

each of which is run as a dedicated Java process. The metrics manager is responsible for routing metrics to the topology master or a central metrics service in data center, while the stream manager is responsible for routing tuples between running instances. An instance processes streaming data one tuple at a time (thus emulating the continuous operator model) and forwards the tuples to the next components in the topology, via the stream manager. Systems that follow such a model include Apache Heron [112], Samza [6], Flink [62] and Storm [154].

3.2.4 A Topology Example

We present a sample topology in Figure 3.1. Figure 3.1a) shows the logical representation of the topology, where tuples are ingested into the topology from the spout, and passed onto Splitter bolts that split sentences in the tuples to words. The resulting tuples are then passed onto the Counter bolts that count the occurrence of each unique word.

Figure 3.1b) shows how the topology may look when launched on a physical cluster. The parallelism level of the spout and the Splitter bolt are both 2 and the parallelism level of the Counter bolt is 4. The topology is run in two containers, each of which contains a stream manager for inter-instance communication. This representation of a topology is called its *packing plan*.

Figure 3.1c) shows a possible path a tuple could take in a topology. Though only one path is shown here, there are 16 possible paths for the job, given the parallelism levels of the components. Stream managers are used for passing tuples between two instances. If two instances on the same container need to communicate, data will only pass through the local container's stream manager. On the other hand, if the instances run on different containers, the sender's output tuples will first go

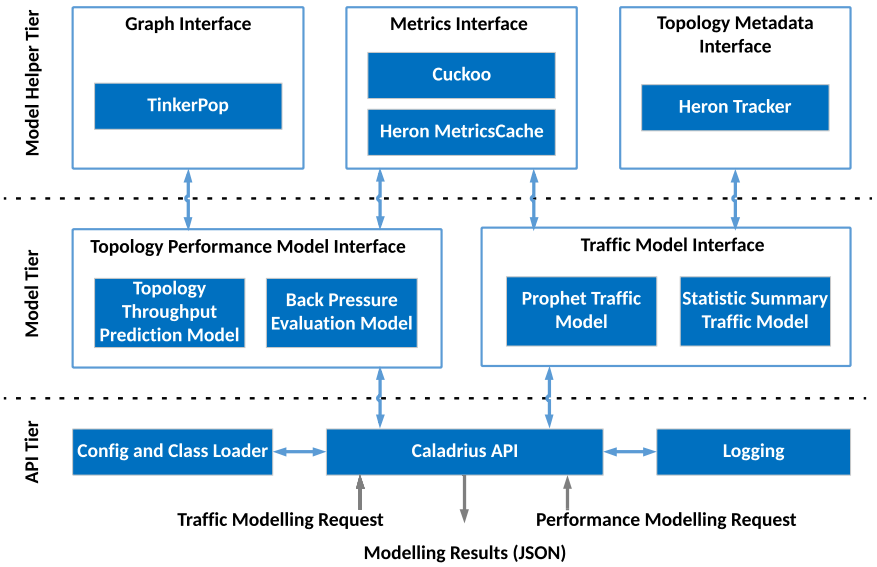


Figure 3.2: Caladrius System Overview.

to its local stream manager, who will then send them to the stream manager on the remote container. The receiving stream manager will be in charge of passing those tuples onto the local receiving instance. Note that this does not increase the number of possible paths in the topology.

3.3 SYSTEM ARCHITECTURE

This section presents a brief overview of Caladrius' architecture, particularly with respect to its interface with Apache Heron [112]. Caladrius consists of three tiers—the API tier, the model tier and the shared model helper tier—as shown in Figure 3.2. It is deployed as a web service that can easily be launched in a container, and is accessible to developers through a RESTful API provided by the API tier.

3.3.1 API Tier

The API tier faces users who would like to use Caladrius to predict future traffic levels or performance of a topology. It is essentially a web server translating and routing user HTTP requests to corresponding interfaces. Caladrius exposes several RESTful endpoints to allow clients to query the various modelling systems it provides. Currently, Caladrius provides topology performance (throughput, back pressure etc.) and traffic (incoming workload) modelling services. Besides performing HTTP request handling, the API tier is also a web container that hosts implementa-

tions of the aforementioned models and fulfills system-wide common shared logistics including configuration management and logging, etc.

It is important to consider that a call to the topology modelling endpoints may incur a wait (up to several seconds, depending on the modelling logic). Therefore, it is prudent to let the API be asynchronous, allowing the client to continue with other operations while the modelling is being processed. Additionally, an asynchronous API allows the server side calculation pipelines to run concurrently with ease.

A response for a call to a RESTful endpoint hosted by the API tier is a JSON formatted string. This string contains the results of modelling and prediction. The type of results listed can vary by model implementation. By default, the endpoint will run all model implementations defined in the configuration and concatenate the results into a single JSON response.

3.4 MODELS: SOURCE TRAFFIC FORECAST & TOPOLOGY PERFORMANCE PREDICTION

As discussed in the Section 3.3, Caladrius is a modular and extensible system that allows users to implement their own models to meet their performance estimation requirements. Here, we present a typical use case of Caladrius to evaluate one of the four golden system performance signals – "traffic", and extensively discuss the implementation of the "source traffic forecast" and "topology performance prediction" models on top of Heron. Our models can be applied to other DSPSs as long as they employ topology-based stream flow and backpressure-based rate control mechanisms.

3.4.1 Source Traffic Forecast

Caladrius must be able to forecast a job's source traffic rate. This is necessary for finding out the topology's performance in the near future. We measure the job's source traffic rate as a timeseries for the purposes of forecast. The timeseries consists of tuple counts waiting to be processed by a topology per minute per minute. We assume that no bottlenecks exist between the spouts and the external source, such that spout throughput is the same as source throughput. Intuitively, this remains true unless the topology experiences backpressure where spout throughput falls below source throughput.

Time series forecasting is a complex field of research and many methods for predicting future trends from past data exist. A simple statistics summary such as mean or median of the last given period of data points may be sufficient for a reasonable forecast of a random timeseries.

However, we find that a large percentage of topologies in the field depict strong seasonality. A simple statistical model is not able to predict such strongly seasonal traffic rates. To deal with

seasonality, we use more sophisticated modelling techniques. Specifically, we use Facebook's Prophet, a framework for generalized timeseries modelling [150]. It is based on an additive model where non-linear trends are fit with periodic (yearly, weekly, daily, etc.) seasonality. It is robust to missing data, shifts in the trend, and large outliers [150].

Caladrius allows users to pass the model a length of historic data and specify whether a single Prophet model should be used for tuple streams entering each spout as a whole, or separate models should be created for tuple streams entering each spout instance. A spout can have many instances, depending on its parallelism level. The latter method is slower but more accurate. The user also specifies the time in future we should forecast the traffic rate. The model then produces various summary statistics e.g. mean and median, for the predicted input rate at the future instance, based on historic data.

3.4.2 Topology Performance Prediction

The previous subsection described how we forecast source traffic levels for a topology. This incoming data is passed to downstream components that process it to find results. To make an accurate performance estimation about how the topology will perform at a particular traffic level, we must study the impact of the traffic level on each instance's performance.

Modelling Single Instance Throughput

Based on our production experience with Heron, we have the following topology performance observations and we summarize them into assumptions used for topology performance modelling.

Neither Heron-Stmgr nor the network is a bottleneck. The Heron-Stmgr behaves like a router inside the container. It routes all the traffic for all instances in the container. As the Heron-Stmgr is hidden from the end user's perspective, it is hard for the end users to reason about it if it forms a bottleneck. Thus, almost all users in the field allocate a large number of containers to their jobs. This means that there are a few instances per container, ensuring that the Heron-Stmgr is never a bottleneck despite variable input rates. Thus, we assume that the throughput bottleneck is not Heron-Stmgr and backpressure is triggered only when the processing speed of the instances is less than their input throughput.

Backpressure is either 0 or 1; there is no intermediate state. In Heron, backpressure is triggered by default if the amount of data pending for processing at one instance exceeds 100Mb (high water mark). Backpressure is resolved if the amount of pending data is below 50Mb (low water mark). Given Twitter's traffic load, tiny traffic variance can push 50Mb of data to

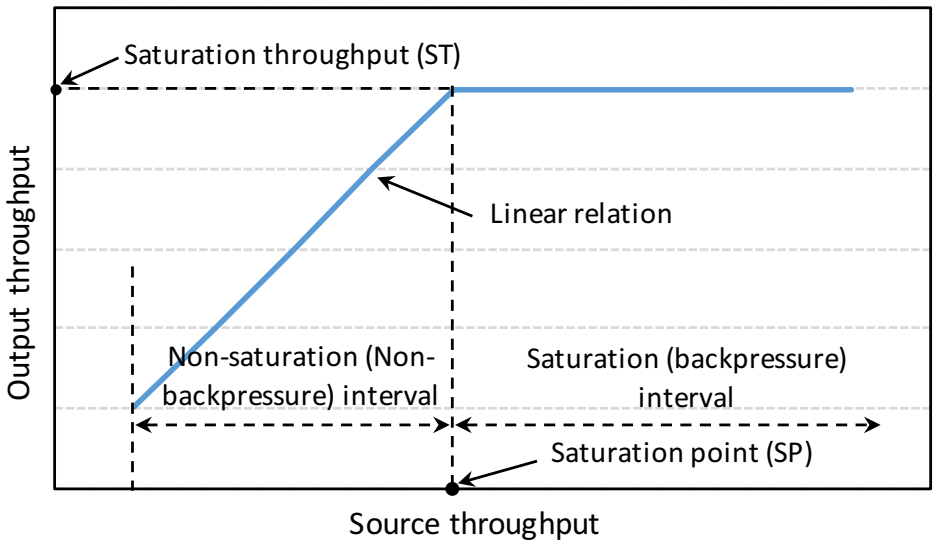


Figure 3.3: Topology performance observation in Heron production experience.

instances. This means that although the instances just finished processing data and reduced the amount of pending data below the lower water mark, just enough data is pushed to them that the high water mark is exceeded again. This forces the instance to continue to be in backpressure unnecessarily.

Heron adopts a metric named "backpressure time" to measure how many seconds (in the range from 0 to 60) in a minute when the topology is in backpressure state. Based on the above, "backpressure time" is either close to 60 or is 0, rather than being evenly distributed. Thus, we can approximate the topology's backpressure state to be either 0 or 1.

Based on the assumptions above, we draw the output throughput performance of an instance with a single upstream instance and single downstream instance in Figure 3.3. We observe the following features:

Saturation Point (SP) of source throughput: If input traffic rates exceed a threshold, a job's instances can trigger backpressure. We call this the saturation point (SP) of the instances. When input traffic rises beyond the SP, instances will always face backpressure.

Saturation Throughput (ST) of output throughput: After the input traffic rate exceeds the SP, an instance's output throughput reaches and stays at a maximum value, called its saturation throughput (ST). This is because even though input rates are rising, the instance is already pushed to its maximum processing rate.

Linear relation (α) of output and source throughputs: When backpressure is not present and the instance is not saturated, its processing throughput is the same as its input throughput. A linear relation between output throughput and input throughput exists, and its slope (α) represents the amplifier coefficient of the instance which is determined by its linear processing logic (together with grouping types if multiple downstream instances exist). Intuitively, $ST = \alpha SP$.

It should be noted that for a large amount of tuples, such as those belonging to Twitter's traffic load, the variation of processing times per tuple are normalized, rendering the processing speed steady and content agnostic.

Given these observations, we express the output throughput T_i of a single-input single-output instance i against source throughput t_λ, which is most commonly seen in production, as follows:

$$T_i(t_\lambda) = \begin{cases} \alpha_i t_\lambda & : t_\lambda < SP_i \\ ST_i & : t_\lambda \geq SP_i \end{cases} \tag{3.1}$$

or simply

$$T_i(t_\lambda) = \min(\alpha_i t_\lambda, ST_i). \tag{3.2}$$

The output throughput of instances with multiple (m) input streams can be calculated as:

$$T_i(t_\lambda) = \sum_{\lambda=1}^{m} \min(\alpha_i t_\lambda, ST_i) \tag{3.3}$$

This approach assumes that input and output streams have a linear relationship, which works well in practice for most topologies. When an instance has only one input, Equation 3.3 reduces to Equation 3.2.

Moreover, if there are n outputs, Equation 3.3 becomes:

$$T_i(t_\lambda) = \sum_{j=1}^{n} T_j(t_\lambda) \tag{3.4}$$

$$T_j(t_\lambda) = \sum_{\lambda=1}^{m} \min(\alpha_j t_\lambda, ST_j), j \in [1, 2, ..., n] \tag{3.5}$$

where $T_j(t_\lambda)$, α_j and ST_j represent the output throughput, the amplifier coefficient and the saturation throughput of the jth output stream respectively. α_j is determined by both the instance's processing logic and the type of tuple grouping.

Modelling Single Component Throughput

Adding the throughput of all instances of a component gives the component's throughput. Let's consider a single-input single-output component c first; the multi-input multi-output component's throughput can be derived from the single-input single-output component in the same way as done for instances in section 3.4.2. Given the component's parallelism level p, let the input throughput of each instance of the component be $t_{\lambda(1)}, t_{\lambda(2)}, ..., t_{\lambda(p)}$. The component's input throughput is then:

$$t_\lambda = \sum_{i=1}^{p} t_{\lambda(i)}. \tag{3.6}$$

The component output throughput is:

$$T_c(p, t_\lambda) = \sum_{i=1}^{p} T_i(t_{\lambda(i)}). \tag{3.7}$$

Since a component's instances have the same code, they perform the same function $T()$. However, the input throughput $t_{\lambda(i)}$ to each instance i may not be same due to the upstream grouping type, specified by the user. Here we discuss the impact of the most commonly used grouping types of shuffle (round robin or load balanced) and fields (key) grouping.

Shuffle Grouped Connections Shuffle grouped connections between components share output tuples evenly across all downstream instances, leading to

$$t_{\lambda(1)} = t_{\lambda(2)} = ... = t_{\lambda(p)} = \frac{t_\lambda}{p}. \tag{3.8}$$

This means that the routing probability from a source instance to a downstream instance is always simply $1/p$ where p is the number of downstream instances, irrespective of the input traffic's content and changes in it over time.

The component output throughput is:

$$T_c(p, t_\lambda) = pT_i(\frac{t_\lambda}{p}). \tag{3.9}$$

Particularly, when $p = 1$, the component has a single instance and Equation 3.9 reduces to $T_c = T_i$. When $p > 1$, Equation 3.9 shows that T_c becomes T_i times the number of instances (p).

Consider a component running for a while with seasonally varying input rates. We observe several data points of the same parallelism p and a range $t_\lambda \in (\eta_1, \eta_2)$ of input. Thus, we can draw a line $T_c(t_\lambda)$ similar to Figure 3.3 as long as SP exists in the range (η_1, η_2). This line corresponds to

the particular parallelism p. Given this line, we can draw another line of a new parallelism $p' = \gamma p$ by scaling the existing line by γ.

Fields Grouped Connections Fields grouping chooses downstream instances based on the hash of the data in the tuple and thus can depend on the content of the data. However, we observed that the data set bias towards certain downstream instances remains unchanged in a long-time window, or the bias changes slowly. This bias can be measured by periodically auditing traffic. Thus, we assume that the input traffic bias remains unchanged over time in the following discussions. We discuss two changes in a job's execution: 1) varying input traffic load and 2) scaling a component to a new parallelism.

- **Varying source traffic load while keeping parallelism constant:**

  By observing the input throughput at a particular parallelism level of an operator, we can identify whether the data flow is biased towards a subset of instances belonging to downstream components. This allows us to predict the amount of data each downstream instance will receive if the input rate changes.

  Let the new overall input throughput be $t'_\lambda = \beta t_\lambda$. With the steady data set bias assumption, traffic distribution is measured along time and is distributed across all p parallel instances of the operator.

  Thus, we have:

$$t'_\lambda = \beta t_\lambda = \sum_{i=1}^{p} \beta t_{\lambda(i)}. \tag{3.10}$$

  We can calculate the component output throughput under a different input traffic load (Equation 3.11). We observe that the output throughput of each instance is proportional to the original one by β when its new input traffic load falls into the linear interval, and reaches the saturation throughput otherwise.

$$\begin{aligned} T_c(p, t'_\lambda) &= T_c(p, \beta t_\lambda) \\ &= \sum_{i=1}^{p} T_i(\beta t_{\lambda(i)}) \\ &= \sum_{i=1}^{p} \min\left(\beta T_i(t_{\lambda(i)}), ST_i\right) \end{aligned} \tag{3.11}$$

- **Varying parallelism while keeping input traffic load constant:**

  Changing parallelism can effect how tuples are distributed among instances when using fields grouping. This complicates the calculation of the routing probability for fields grouped

connections under a different parallelism. The routing probability is a function of the data in the tuple stream and their relative frequency; thus, the proportion of tuples that go to each downstream instance depends entirely on the nature of the data contained in the tuples.

Fields grouping chooses an instance to send data to by moding the hash value of the tuple by the number of parallel instances. As the hash result depends on the tuple data, the result is unpredictable. However, we found in some cases that the data set distribution is uniform or load-balanced in a large data sample set. Under this circumstance, the component behaves as Equation 3.9. A potential solution for a biased data set is that a user can implement their own Heron-customized-key-grouping to make the traffic distribution predictable and plug the corresponding model into Caladrius.

Modelling Topology Throughput

A topology is a DAG, which can contain a critical path – the path which limits the whole topology's throughput. Once the model for each component is built, the throughput performance on the critical path can be evaluated. We assume that there are N components on the critical path, and the input traffic coming through the spouts is t_0, which can be either the measured actual throughout or forecasted source throughput in Section 3.4.1. The user specifies the parallelism configuration for each component to be $\{p_1, p_2, ..., p_N\}$. The output throughput of the critical path (t_{cp}) can be calculated by chaining Equation 3.7:

$$t_{cp} = \underbrace{T_{c(N)}(p_N, T_{c(N-1)}(...T_{c(2)}(p_2, T_{c(1)}}_{\text{N}}(p_1, t_0)))...) \tag{3.12}$$

Once we have t_{cp}, we can trace backwards and find the saturation point of the topology:

$$t'_0 = \underbrace{T^{-1}_{c(1)}(T^{-1}_{c(2)}(...T^{-1}_{c(N-1)}(T^{-1}_{c(N)}}_{\text{N}}(t_{cp})))...) \tag{3.13}$$

Moreover, we can identify if there is or will potentially be backpressure by comparing t_0 and t'_0:

$$risk_{backpressure} = \begin{cases} low & : t'_0 < t_0 \\ high & : t'_0 \sim t_0 \end{cases} \tag{3.14}$$

We can also locate the component or instance with high backpressure risk while creating the chain in Equation 3.12. For some topologies, the critical path cannot be identified easily. Under this situation, multiple critical path candidates can be considered and predicted at the same time. The critical path selection problem is out of the scope of this chapter.

3.5 EXPERIMENTAL EVALUATION

As we depend on the Prophet library for the source traffic forecast modeling, the performance evaluation of Caladrius' traffic prediction will not be discussed here. We focus on the evaluation of the topology performance prediction model and its integration into Caladrius as an end-to-end system. The evaluation is conducted in two main parts.

1. First, we evaluate the output throughput. We validate our observation and assumptions for the single instance in Section 3.5.2, and the models for the single component in Section 3.5.3 and the critical path in Section 3.5.4. DSPSs usually provide scaling commands to update the parallelism of their components. For example, Heron provides the 'heron update [parallelism]' command to update a component's parallelism in the topology DAG. Although users have tools to scale jobs, it is hard to predict changes in performance after running the commands. Some existing systems, such as Dhalion, use several rounds of 'heron update [parallelism]' to converge Heron topologies to the users' expected throughput service level objective (SLO) in a time-consuming process.

   Conversely, Caladrius can predict the expected throughput given new component parallelism levels, which gives users useful insights on how to tune their jobs. This can be done by executing the command 'heron update –dry-run [parallelism]'. It should be noted that with the '–dry-run' parameter, the new packing plan and the expected throughput is calculated without requiring job deployment, thus significantly reducing iteration time.

2. Besides throughput, we also conduct CPU load estimation for updated parallelism levels in Section 3.5.5. The CPU load primarily relates to the processing throughput, which makes its prediction feasible once we have a throughput prediction.

3.5.1 Experimental Setup

Previous work on DSPSs [75] used a typical 3-stage Word Count topology to evaluate the systems under consideration. In our work, we use the same topology, shown in Figure 2.1. In this topology, the spout reads a line in from the fictional work *"The Great Gatsby"* as a sentence and emits it. The spouts distribute the sentences to the bolts (Splitter) belonging to the second stage of the topology using shuffle grouping. The Splitter bolts split the sentences into words that subsequently forward the words to the third stage bolts (Counter) using fields grouping. Finally, the Counter bolts count the number of times each word has been encountered. Unless mentioned otherwise, the spout's parallelism in each experiment is set to 8.

As there is no external data source in the experiments, the spout output traffic matches the

configured throughput if there is no backpressure triggered by the job instances, and their throughput is reduced if backpressure is triggered.

We run the topology on Aurora, a shared cluster with Linux "cgroups" isolation. The topology resource allocation is calculated by Heron's round-robin packing algorithm – 1 CPU core and 2GB RAM per instance, with disk not involved in the evaluation. Note that the evaluation topology was constructed primarily for this empirical evaluation, and should not be construed as being the representative topology for Heron workloads at Twitter.

We use throughput (defined in Section 3.2) as the evaluation metric in our experiments. We note that the throughput of a spout is defined as the number of tuples emitted by the spout per minute. The throughput of a bolt is defined as the number of tuples generated by the bolt per minute.

We tune the Word Count topology to perform in ways that we expect in production settings i.e., there are no out-of-memory (OOM) crashes, or any other failure due to resource starvation during scheduling or long repetitive GC cycles. The experiments were allowed to run for several hours to attain steady state before measurements were retrieved.

3.5.2 Single Instance Model Validation

To validate the single instance model in Figure 3.3, we set the Splitter component's parallelism to 1. The topology was configured to have an input traffic rate of from 1 to 20 million tuples per minute with an additional step of 1 million tuples per minute. Meanwhile, the Counter parallelism is set to 3 to prevent it from being the bottleneck. We collect the Splitter processed-count and emit-count metrics as they represent the instance's input and output throughput. The observation was repeated for 10 times, and the throughput with 0.9 confidence is drawn in Figure 3.4.

There are two series of measurements of the Splitter instance in Figure 3.4. One is the input throughput and the other is output throughput. The x-axis is the spout's output rate, and the y-axis shows the two series value in million tuples per minute. We can see the two series increase until around the point of 11 million spout emit-throughput, which is the SP. After SP, both series tend to be steady, among which the output throughput is the ST.

Figure 3.5 shows the ratio of output over input throughput, which is between 7.63 and 7.64 - a very tight window, thus it can be roughly treated as a constant value. The slope actually represents on average, the number of words in a sentence.

Moreover, we noticed the slope slightly fluctuates in the non-saturation interval, which is possibly due to competition for resources in instances. An instance contains two threads: a gateway thread that exchanges data with the Heron-Stmgr and a worker thread that performs the actual processing of the instance. When the input rate increases, the burden on the instance gateway thread and communication queues increases, which results in less resources allocated to the processing thread.

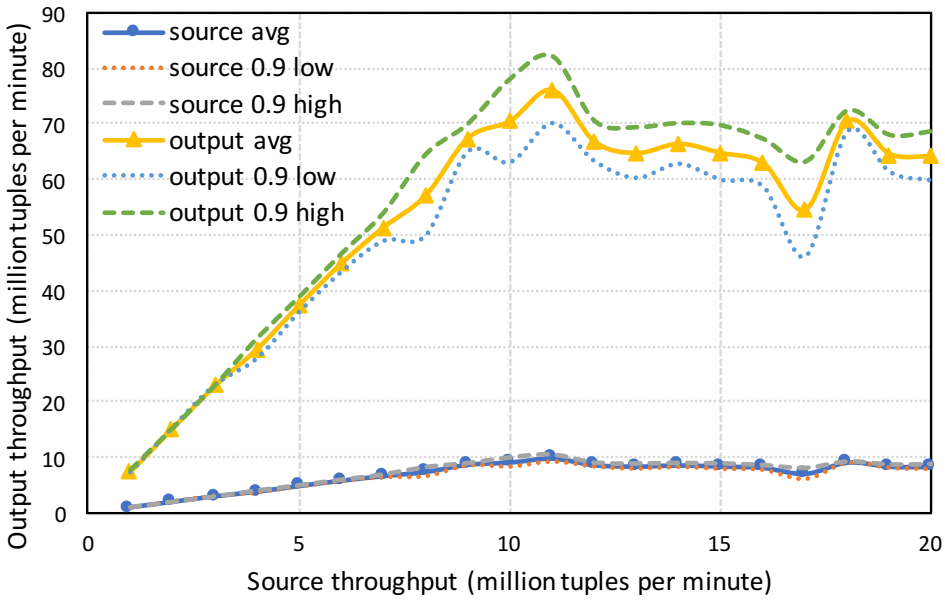


Figure 3.4: Instance output and input throughput vs. topology input rate.

However, the performance degradation in the processing thread is small and transient.

Time spent in backpressure is presented in Figure 3.6. It can be observed that backpressure occurs when the spout throughput reaches around 11 million (SP). The time spend in backpressure rises steeply from 0 to around 60000 milliseconds (1 minute) after it is triggered.

From the observation above, we note that to uniquely identify the curve for a given instance, we need at least two data points: one falls in the non-saturation interval and one in the saturation interval. We can get these points from two experiments: one without and one with backpressure.

3.5.3 Single Component Model Validation and Component Throughput Prediction

When we observe jobs in a data center, source traffic varies along time and may have the same throughput at multiple times. This means that we can observe multiple instances of a particular source traffic rate. In the experiments, we emulate multiple observations of the same input rate by restarting the topology and observing its throughput multiple times.

To validate the single component model, we follow the previous single instance evaluation: we focus on the Splitter component and start with a parallelism of 3 as in Figure 3.7.

We can see that the throughput lines of a component have similar shape to those of the instances shown in Figure 3.4, but scaled according to the parallelism. The topology input rate ranges from 2 to 68 million tuples per minute, and the SP is around 30 million. The piecewise linear regression lines are also marked as dash lines, and the output over input ratio can be calculated to be 7.638

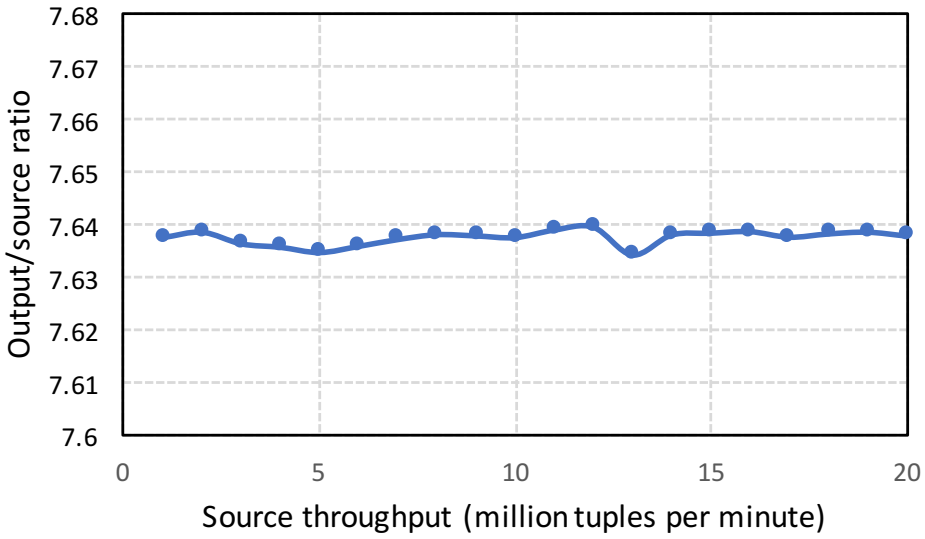


Figure 3.5: Instance output/input throughput ratio vs. topology input rate.

(210.239/27.526), which is consistent with the result in Figure 3.5.

Based on the observation of the Splitter bolt with a parallelism of 3, we can predict its throughput with another parallelism level. Given the discussions of Equation 3.9, we plotted the predictions of throughput with parallelism 2 and 4, as dashed lines for both input and output, in Figure 3.7. The predicted input and output ST with a parallelism of 2 is around 18 million and 140 million respectively, while those for parallelism of 4 are 36 million and 280 million.

To evaluate the prediction, we deployed the topology with Splitter component parallelism to be 2 and 4, and measured throughput as shown in the Figure 3.8. In the non-backpressure interval, the predicted curves strictly match the measured ones. The ST prediction error, which is defined as the difference between the corresponding predicted and observed regression lines over the observed regression line of output throughput ($|ST_{prediction} - ST_{observation}|/ST_{observation}$), is around $(140 - 136)/136 = 2.9\%$ for parallelism of 2 and $(287 - 280)/280 = 2.5\%$ for parallelism 4.

We can see that the ST predictions of parallelism 2 and 4 match well with the measured ones, with acceptable small variations.

3.5.4 Critical Path Model Validation and Topology Throughput Prediction

For the example job in Figure 2.1, the critical path is the only path going through the three components. In the previous experiments, we have built the model for the Splitter component. We did the same for the Counter component and show its model in Figure 3.9. Moreover, we observed

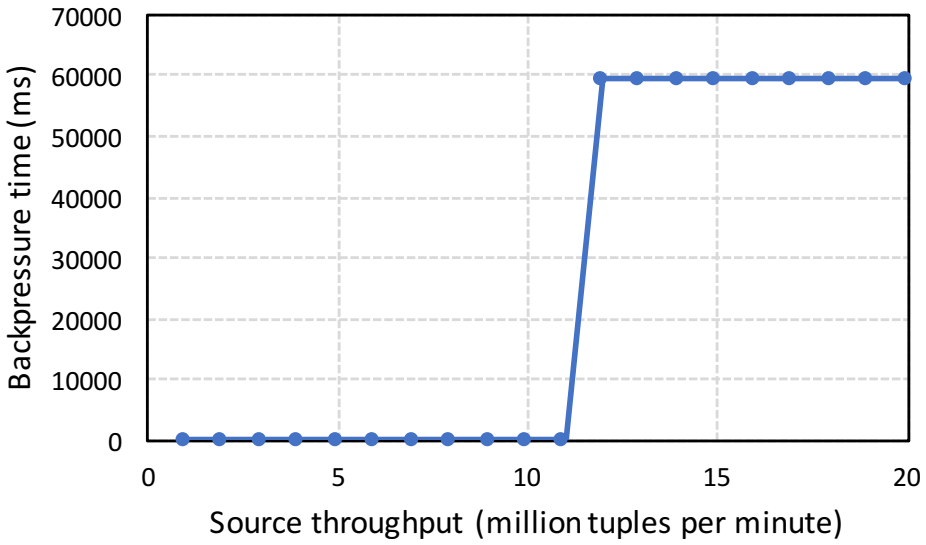


Figure 3.6: Instance backpressure time vs. topology input rate.

the test dataset is unbiased fortunately, thus we use Equation 3.9 for the sink bolt.

Now we have all the three component models on the critical path, and we can predict the critical path throughput by applying Equation 3.12. We choose the parallelisms in Figure 2.1. The predicted sink bolt's output rate is shown in Figure 3.10. Meanwhile, we deployed a topology with the same parallelism in our data center, and measured its sink bolt output rate, shown in Figure 3.10. The figure shows the observation matches the prediction with an error of $(139 - 135)/139 = 2.8\%$.

3.5.5 Use Case: Predict CPU load

Input rate significantly impacts an instance's resource usage such as CPU and memory. The tasks assigned to a job's instances can be categorized as CPU-intensive and memory-intensive tasks, whose CPU and memory load can be predicted.

Two factors are worth considering while performing our micro-benchmarks: 1) The saturation state i.e., whether a component triggers backpressure. When the component triggers backpressure, its CPU or memory load is supposed to be at the maximum possible level as the processing throughput of its instances also reaches their maximum points. 2) The resource limits of containers that run the instances (especially in terms of memory). Instances may exceed the container memory limit when their input throughput rises to high levels, which is rare in a well-tuned production job but can still happen.

In this section, we choose the CPU load of instances as an example. We observed that the CPU

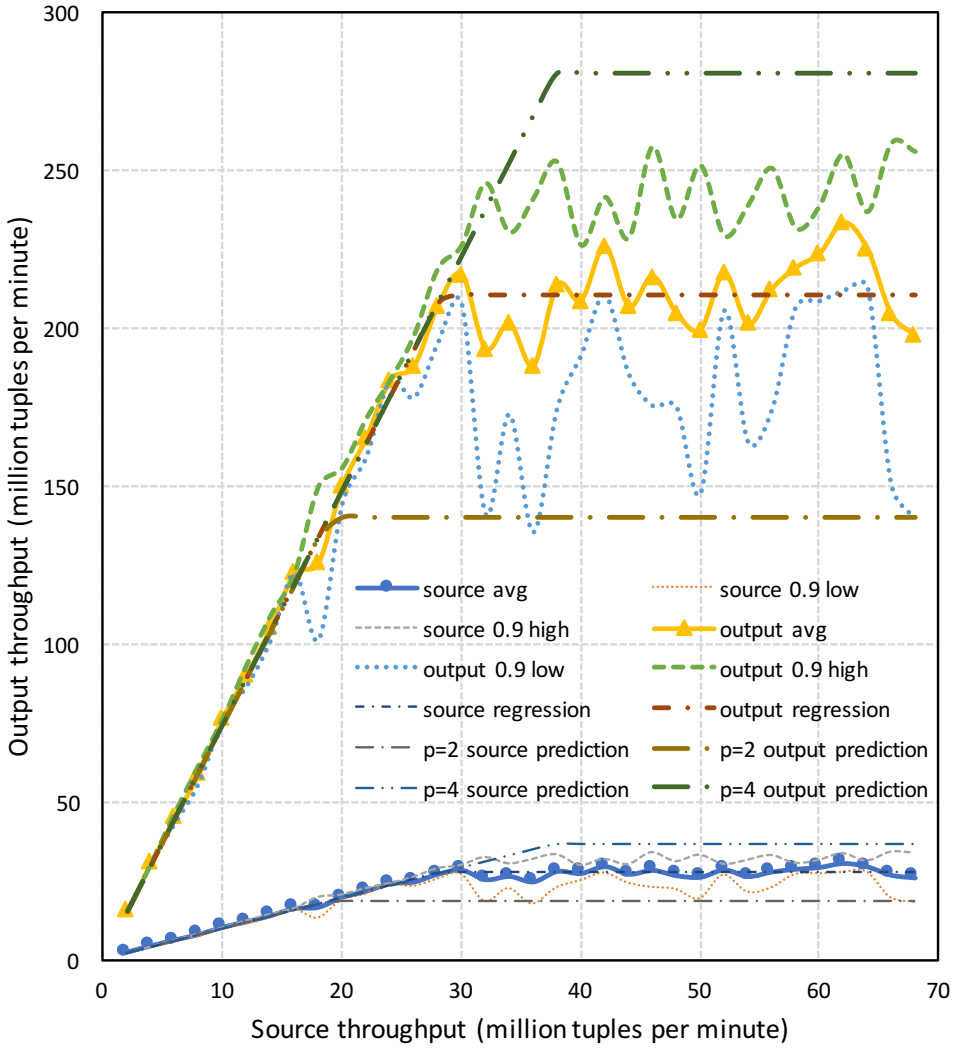


Figure 3.7: Component throughput measurements of the Splitter bolt for parallelism 3 and predictions for parallelism 2 and 4.

usage is linear to the output throughput per instance. Once we have the observation of several data points of <CPU load, output rates, source rates>, we can prepare two intermediate results:

- We can depict the throughput model <CPU load, output rates, source rates>, as we did in the previous evaluations.
- We can then use the model <CPU load, output rates, source rates> to calculate the linear ratio or the slope $\psi = \frac{CPU\ load}{output\ throughput}$.

Given the target input throughput, we use our model <CPU load, output rates, source rates>, to find the estimated output rates. We then amplify the output rates by ψ to get the CPU load

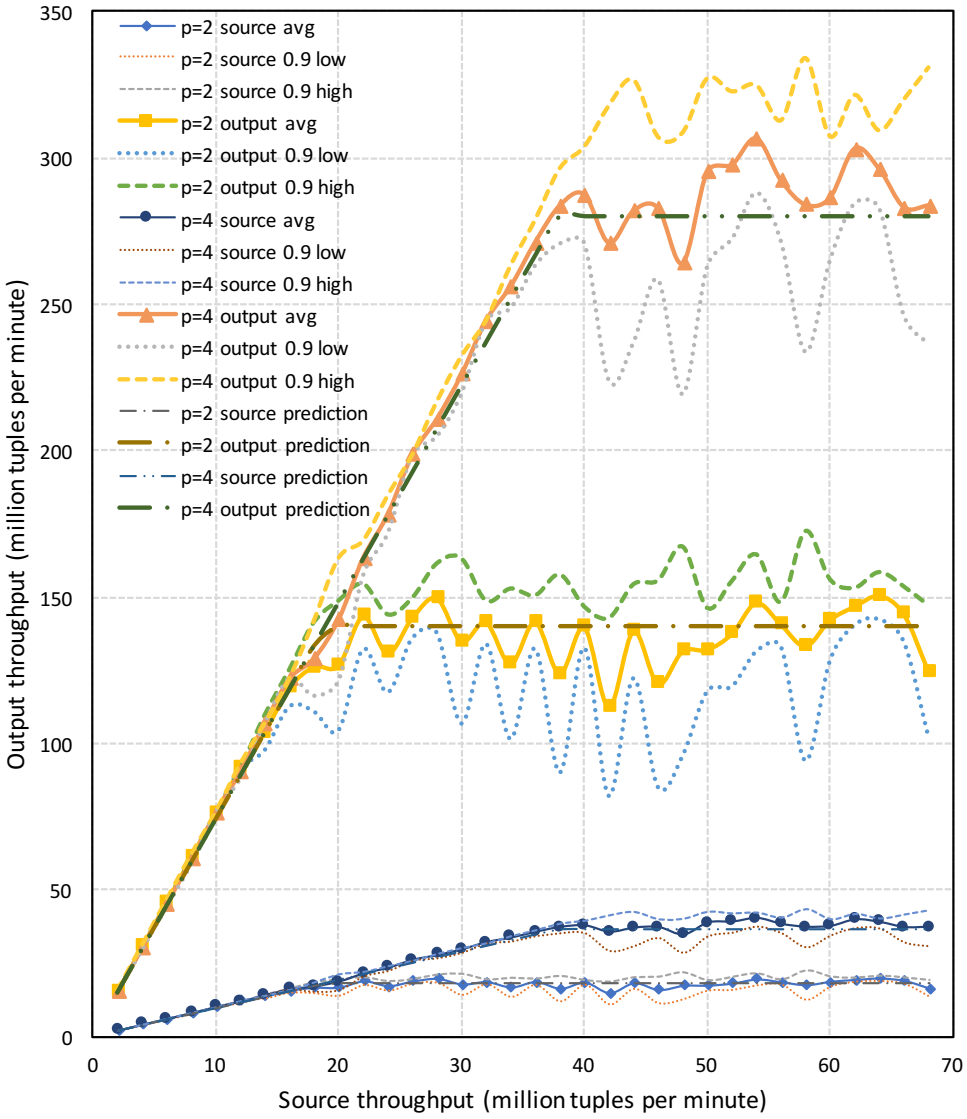


Figure 3.8: Validation on throughput prediction for parallelisms 2 and 4 for the Splitter bolt.

estimation.

We set the parallelism level of the Splitter bolt to 3 and observe the CPU load of its instances in Figure 3.11. The CPU load is collected from the Heron JVM native metric '__jvm-process-cpu-load', which presents the "recent cpu usage" for the Heron instance's JVM process. The predicted CPU load regression line are shown in the same figure as dashed lines for parallelisms 2 and 4 for the Splitter bolt.

Additionally, we configured the source throughput to the corresponding throughput and measured its CPU load for parallelisms 2 and 4 for the Splitter bolt. Figure 3.12 shows the measured CPU load vs. the predicted values. The prediction error is $(2.399 - 2.284)/2.399 = 4.8\%$ for a parallelism

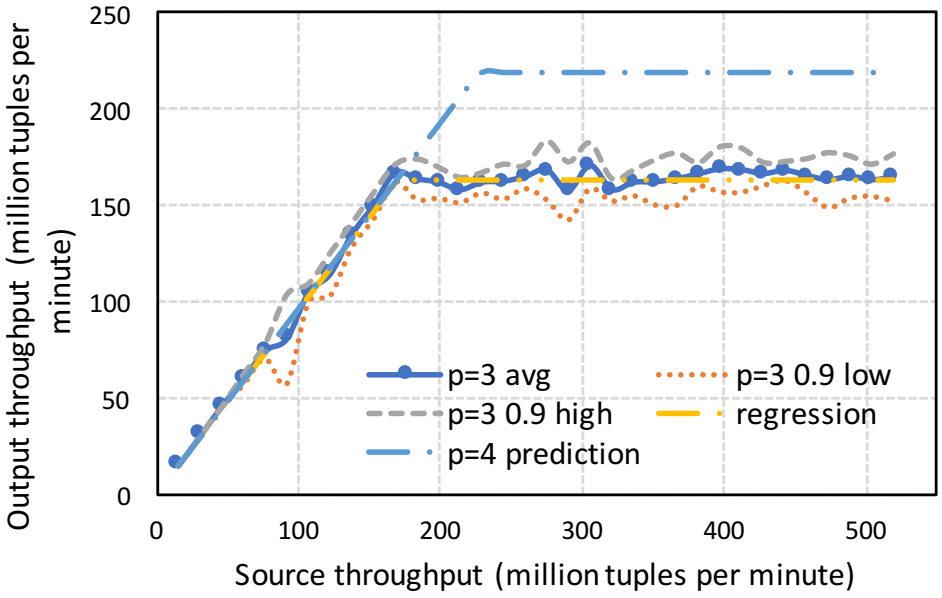


Figure 3.9: Component Counter throughput: observation and prediction.

of 2 and $(4.568 - 4.435)/4.435 = 3\%$ for a parallelism of 4, which is higher than the throughput prediction error. This is because error has accumulated for the chained prediction steps.

3.6 RELATED WORK

3.6.1 Performance Prediction

Traffic prediction is used for performance improvements in several areas. For instance, the ability to predict video traffic in video streaming can significantly improve the effectiveness of numerous management tasks, including dynamic bandwidth allocation and congestion control. Authors of [118] use a neural network-based approach for video traffic prediction and show that the prediction performance and robustness of neural network predictors can be significantly improved through multiresolution learning.

Similarly, several predictors exist in the area of communication networks such as ARIMA, FARIMA, ANN and wavelet-based predictors [74]. Such prediction methods are used for efficient bandwidth allocation (e.g., in [120]) to facilitate statistical multiplexing among the local network traffic.

However, the work done on traffic prediction in DSPSs is limited. For instance, authors of [166] perform traffic monitoring and re-compute scheduling plans in an online manner to redistribute

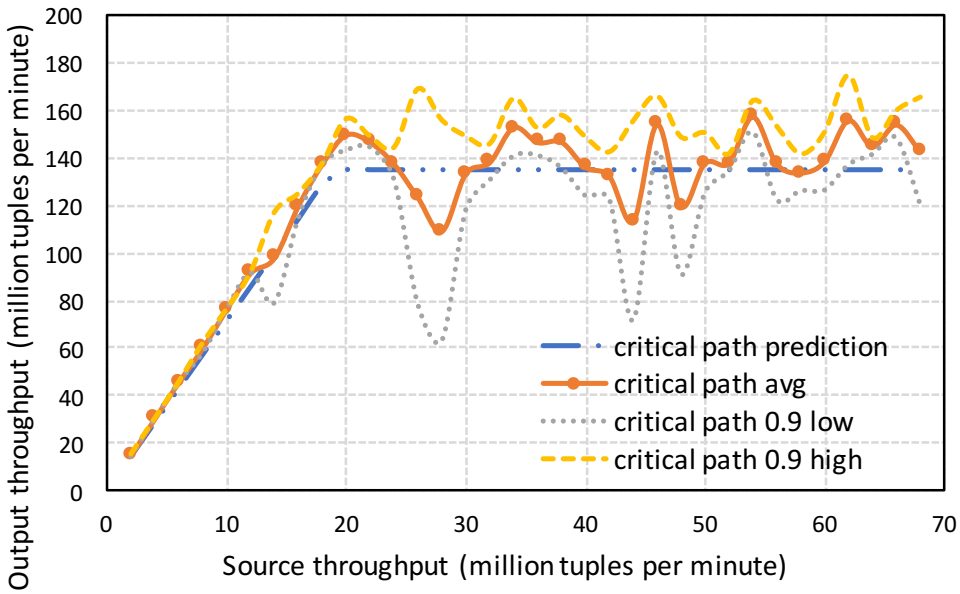


Figure 3.10: Topology predicted throughput and measured throughput.

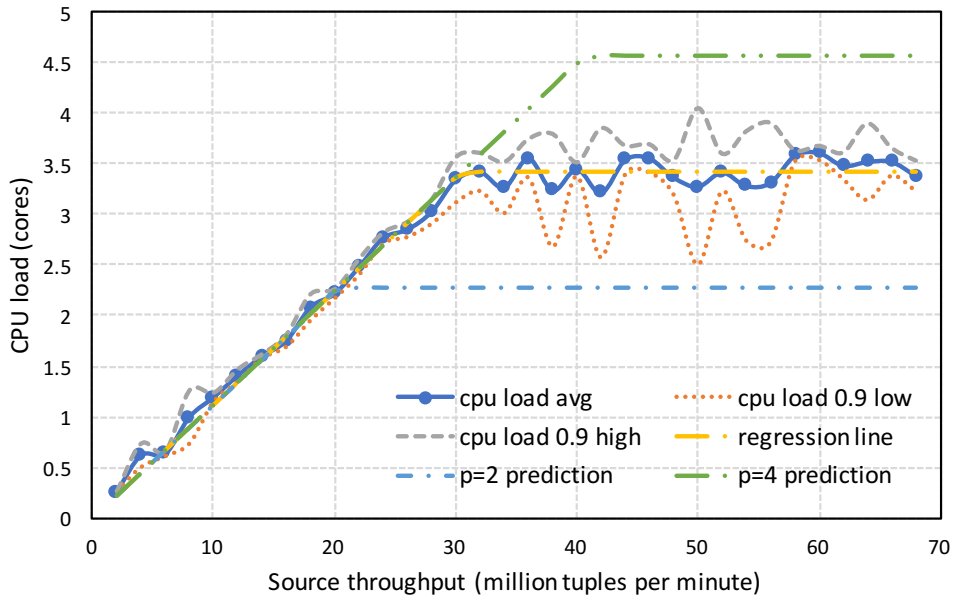


Figure 3.11: CPU load observation and prediction.

Storm job workers to minimize internode traffic.

Authors of [117] proposed a predictive scheduling framework to enable fast, distributed stream processing, which features topology-aware modeling for performance prediction and predictive scheduling. They presented a topology-aware method to accurately predict the average tuple

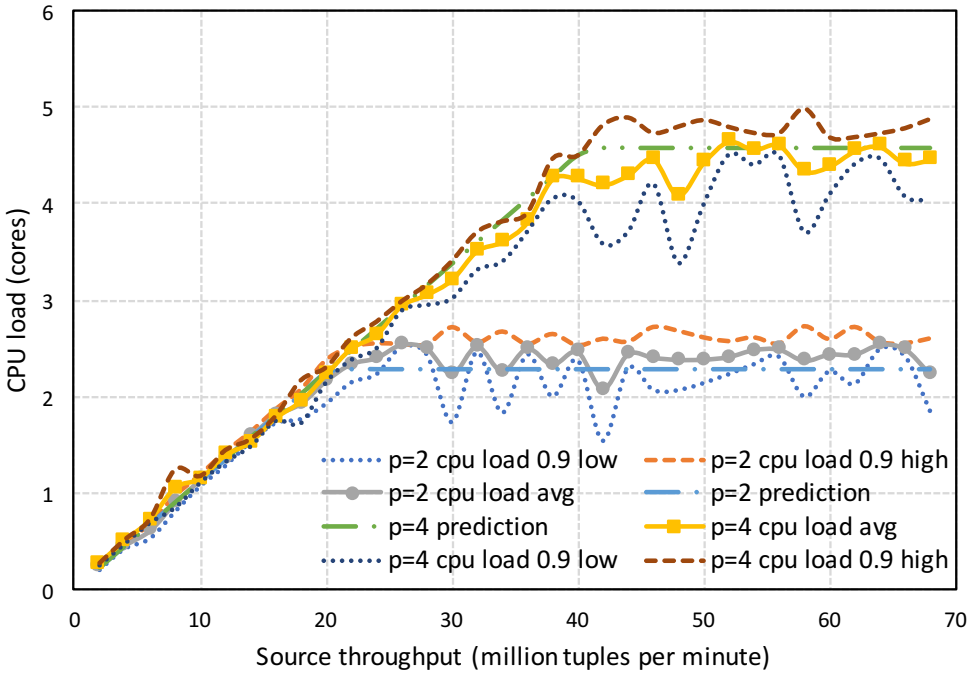


Figure 3.12: Validation on CPU load prediction.

processing time of an application for a given scheduling solution, according to the topology of the application graph and runtime statistics. They then present an effective algorithm to assign threads to machines under the guidance of prediction results.

3.6.2 Resource Management in Stream Processing

Job scheduling in DSPSs is a thoroughly investigated problem. Borealis [46], a seminal DSPS, proposed a Quality of Service (QoS) model that allows every message or tuple in the system to be supplemented with a vector of metrics which included performance-related properties. Borealis would inspect the metrics per message to calculate if the job's QoS requirements are being met. To ensure that the QoS guarantees are met, Borealis would balance load to use slack resources for overloaded operators. STREAM [126] is a DSPS that copes with a high data rate by providing approximate answers when resources are limited.

Authors of [136] focus on optimal operator placement in a network to improve network utilization, reduce job latency and enable dynamic optimization. Authors of [79] treat stream processing jobs as queries that arrive at real-time and must be scheduled subsequently on a shared cluster. They assign operators of the jobs to free workers in the cluster with minimum graph partitioning cost (in terms of network usage) to keep the system stable. However, given that the network is rarely a

bottleneck in today's high-performance data center environments, operator placement strategies do not necessarily reduce end-to-end latency. Furthermore, as operator parallelism can be very high, it is difficult to co-locate all parallel instances on a machine, making such algorithms in-applicable.

As mentioned earlier, a plethora of work [75, 104, 167] exists that gathers metrics from physically deployed jobs to find resource bottlenecks and scale jobs out in multiple rounds to improve performance. A relatively new topic for DSPSs is using performance prediction for proactively scaling and scheduling tasks; Caladrius takes a step in this direction.

3.7 CONCLUSION

In this chapter, we described a novel system developed at Twitter called Caladrius, that models performance for distributed stream processing systems and is currently integrated with Apache Heron. We presented Caladrius' system architecture, three models for predicting throughput of input data, and one use case for CPU load prediction. We illustrated the effectiveness of Caladrius by validating the accuracy of our 1) models and 2) Caladrius' prediction of throughput and CPU load when changing component parallelism.

Chapter 4: Meezan: Stream Processing as a Service

In spite of growing popularity of open-source stream processing systems, they remain inaccessible to a large number of users. A major obstacle in adoption is the overhead of configuring stream processing jobs to meet performance goals in heterogeneous environments, such as public cloud platforms. These platforms offer a variety of VMs that have different hardware specifications and prices. Despite the flexibility the many choices offer, an inexperienced developer may not be fully familiar with the impact and intricacies of difference choices on job performance [49, 158]. This makes the selection of a combination that fits both budget and performance goals a challenging task.

We propose Meezan, a system that allows users to provide their stream processing jobs as input and analyzes the jobs' performance empirically, to tune them for high throughput. As output, it provides users with a range of possible throughput guarantees for the jobs, where the most performant choices are associated with the highest resource usage and cost, and vice versa. Given this range, users are free to choose the cost-performance combination that works well for their application context and budget. Then, Meezan goes onto configure and deploy their jobs, providing a seamless stream-processing-as-a service experience.

4.1 INTRODUCTION

Stream processing is widely used to process the enormous amounts of continuously-produced data pouring into organizations today. High variability across use cases has led to the creation of many open-source stream processing systems and commercial offerings that have gained massive popularity in a short time e.g. Apache Heron [112], Apache Flink [62] and Amazon Kinesis [2]. To provide examples of their use cases, Heron runs stream processing jobs at Twitter scale to support functions such as finding patterns in user tweets and counting the top trending hashtags over time. Similarly, Flink is used at Uber to perform functions that are essential to revenue such as calculating surge pricing [42]. Kinesis is a commercial offering from Amazon that allows users to collect, process, and analyze real-time, streaming data and is used by companies such as Zillow and Netflix [2].

However, these systems still present a barrier to entry for non-expert users, who may range from students in a lab with very limited resources to experienced software engineers who work alongside the developers of these systems (e.g. developers of Heron at Twitter), but do not have experience with running the framework themselves and thus have difficulty optimizing jobs. As stream processing jobs are often-times business-critical, they are generally associated with throughput and/or latency service level objectives (SLOs). This makes tuning jobs for performance all the more important.

Performance tuning of jobs becomes increasingly difficult when the underlying stream processing

engine adds components to the jobs that are transparent to the user but are necessary for the job to execute well. For example, Apache Heron adds a "stream manager" to every container that runs Heron jobs, that provides communication between the job's operators (details described previously in Section 3.2). As we show later in Section 4.3.3, it is possible for such components to bottleneck, which makes performance tuning hard for users who are unaware that such components even exist.

Such problems have led to the creation of system-specific trouble-shooting guides [27] that offer advice on how to deploy jobs correctly and tune them to achieve optimal performance. However, these guides are usually not modified as the system continues to evolve. Furthermore, users can require very specific guidance for tuning parameters that are pertinent to only their jobs. Although commercial offerings such as Kinesis [2] automatically deploy jobs and scale them on a user's request, they still leave job tuning and configuration for optimal performance to the user.

An additional deployment challenge for inexpert users involves finding a resource allocation for their job that allows it to run without bottlenecking on resources, thus ensuring that it will meet its performance goals. This becomes more challenging for novice users who do not have access to large data center environments and often use cloud platforms, such as Amazon AWS and Microsoft Azure for deployment. Selecting a set of VMs that allows them to meet their performance goals is no easy feat as today there are more 280 types to choose from on AWS alone! [11]. All of these VMs differ from each other in terms of their hardware specifications. Expecting novices to understand the impact of each of these VM specifications on their job's performance is unreasonable as often, such a task cannot be accomplished off-hand by expert researchers on the subject [140].

Finally and equally importantly, the job deployment must also fit within the user's allocated budget. It is possible that a budget constraint may force a user to make sub-optimal choices (such as selecting several cheaper, small VMs instead of a few larger, more expensive VMs that will cost less in aggregate, but may result in over-allocation).

We propose Meezan, a system that tunes jobs for high throughput performance, and removes this burden from the user, thus providing a seamless, stream processing as a service experience. Furthermore, once a user submits a job for a specific cloud platform, Meezan presents users with a spectrum of deployment options, ranging from the cheapest, least resource-consuming and least performant to the most expensive, highest resource-consuming and most performant. Meezan's goal is to provide users with the cheapest and smallest deployment option in each performance range so that users do not have to pay for resources that cannot improve job performance. This allows users to choose a deployment option that is most suitable for their performance requirements and budget.

Given a user's stream processing job and choice of cloud platform as input, Meezan creates several deployment options of the job for the user to choose from as output, where each of the deployment options has an associated price and throughput guarantee. Meezan creates deployment options with varying throughput guarantees as it varies how much data each option ingests. For

instance, in production settings, stream processing jobs usually ingest data from an external source such as Apache Kafka [6], which divides and stores incoming data into several partitions. In this case, Meezan's smallest job deployment option would have a single job operator that reads input from a single partition [111]. In production settings, a Kafka cluster can support up to 200K partitions [53]. This means the largest deployment option may be required to read and process data from all partitions at once, and must thus provide 200K$\times$ the throughput. Therefore, the largest deployment option Meezan creates will scale the user's job to support this required throughput and will create a deployment plan for it that be priced in accordance with the resources used.

Thus, Meezan allows job developers to focus only on the functionality of their jobs and forms a layer of abstraction between the developer and the cloud provider. Once their job is fully written and packaged, e.g., in the form of a jar for jobs written in Java, it is submitted to Meezan, which then performs the following:

- Meezan creates an estimate of the throughput of every processing instance in the job. This allows it to estimate the number of instances required to keep up with the incoming job workload.
- It estimates the resources required per instance to ensure that no instance is bottlenecked.
- Then, Meezan creates several deployment options for the job: these options consists of a mapping of job instances to VMs run by a cloud provider that execute the job, and their associated cost. Meezan's goal per option is to ensure that the cost of the VMs used to run the job is minimized while meeting a given performance target. Then, the user is allowed to choose one of the deployment options that fits well within their job context and budget.
- Given the user's selection, Meezan deploys the job onto the cloud platform and ensures that the job is not bottlenecked by the network or the job's own hidden structural components.

In Figure 4.1, we present Meezan's cost-performance spectrum for three different jobs (described in detail in Section 4.4). We note that that as the sizes of different jobs increase, their cost increases differently; this is due to the differences in the nature of their business logic and workloads. Irrespective of job type, we note that the cost of the deployments increase linearly as job throughput increases. We compare Meezan's performance with that of RCRR, a version of the default Heron scheduler, that we have modified to support job scheduling for a commercial cloud platform with heterogeneous VMs. Meezan is able reduce cost of deployment while sustaining the same throughput by up to an order of magnitude (2.18$\times$ for the Yahoo! Advertising Benchmark, 33.4$\times$ for the LRB Toll Calculation Job and 28.62$\times$ for the WordCount topology). Meezan is able to accomplish this goal because of two reasons: 1) it scales out the job linearly to increase throughput,

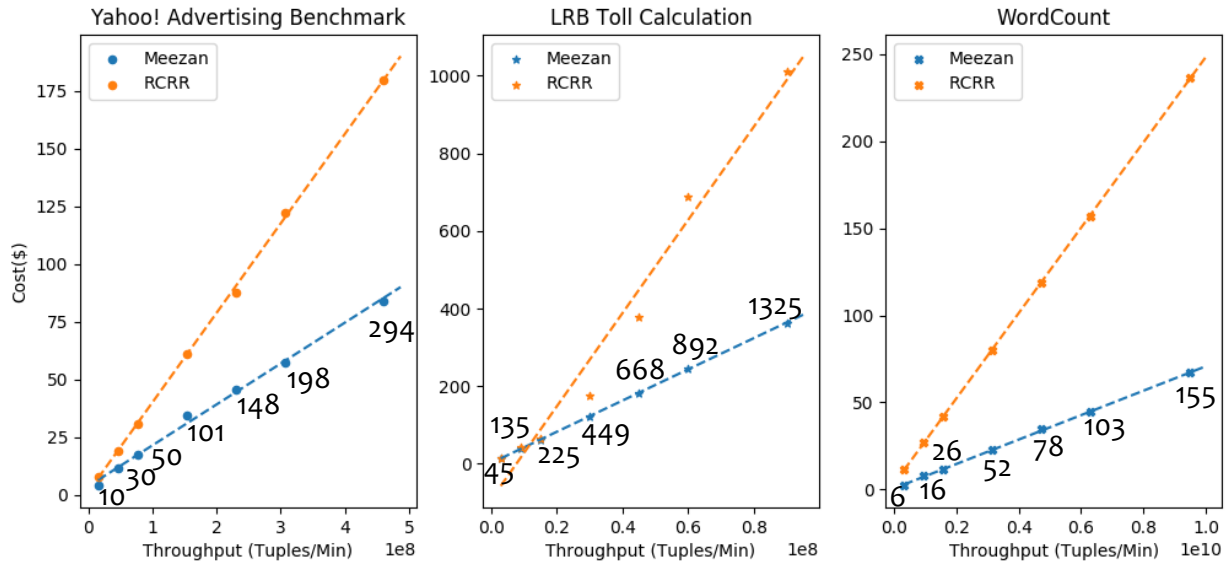


Figure 4.1: Meezan's cost-performance spectrum for three jobs, where each point shows the cost of a deployment (y-axis) that provides a given throughput throughput (x-axis). Dotted lines are linear lines of best fit. Numbers on markers indicate job size in terms of number of operators in job. RCRR is a version of the default Heron scheduler, that we have modified to support job scheduling for a commercial cloud platform with heterogeneous VMs. Details are in Section 4.3.4).

as it identifies and remove bottlenecks in the job's structure correctly and 2) it packs jobs efficiently into VMs, such that both deployment cost and resource fragmentation are minimized. In the next section, we explore details of Meezan's design and the insights behind them.

4.2 BACKGROUND

We refer readers to Section 3.2 which covers background relevant to stream processing systems in general, and Apache Heron [112] in particular, on which we have built Meezan.

4.3 SYSTEM DESIGN

In this section, we describe how Meezan creates a spectrum of performance goals for the user to choose from. Concretely, we present the user with a variety of throughput goals. These throughput goals are expressed in terms of *input throughput* or how much data the job ingests per unit time. Each throughput goal is associated with a different job DAG: intuitively, the lower the throughput, the less data that is ingested by the job for processing, which leads to a less parallelized job DAG. We provide a brief overview of our design and then dive into details of how Meezan creates a

packing plan for the job per throughput goal, such that it packs the job DAG into a set of VMs provisioned by a cloud provider such as AWS and Azure, that minimize the user's cost.

4.3.1 Overview

Given a user's stream processing job, Meezan must first identify the resource requirements and required parallelism of each of the job's operators, so that the job does not bottleneck or in other words, its output throughput rate is equal to its input throughput. In order to derive the job's resource requirements and operator parallelism levels, Meezan profiles the job at small scale, where it has a single operator that reads data from an output source.

Next, it generates several deployment options for the user to choose from. These deployment options are created by varying the number of operators that ingest input from external sources. Such operators are referred to as spouts. The greater the number of spouts, the greater the input the job is expected to process. Meezan scales up the job to sustain the higher input rate, given that it already has information on the resource requirements and operator parallelism levels of the job so that it is able to sustain input from a single spout. We provide details of these steps in Section 4.3.2.

Finally, Meezan packs each of the derived job structures for each of the deployment options onto VMs available on the cloud platform of the user's choice. It uses two key insights while performing this packing (detailed in Section 4.3.3): 1) VM prices on popular cloud platforms such as AWS and Azure scale linearly in proportion to VM size. This incentivizes Meezan to select large VMs so that many operators can be packed together, as picking several smaller VMs may lead to equal cost but would increase communication over the network. 2) In addition to the generally considered resources of CPU, memory, disk, and network, we must also consider the stream manager in every container as a resource. The stream manager is responsible for transferring data between all operators in Heron. If too many operators are packed together into a VM, the stream manager can easily bottleneck, slowing down the operators. Therefore, we must first derive the throughput a stream manager can tolerate per VM type and pack operators into VMs such that the stream manager is not bottlenecked.

We use these insights to derive Meezan's packing policy (described in Section 4.3.4), which aims to minimize cost of deployment, reduce resource fragmentation and sustain the job's input throughput.

4.3.2 Providing a Cost-Performance Spectrum

We first describe how Meezan fine-tunes the resource allocation per job operator and its required parallelism level, before it scales up the job's operators to provide users with a range of performance

options.

Resource Allocation & Traffic Rates Per Instance: Given a user's job, Meezan profiles it with the minimum number of spouts to discover the job's resource requirements and potential bottlenecks.

Jobs can have severals different spout types that can be reading data from several different sources. If there is only one such spout type, we profile a job with only one instance of it. If there are N number of such spout types, we profile the job where there is only a single instance of each spout type.

Once the least number of spouts are determined, Meezan then deploys and profiles the job to discover the amount of resources job operators require. Usually, user-defined functions in instances e.g., filters and transformations show a linear trend, where an increasing amount of resources (in terms of CPU cores, GBs of memory) leads to an increase in instance throughput for a given input rate. However, if the input rate is increased beyond a saturation point, the instance throughput plateaus even though there is no resource bottleneck. This is the maximum possible processing rate of the instance: to keep up with a growing input rate, the instance's parallelism must be increased.

Meezan's goal is to find this input throughput rate, the associated output rate and the resource allocation for each instance. It does so by profiling the job in rounds. In each round, it either changes the resource allocation of each operator or its parallelism level. Meezan determines that an operator's resource allocation should increase if its utilization with respect to that resource is over a bottleneck threshold (=90% in our experiments). Therefore, in the next round, it doubles that specific resource for the instance.

This can lead to two outcomes: 1) the instance's utilization decreases or drops below the bottleneck threshold or 2) its utilization with respect to that resource does not change significantly. In the former case, Meezan interprets that increasing a specific resource removed a bottleneck for the instance, and continues profiling with the increased resource allocation. In the latter case, it recognizes that the instance is saturated, and in order to improve the instance's throughput, increasing resources will not help. Therefore, it reverts to the resource allocation used in the previous round, and doubles the instances parallelism level in the next round to improves the operator's throughput. Meezan continues this until the utilization of all instances of all operators is below the bottleneck threshold.

We show an example of how this works in Figure 4.2. In the figure, Meezan profiles the LRB Toll Calculation Topology (described in Section 4.4. The job has 5 operators, other than the spout. The x-axis shows the rounds Meezan's profiling process employs. Each of the subplots along the y-axis shows the average CPU utilization of all instances of each of the different operators. We notice that in round one, CPU utilization for each operator is greater than 90%. However, we would like to ensure that each operator's utilization lowers to an acceptable range. Thus, in the second

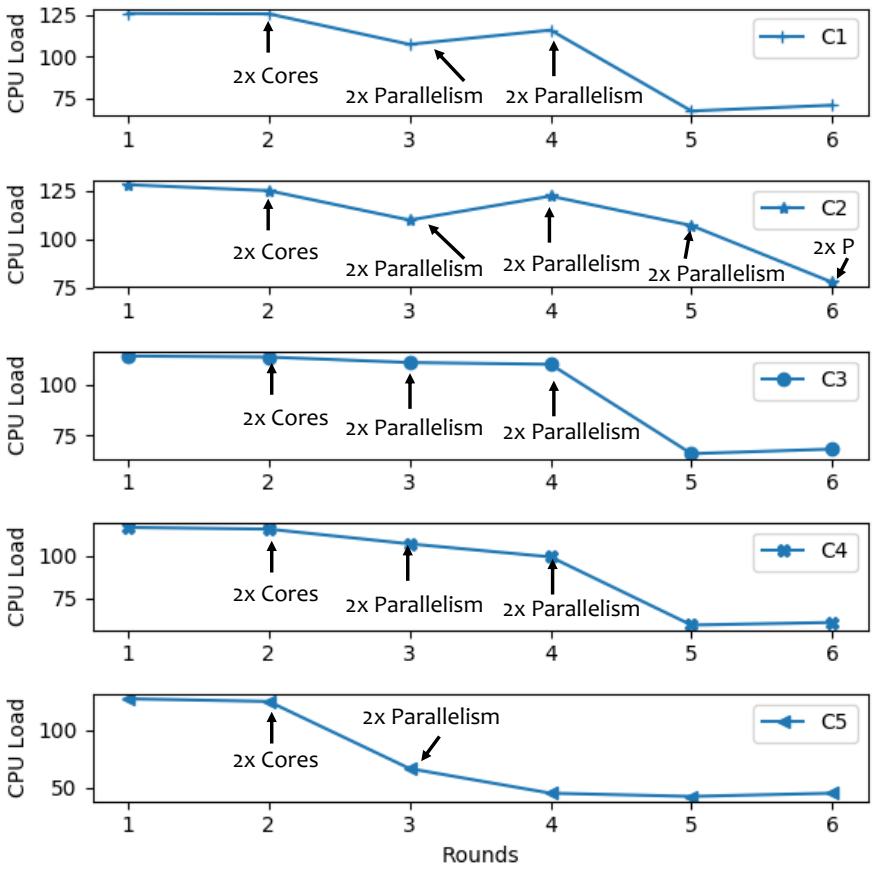


Figure 4.2: Meezan's profiling process for finding job resource allocation and parallelism for the LRB Toll Calculation Topology, which has 5 components (C1-C5) other than the spout. Every subplot shows how Meezan configures an individual component. First, it increases the operator's resource allocation. As that does not lead to a decrease in CPU utilization, it doubles parallelism of the component until its CPU utilization falls belore 90%.

round, Meezan doubles each operator's cores. However, we note from the utilization in round 2 that this does not have an effect on utilization. Therefore, in the next rounds, Meezan adopts a strategy of doubling the parallelism of each operator until the average per parallel instance CPU utilization of each operator falls below 90%. This job's operators are CPU-bound and therefore, we only see changes in CPU cores and parallelism, however, Meezan is able to do the same for memory as well.

When Meezan profiles jobs, it provides them with input from the job's actual sources for a configurable period of time (=5 minutes as default), to understand the job's resource requirements. However, it is possible that during job execution, different input may trigger different parts of computation, causing the job's resource requirements to change. In order to deal with this problem, jobs could be re-profiled with new data, much like in other approaches [99] or scaled out at runtime [75] [104].

The operator resource allocations, data rates and parallelism levels derived in the last round of

this profiling step are used for modeling the job in all next steps.

Parallelism: the Cost-Performance Spectrum: We can vary the amount of input that can be processed by the job by varying the parallelism of the spouts. The minimum input throughput that a job can provide is when the parallelism of each spout is one. The maximum possible throughput is equal to the external source's input rate i.e. how quickly data arrives in the source system such as Kafka and is stored in different partitions.

In order to accommodate this rate, the topology should have the maximum possible number of spouts. If the user is aware of this rate, Meezan can calculate the optimal number of spouts given the equation below. However, if the user is unaware of the maximum input rate the job is expected to have, the user can ask Meezan to offer them a certain number of options within a throughput range.

Meezan's real challenge is to provide a spectrum of throughput goals between the two aforementioned performance extremes. It does so by varying the number of spouts between $[1, max\ spout\ instances]$, where $max\ spout\ instances$ can ingest input as fast as it is arriving in the source. By varying the number of spouts in this way, we vary the input the topology is able to ingest. If the number of spouts is $< max\ spout\ instances$, it is clear that input will queue in the source pub-sub system and that the user is aware of this tradeoff.

Given every possible configuration of spout parallelism, the topology's downstream instances must be scaled up to accommodate the varying input rate.

In order to find the optimal number of instances required per operator, we move stage by stage downstream through the topology. For every operator, its optimal parallelism can be determined by:

$$Parallelism = \frac{\sum_{i=0}^{n=\#\ parents} Output\ Rate_i}{Processing\ Rate\ per Instance} \tag{4.1}$$

This formulation works well when an operator receives data from its upstream parent through a shuffle-grouped connection, where the parent operator's output is equally distributed among the downstream operator's parallel instances. In the case of a connection that has fields- or a user-defined-grouping, this equal distribution may not necessarily be true. In order to handle this case, Meezan makes use of the iterative long-running nature of jobs. It begins job deployment with the parallelism given by the formulation in equation 4.1. If data skew is discovered in a connection, it can increase the parallelism of instances with the highest incoming rates.

We remind the reader that the maximum output rate of each instance was already determined alongside the optimal resource allocation in the previous section through profiling. Although other analytical methods of finding the optimal parallelism of operators exist e.g. based on queueing theory [76], they have been shown to not work well in practice and instead are replaceable with the simple approximation above [102].

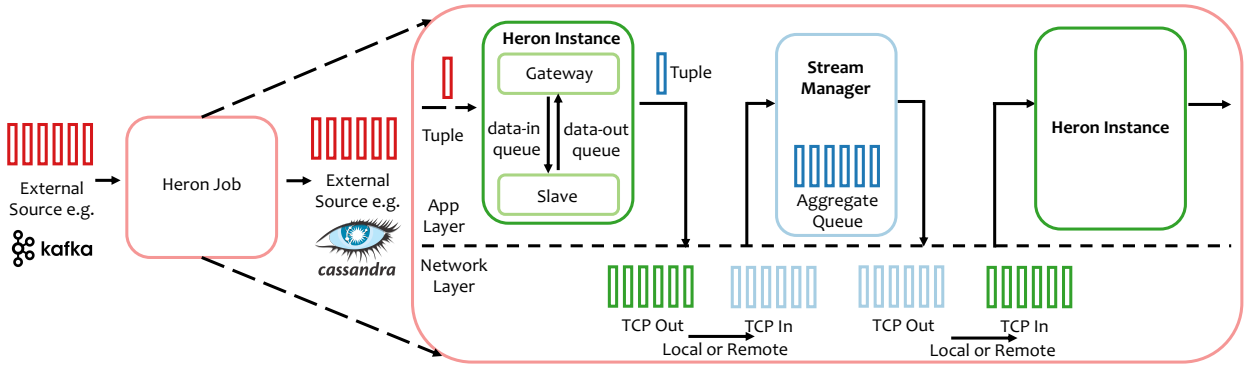


Figure 4.3: A tuple's path through interconnected Heron instances.

By the end of this process, the optimal parallelism of each operator in the topology is determined, per input rate, supplied by varying the number of spouts. As Meezan is already aware of the optimal amount of resources required per instance, it creates a scheduling plan for the topology for every every configuration of spout parallelism (Section 4.3.4). As each of these plans is optimized to have no resource bottlenecks, each of them provides the same the end-to-end latency a tuple experiences from the time it enters a spout to the time when it produces a tuple at the sink.

Now that Meezan has determined the number of resources every individual instance requires to run at maximum capacity, it must map each of the instances to containers that are run on physical machines and create a *container* or *packing plan*.

In the next section, we describe the set of insights that are important to keep in mind while building the packing plan.

4.3.3 Insights & Goals

Creating a cost-performance tradeoff involves assessing all the possible throughput rates that can be achieved for a job, correctly configuring the job DAG for each targeted rate, and packing each configuration into containers to minimize the user's cost. To do so, it is important to keep the following factors in mind.

Stream Managers as Potential Bottlenecks Packing Heron instances into containers is akin to packing tasks (balls) onto available resources (bins). Although bin-packing problems are well-researched, Meezan's bin-packing solution must also take into the account the stream manager that is placed in every container, and is responsible for routing tuples between instances in the container or across containers.

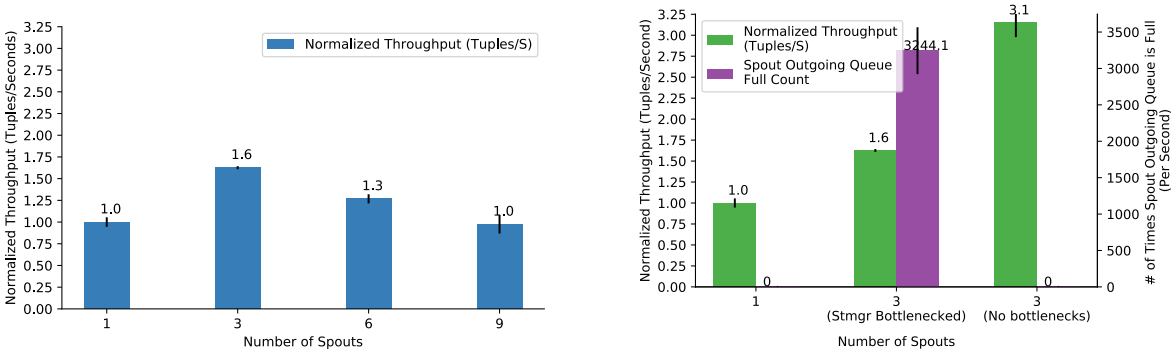
We use Fig. 4.3 to show a tuple's path through a Heron job, and describe the series of events that occur when a stream manager becomes a bottleneck. Spout are a specific type of Heron instances

that pull in tuples from external sources such as Kafka queues, that store fast-arriving data. Each Heron instance is a Java process that contains two sets of threads: the slaves and the gateway threads. The gateway thread is responsible for pulling incoming data from the stream manager, and sending outgoing processed tuples to the stream manager. The gateway and slave threads share two queues between them: the incoming and outgoing data queues. Whenever new tuples arrive from the stream manager, the gateway thread deserializes them and places them on the incoming data queue. The slave threads are the task executors: they read input tuples from the incoming queue and apply user-defined functions to them. If new tuples are produced as a result of processing, the slave threads place them on the outgoing queue. The gateway thread then batches several tuples on the outgoing queue, and serializes them. It also holds a TCP client that sends data to a server on the stream manager.

The stream manager essentially behaves like a switch: it runs two servers which receive data from local instances and remote stream managers respectively. The stream manager inspects a received message from instances to determine whether they need to be forwarded to a local instance or to a remote stream manager that will then direct the packet to its local instance. Packets received from remote stream managers can only be forwarded to local instances. Once the destination is determined, the packet is placed in an outgoing queue for the destination task. Once all queues cumulatively reach a certain threshold size, all packets are forwarded to their destinations and the queues are flushed. Due to this switch-like behavior, the stream manager can easily become CPU-bound.

Once the stream manager becomes CPU-bound, it cannot read data from its TCP connections quickly enough. Thus, the TCP connection must keep the packets buffered on the client (instance) side, until they can be successfully pushed through to the stream manager. Once the TCP connection buffers start to fill up, the gateway thread is unable to send out data from the instance's outgoing queue. In order to prevent the outgoing queue at the instance from becoming unbounded, the slave thread stops running the user-defined code that processes incoming data in bolts, and pulls in data from external sources in spouts. As the gateway thread continuously tries to push data out of the outgoing queue, the slave thread is usually not blocked for long and restarts processing again.

Figure 4.4a shows what happens to a job's input rate when it's stream managers are bottlenecked. We run a simple stream processing job that emits simple strings as tuples from spouts that are passed down to downstream bolts that simply append to input strings. We deliberately make the stream manager a bottleneck in the job by adding all instances into a single container, where a single stream manager is responsible for all message passing between instances. As we increase the number of spouts from 1 to 9, we also increase the number of downstream bolts and increase the size of the container, to ensure that there are no other bottlenecks than the stream manager. We observe that as we increase the number of spouts, cumulative throughput of all spouts does not



(a) Aggregate spout throughput when stream managers are bottlenecked.

(b) Aggregate spout throughput increases in proportion to # of spouts, when stream managers are not bottlenecked.

Figure 4.4: Stream Manager Behavior

increase proportionally as we expect. This is especially problematic for Meezan, whose main goal is to allow users to vary input throughput by increasing the number of spouts.

Figure 4.4b compares the cumulative throughput of the same bottlenecked job with 3 spouts, with the throughput of a job that is distributed onto multiple containers. More precisely, we ascertain the maximum throughput a stream manager can reach at 100% CPU utilization. Then, we placed instances on containers such that no stream manager would receive packets from instances at a rate that exceeded this maximum throughput.

We also plot the count of the number of times the slave thread found the outgoing queue full per second before it could process more data (labelled as Spout Outgoing Queue Full Count). We can see in the case of the job that is not bottlenecked on stream managers, the outgoing queues on spouts are never full and that their mean cumulative throughput over time (normalized by the throughput of a single spout) is in proportion to the number of spouts[1].

As the stream manager can quickly become CPU-bound, Meezan determines a maximum throughput rate T the stream manager can tolerate, before its queues fill up. When placing instances in a container, Meezan must consider that the sum of both the rate of tuples arriving on the *incoming* and *outgoing* edges of each of the instances placed in a container should not exceed T, in order to ensure that the stream manager is not overloaded, as shown in Figure 4.5.

Figure 4.6 depicts Meezan's approach in determining T for a specific hardware type. Meezan profiles the stream manager in rounds. In the first round, it starts by packing a single spout with a downstream bolt onto a single container with one stream manager. If it finds that the spout's "Spout Outgoing Queue Full Count" is close to zero and if the stream manager's CPU utilization is low ($<$ 80%), it doubles the number of spouts running in the container in the next round. In Figure 4.6, this

[1]In fact, the job has a mean cumulative throughput of 3.1$\times$ the throughput of a job with a single spout whereas the reader might expect a normalized throughput of 3.0$\times$. This value has been ascertained by finding mean throughput for both bottlenecked and non-bottlenecked jobs over a period of time. Thus, the error of $\frac{0.1}{3.0}$ is empirical.

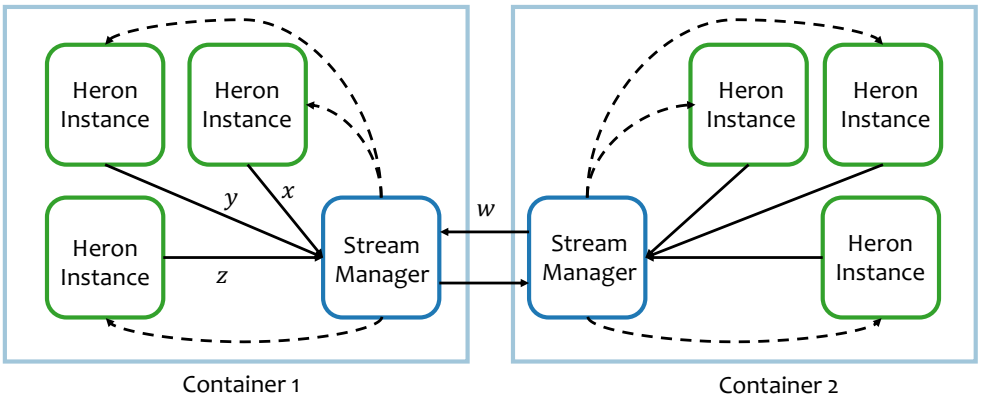


Figure 4.5: $w + x + y + z \leq T$ to prevent stream manager bottlenecks.

is how Meezan moves from 1 to 2 spouts.

At this point, the stream manager is still not bottlenecked, and so, Meezan doubles the number of spouts again (=4). At this point, it finds that the average "Spout Outgoing Queue Full Count" per spout exceeds 0, and the CPU load on the stream manager also exceeds 80%. Therefore, it performs a binary search between the current number of spouts (4), and the maximum number of spouts that led to low CPU load on the stream manager (2). Thus, it runs the next round with 3 spouts and finds that CPU load still exceeds 80% and the "Spout Outgoing Queue Full Count" exceeds 0. Since there is no further room to continue the binary search, it determines that the maximum throughput the stream manager can tolerate is the throughput that is associated with 2 spouts (or 45549.7 K tuples/Min) on this machine.

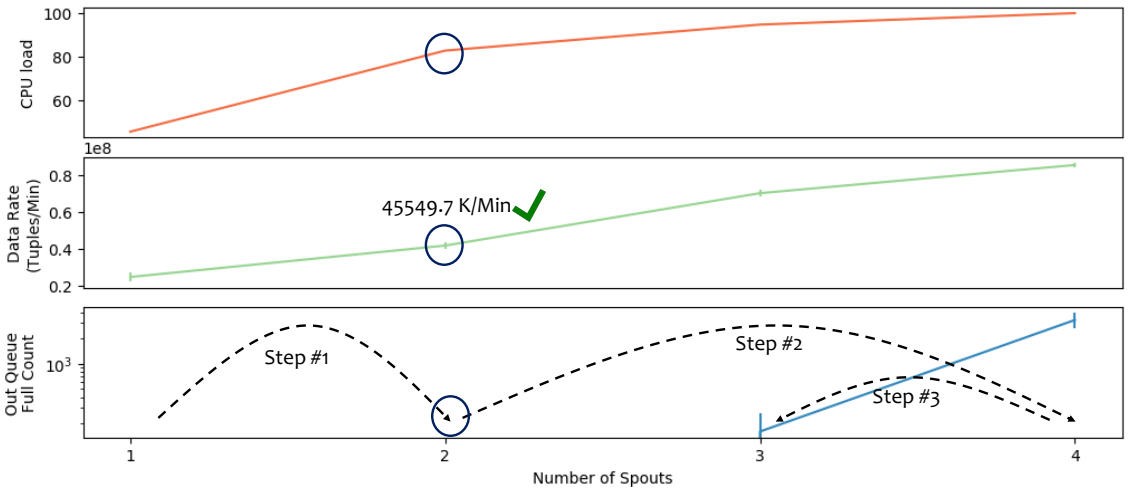
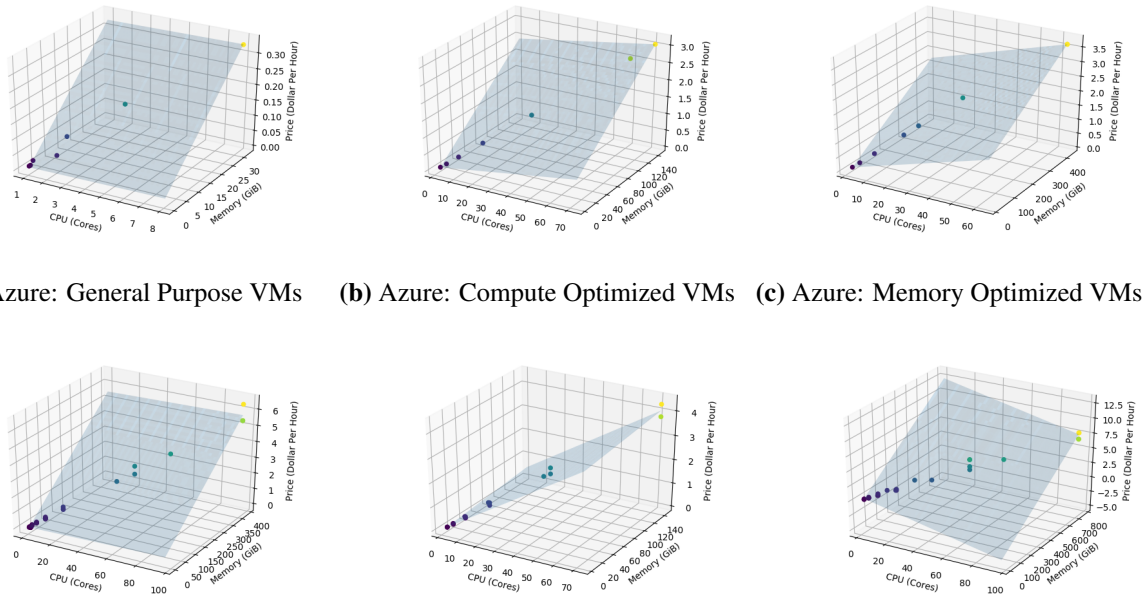


Figure 4.6: Meezan's exploration to determine the maximum sustainable throughput at the stream manager.



(a) Azure: General Purpose VMs **(b)** Azure: Compute Optimized VMs **(c)** Azure: Memory Optimized VMs

(d) EC2: General Purpose VMs **(e)** EC2: Compute Optimized VMs **(f)** EC2: Memory Optimized VMs

Figure 4.7: Figures a-c show a linear trend in cost of VMs, provided by Azure (US2), as CPU and memory allocations increase. Figures d-f do the same for VMs in EC2, Northern California. Although the dollar amount changes with region, the trend to remains the same.

Instance Locality Cross-container data transmission requires serialization by the stream manager. However, if data is to be passed to an instance in the same container, it can be passed in memory. In addition, colocating instances that directly communicate with each other has the additional benefit of utilizing less network resources. Therefore, Meezan is incentivized to keep two communicating instances in the same container. This has the added benefit of minimizing the number of stream managers in a tuple's path.

Reducing Fragmentation To reduce costs and resource fragmentation, Meezan must perform packing of instances onto containers such that each container is fully utilized. Containers that use up an even proportion of all resources (rather than those that consume a large amount of a single resource) cause less fragmentation on machines and lead to higher resource utilization.

VM Sizes and Pricing models Instead of packing containers onto VMs in a cloud offering such as Amazon EC2 or Azure, Meezan maps a single container onto a VM. This means that the problem of choosing a container size during bin-packing essentially becomes the problem of choosing the correct VM.

Choosing the correct VM size for a workload in order to reduce cost and maintain optimal perfor-

VM Type	Ratio $\frac{CPU(Cores)}{Memory(GiB)}$
Memory Intensive	0.13
General Purpose	0.25
Compute Intensive	0.53

Table 4.1: Resource Ratio in AWS and Azure offerings [1, 31]

mance is not an easy problem. Existing solutions employ complex machine learning algorithms to solve this problem for specific workloads [49]. However, after studying VM price on platforms such as Amazon EC2 [1] and Microsoft Azure [31], we can simplify the problem to a great extent.

In Figure 4.7a-c, we show how the costs of VMs provided by Microsoft Azure increase with increased resource allocations. These costs are shown for the US2 region and are representative of costs in other regions. In Figure 4.7d-f represent the same for Amazon EC2 instances in Northern California. We find that cost increases very linearly with increasing resources. In fact, we are able to successfully fit two-variable linear equations to each set of data points, with R^2 values between 0.814-0.999. This entails that there is no benefit of choosing several small VMs over few, large VMs or vice versa on any of today's popular platforms. Therefore, whenever a new bin has to be opened in Meezan's policy, it always chooses the largest possible bin in its category (based on instance type, details in Section 4.3.2). Intuitively, a larger bin means that there is more room to pack instances and try to keep instances in the same path together in the same bin to maintain locality.

4.3.4 Meezan's Packing Algorithm

Meezan's goal is to ensure that there are no resource or communication bottlenecks when a job is deployed, and to minimize the cost of deployment. Minimizing communication across a deployed DAG is essentially a cut problem. However, solving a cut problem limits which instances can be placed together. On the other hand, packing a job into the cheapest, fewest containers ignores edges crossing over container boundaries which increase communication cost. Therefore, we propose a heuristic that attempts to balance both requirements.

Choosing VMs Commercial cloud offerings allow users to use many kinds of VMs, including general purpose, compute-intensive, memory-intensive, storage-intensive, GPU-intensive or FPGA-intensive ones [1, 31]. As stream processing jobs are generally compute or memory intensive, we focus on the first three VM types. During the bin-packing process, whenever we find that we cannot place instances in the existing bins, we need to open a new bin. In order to choose the VM type, we calculate the ratio of compute to memory requirements.

Table 4.1 shows the ratio of compute to memory resources provided by different VM types in

AWS and Azure cloud offerings. This allows us to create a simple policy for choosing which type of bin to open whenever an instance cannot be fit into previous bins: if we have an instance at hand that has a compute:memory ratio of ≤ 0.13, we can open a memory intensive bin. If the ratio falls between $0.13 - 0.53$, we can open a general-purpose bin, and a compute intensive bin if the ratio is > 0.53.

As mentioned in Section 4.3.3, once the type of bin is chosen, Meezan always selects the largest bin in that category. This provides us with an opportunity to fill the bin with as many instances as we possibly can to fully utilize the bin, and ensure that there as few edges crossing over the container boundary as possible. We later optimize our policy such that if resources are left unused in the VM, we replace the VM with the smallest VM that can fit all the instances and has less fragmented resources.

Bin Packing: In order to reduce cross-container communication, Meezan packs as many directly connected operators as possible into a single bin. Once the parallelism of each instance is determined, alongside its resource allocation and the rate of data flowing along each edge, Meezan sorts all edges in descending order by weight. Then, it chooses the heaviest edge and attempts to insert its vertices (instances) into a bin. In order to do so, a new bin (VM) must be selected first. Meezan aggregates the requirements of the two instances and calculates their compute:memory requirements ratio. Given the conditions in the previous section, it chooses a particular type and size of VM to place these instances in. Then, Meezan repeats the steps below until all instances have been inserted into bins.

Consider the next edge amongst the sorted edges, and its associated vertices. There are three possibilities with respect to the associated vertices:

1. Both vertices have already been placed in which case Meezan moves onto the next step.

2. One vertex has already been placed. In this case, Meezan attempts to insert the remaining vertex in the container with the pre-allocated vertex. If the VM does not have remaining resources to do this, Meezan calculates the remaining vertex's *alignment score* [81] per every open bin, and allocates the instance in the bin with the largest alignment score.

   The alignment score is a weighted dot product between the vector of VM's available resources and the instance's resource requirements. Meezan includes the amount of data the stream manager can tolerate alongside CPU, memory and storage. This is beneficial as the alignment score is largest for the VM where the instance uses up the most resources along every resource type. This loosely follows a best-fit approach where VMs with the largest open spaces are filled up first. If there is no VM that can accommodate the instance, a new bin has to be opened.

3. Neither vertex has been placed. In this case, Meezan first attempts to place both vertices with an upstream instance. It sorts upstream instances in the order of highest data rate to the downstream vertices and goes through them until it can place the vertices in their VM. If Meezan cannot collocate the instances with an upstream instance, it tries to place them together in a bin according to their alignment score. If that is not possible, it tries to place them separately according to their alignment score, and finally opens a new bin if that too fails.

Once the packing plan has been created, some VMs may not be packed fully. For every VM that is not fully packed, Meezan iterates through all cheaper VMs, sorted by cost, and tries to replace the VM with a smaller, cheaper one, that fits the instances better. In this way, Meezan tries to reduce the cost of the packing plan and reduce fragmentation.

Baselines: We derive two baselines to evaluate and compare Meezan's packing heuristic.

1. **Analytical Model (ILP):** We use a multi-objective integer linear program (ILP) [32] that aims to produce the optimal, cheapest job deployment with the least fragmentation. This provides us with the best-case solution that heuristics cannot improve over.

   Notations: We have a maximum number f of each VM type to place instances on, on our chosen cloud platform. All job instances have resource requirements along four dimensions: CPU, memory, disk and communication bandwidth (which is the sum of its input and output rates). For each resource r, we state that the an instance d has a resource requirement of d^r and the capacity of that resource on a VM v is c_v^r. Communication bandwidth on a VM is the maximum data rate a stream manager can tolerate. Each VM has an associated price p_v. An indicator variable Y_{dv} is 1 if instance d is placed on VM v and is 0 otherwise. Another indicator variable X_v is 1 if VM v has instances placed on it and is 0 otherwise.

   Constraints: First, an instance d is placed on one VM only:

   $$\sum_{v} Y_{dv} = 1,\ \forall\ d,\ v \tag{4.2}$$

   Secondly, the sum of resource requirements of all instances placed on a VM must not exceed its capacity:

   $$c_v^r - \sum_{r} (Y_{dv} * d^r) >= 0,\ \forall\ d,\ v,\ r \tag{4.3}$$

   Objectives: We have two objectives for this model. First, we minimize the total cost of

deployment.

$$Cost = \sum_{v} (X_v * p_v) \tag{4.4}$$

Second, we minimize the total number of VMs used, to reduce fragmentation:

$$Total\ VMs\ Used = \sum_{v} X_v \tag{4.5}$$

This is essentially a problem of packing multi-dimensional balls into a minimal set of bins which is known to be NP-Hard [162]. In order to solve this problem in a reasonable amount of time, we fix the f i.e., the number of each VM type that is supplied as input to the problem. However, we note that by doing so, we limit the optimization problem's input, thereby getting solutions that may not be optimal globally.

2. **Resource Compliant Round Robin (RCRR):** The best packing policy Heron comes with by default is RCRR. This policy essentially places all instances of a job in a round robin manner, across all given bins, while making sure that the capacity of any of the bins is not violated. However, by default, the user has to specify the resource requirements of each instance, and the number of VMs.

   Confusion has been observed anecdotally amongst engineers as the API for the policy also allows users to set both the maximum number of instances per VM as well as the size of the VM. This is odd, as if the number of VMs and instance size is specified, the sizes of the VMs required can be derived from the instances placed on them. Additionally, the API limits users to specify one VM size only that is meant to be used as the default size of all VMs. Furthermore, the policy has no conception of modelling data rates as one of the resources.

   We modify this policy to accept the resource requirements of each instance, as derived by Meezan's profiler, and the maximum number of bins given by the ILP. However, as a round robin policy does not pack instances efficiently, it is possible that the exact VMs derived from the ILP may not be sufficient for all the instances to be placed in a round robin manner. Therefore, we specify the number of VMs for RCRR as the number derived by the ILP, however we provide RCRR with VMs of the largest available size on the chosen cloud platform so that none of the instances are left unplaced.

4.4 EVALUATION

In evaluating Meezan, we are interested in answering the following questions:

- To what extent does Meezan minimize the cost of resources (or VMs) used on cloud platforms?
- What extent are resources fragmented to with Meezan as compared to other packing algorithms?
- Does Meezan successfully ensure that the stream manager per VM does not become a throughput bottleneck?

4.4.1 Evaluation Setup

In order to answer these questions, we select several real world workloads to evaluate Meezan.

Our first workload is the linear road benchmark [54] which consists of three jobs and is also employed by several previous works to evaluate improvements in stream processing systems [93, 169]. It models a road toll network, in which tolls depend on the time of day and level of congestion. It specifies the following queries: (i) detect accidents on the highways (LRB Accident Detection), (ii) notify other drivers of accidents (LRB Accident Notification), and (iii) notify drivers of tolls they have to pay (LRB Toll Calculation). Each of these benchmarks is run as a single topology, where a single spout is responsible for generating the stream of traffic per highway. As we increase the number of spouts, we are able to process information for proportionally more highways. The first two topologies are linear, with depths of 3 and 4, whereas the LRB Toll Calculation job has joins and a depth of 5.

We use the Yahoo! Streaming Benchmark [151] as our second workload, which models a simple advertisement application. This a linear topology with a depth of 6. There are a number of advertising campaigns, and a number of advertisements for each campaign. The job of the benchmark is to read JSON events, identify them, and store a windowed count of relevant events per campaign into Redis. These steps attempt to probe some common operations performed on data streams. As we scale up the number of spouts, we are able to process proportionally more events.

As our final two workloads, we use 1) a 2-stage Word Count topology that has been used by several previous works [75, 103] and produces words whose frequency is subsequently counted, and 2) a similar Exclamation Topology that appends punctuation to sentences (also used in Figure 4.4a), to evaluate the systems under consideration.

We run our evaluation on a 30-node cluster on Emulab, where we provide each job the same choices of VMs that are available to users on AWS [3].

4.4.2 Implementation

Meezan is integrated into Apache Heron [112], as a scheduler. It is an implementation of the predefined IScheduler. It starts up as a service that profiles major VM types on given cloud platforms

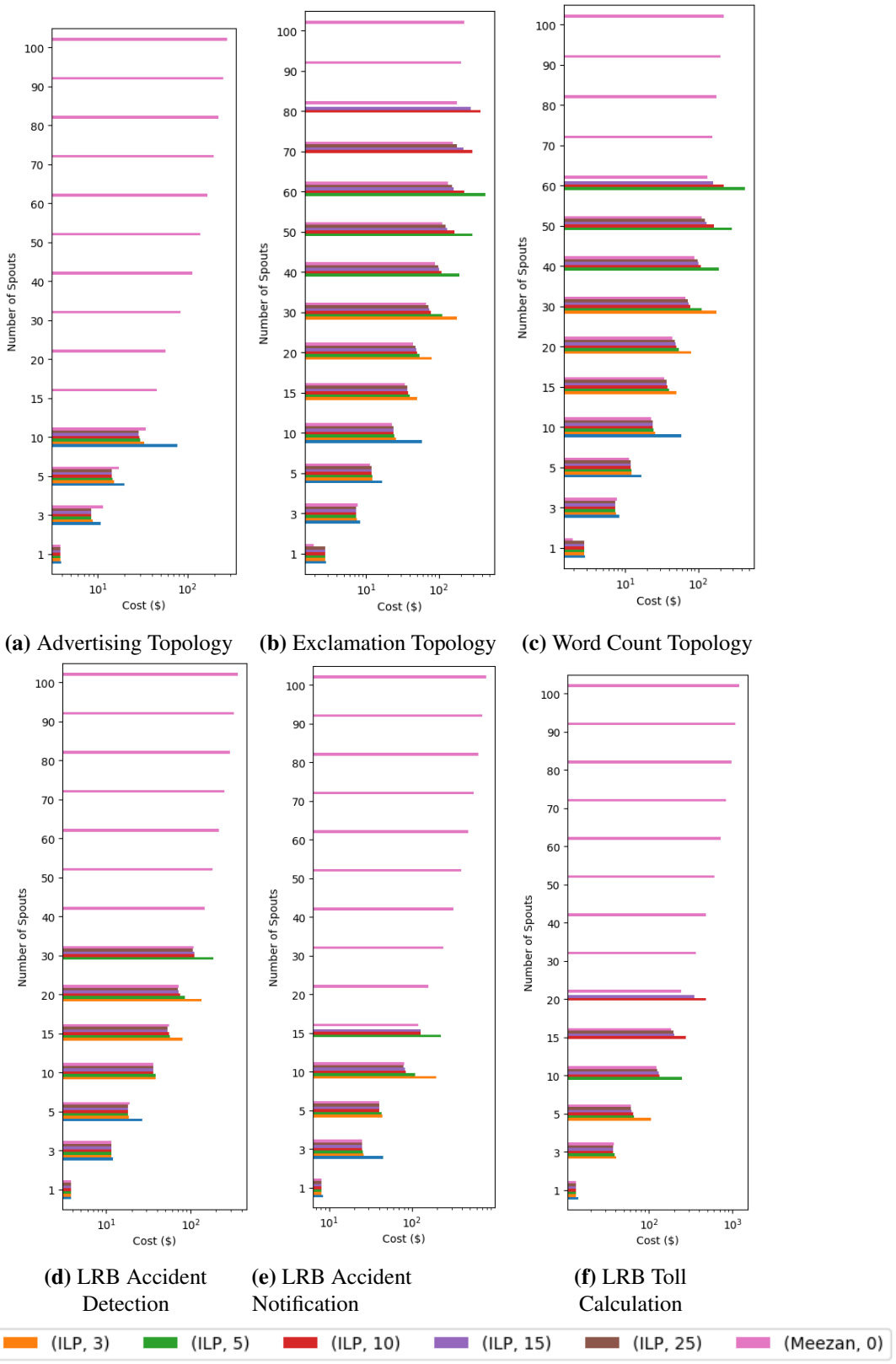
to understand the maximum throughput rates stream managers can tolerate on them (Section 4.3.3). This is performed only once during startup, and can be repeated if new VM types are added.

When users submit jobs for deployment, they can specify Meezan as their scheduler of choice in configuration. During job deployment, Meezan expects that users will provide jobs that implement a predefined interface, that essentially allows Meezan to modify the amount of resources and parallelism allocated to each job operator. Once jobs are submitted to Meezan, it utilizes this flexible configurability to profile the job to find it's optimal resource allocation (Section 4.3.2.) Given the results of profiling, Meezan produces its cost-performance spectrum for the job (Section 4.3). The user can then choose a job deployment that satisfies their throughput goals and falls within their budget, and Meezan submits their job for deployment on the relevant infrastructure of the user's cloud platform. Currently, we allow deployment on Emulab, however, hooks can be added for deployment on AWS, Azure and Google Cloud.

4.4.3 Cost Benefits

One of the main goals of Meezan was to provide users with an array of deployment choices of various costs and performance guarantees to choose from. We evaluate how the cost of a job's deployment increases as we scale it out with Meezan, and compare it to the optimal deployment cost. Results for each of our benchmark jobs are shown in Figure 4.8. The y-axis shows the job's scale (or the number of spouts it is consuming data from) and the x-axis represents the total dollar cost of deployment for an hour. We note that in most cases, the cost of deployment provided by Meezan is comparable to that of the cost provided by the ILP, when the largest (=25) frequency of each VM type is provided to it. This essentially means that the ILP is able to optimize cost the most when it has more VMs to choose for placement. Concretely, when number of VMs provided to the ILP is 25 and the number of spouts for each job are 10, Meezan and the ILP's deployment cost for the Accident Detection Topology are the same = $36.29. In the same circumstances, the cost for the Accident Notification topology given by the ILP is $79.16, and Meezan's cost is $80.21. We observe the largest difference with the Yahoo! Advertising Topology, where Meezan's cost is $34.37 and the cost provided by the ILP is $28.44, which is a 20.85% degradation.

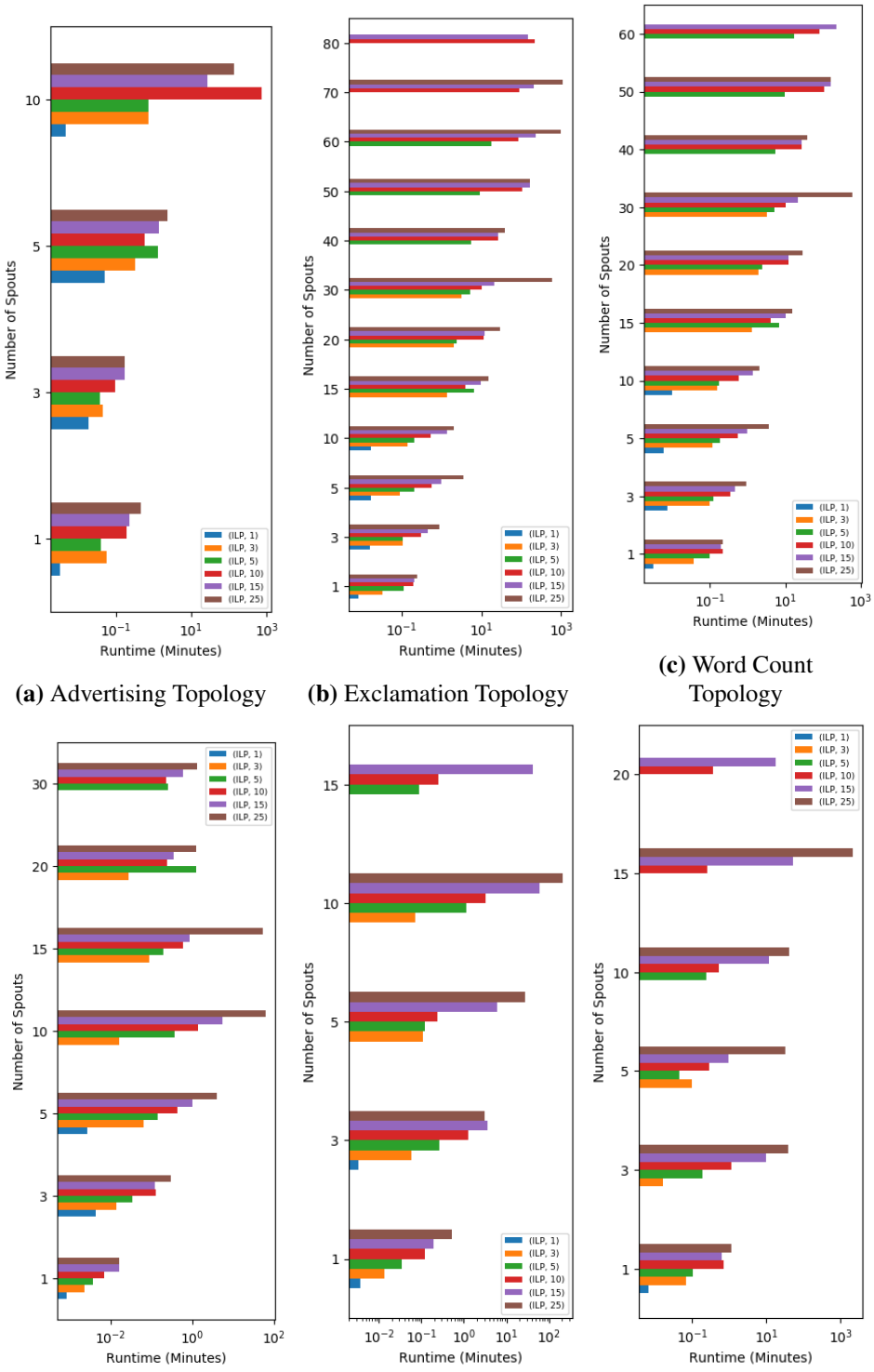
Readers will notice that we do not provide a cost comparison as jobs scale beyond 40 spouts at most. This is because beyond 60 spouts, we found that solving the ILP took an exorbitant amount of time, as shown in Figure 4.9. Generally, we note that as the job size increases and the number each VM type provided to the ILP increases, computation time increases. In reasonable cases, such as the the WordCount Topology scenario described above, Gurobi took 2.8 minutes to solve the problem. In extreme cases such as for the advertising topology, given 25 of each VM types and a job scale of 10 spouts, Gurobi took approximately 75 minutes to solve the problem.



(a) Advertising Topology **(b)** Exclamation Topology **(c)** Word Count Topology

(d) LRB Accident Detection **(e)** LRB Accident Notification **(f)** LRB Toll Calculation

Figure 4.8: Meezan vs ILP in terms of dollar cost of job deployment. Y-axis indicates increasing number of job spouts. The legend indicates (Scheduler type, Number of each VM Type provided to the ILP). As Meezan is not constrained in terms of the number of VMs it can use, we use 0 for VM frequency.



(a) Advertising Topology

(b) Exclamation Topology

(c) Word Count Topology

(d) LRB Accident Detection

(e) LRB Accident Notification

(f) LRB Toll Calculation

Figure 4.9: Time required by Gurobi to solve the ILP for different workloads, with each varying frequencies of available VMs (from 3 to 25 of each type).

Readers will also notice that some costs of deployments are not provided (e.g., for the LRB Accident Detection Topology, where spouts = 30, and frequency of VMs = 3). In such cases, finding a solution was infeasible as even if all the given VMs were used, resources were insufficient for deployment of the entire job.

4.4.4 Fragmentation

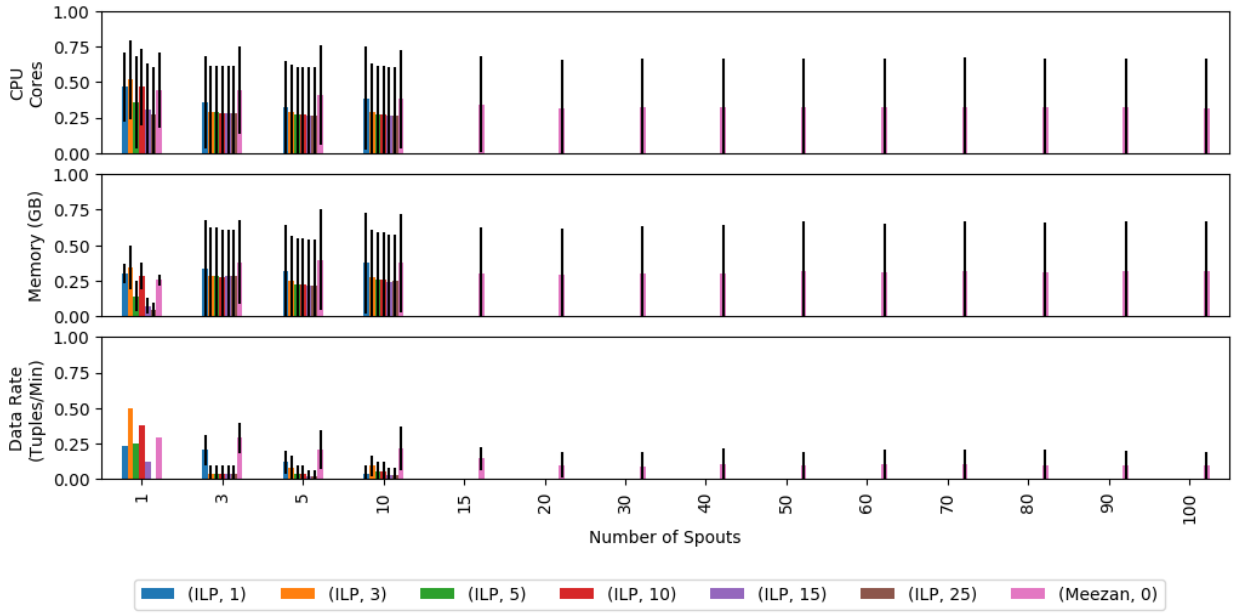


Figure 4.10: Fragmentation of resources in deployments of Meezan vs ILP in the Yahoo! Advertising Topology

We show figures 4.12-4.11 that depict the degree of fragmentation or wasted resources per deployment in the cases of the LRB Toll Calculation, the Advertising Topology and the Word Count Topology, which are representative of the remaining cases. The y-axis represents the ratio of each of the unused resources per VM to the amount available in the VM. In figures 4.10 and 4.12, we note that that the data rate of the stream manager is almost fully consumed, leading to fragmentation of CPU and memory resources. This essentially confirms that the instances placed in each of the VMs are communication-intensive. Figure 4.12 is a particularly pathological case where one instance is placed per VM, as that instance consumes more than half of the VM's data rate. Therefore, CPU, memory and some of the stream manager's resources are fragmented.

Next, we note that Meezan's fragmentation is close to or better than that of the optimal configurations. We compare the product of the ratios of each of the fragmented resources per policy to evaluate this. In the best case, we note in the best case of the Toll Calculation Topology where the number of spouts = 10 and the frequency of VMs is 10 for the ILP, the fragmentation in Meezan's

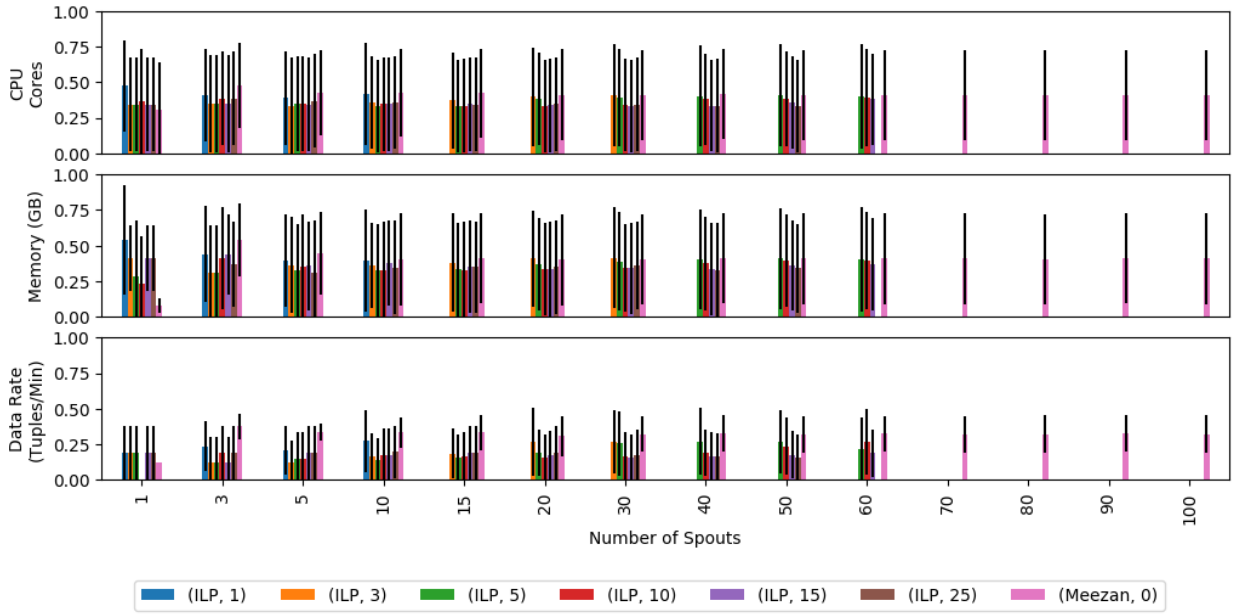


Figure 4.11: Fragmentation of resources in deployments of Meezan vs ILP in the Word Count Topology. The legend indicates (Scheduler type, Number of each VM Type provided to the ILP).

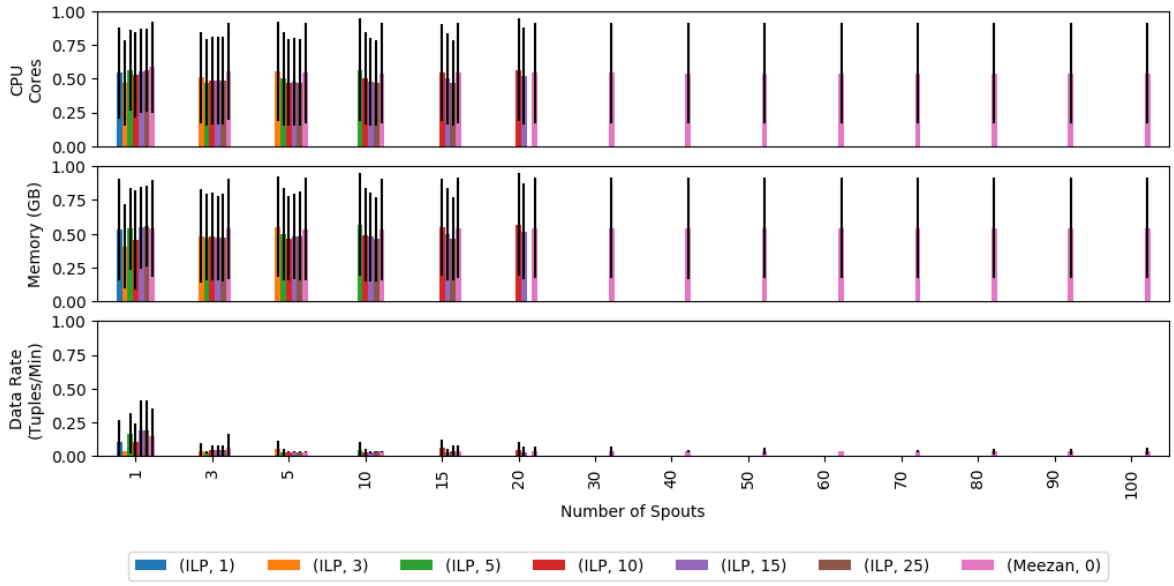
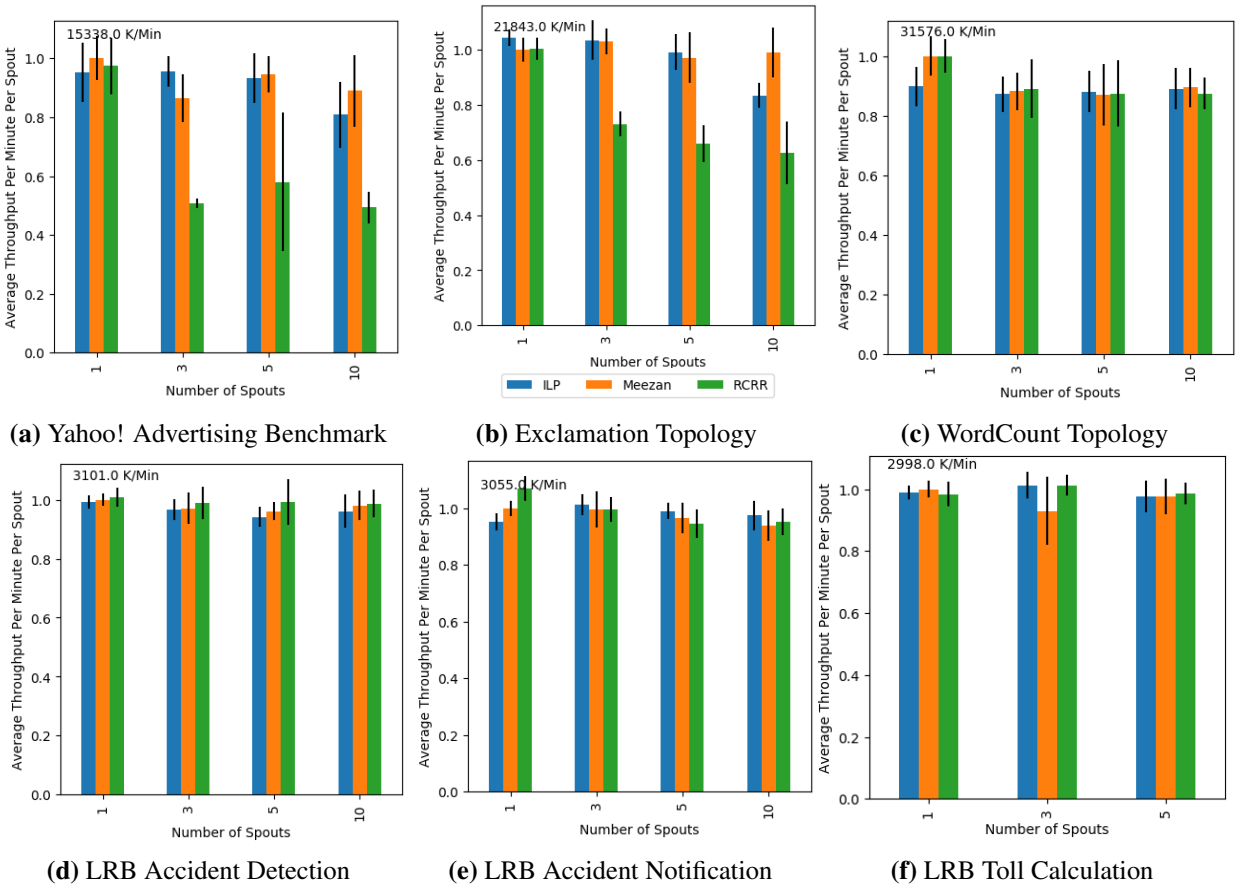


Figure 4.12: Fragmentation of resources in deployments of Meezan vs ILP in the LRB Toll Calculation Topology. The legend indicates (Scheduler type, Number of each VM Type provided to the ILP).

deployment is 0.76 vs the 0.97 fragmentation given by the ILP, which is a 27% improvement. In the worst case of the Advertising Topology, with 3 spouts, and frequency of 10 VM types, Meezan does 15% worse than the ILP. (Meezan's fragmentation is 0.9 and the ILP's fragmentation is 0.78).

4.4.5 Throughput



(a) Yahoo! Advertising Benchmark **(b)** Exclamation Topology **(c)** WordCount Topology

(d) LRB Accident Detection **(e)** LRB Accident Notification **(f)** LRB Toll Calculation

Figure 4.13: Meezan vs ILP in terms of average throughput per-spout for varying workloads as they are scaled out.

Meezan aims to pack instances onto VMs while ensuring that the stream manager in each VM is not bottlenecked by extremely high throughput. Therefore, as we scale jobs out from having 1 to 10 spouts, and pack them onto VMs, we evaluate the average throughput of each spout in each job and present them in Figure 4.13. If Meezan does its job correctly, the average throughput per spouts should remain in the same range. We notice that the average throughput across all spouts in the LRB topologies are similar to each other, irrespective of scheduler choice. This is because each of the spouts performs some computation to generate the LRB workload, which means that their throughput is not high enough to bottleneck the stream manager. This means that regardless of the packing policy, the average throughput per spout remains high.

The Advertising Topology, Exclamation Topology and Word Count Topology all have high throughput rates. In the case of the advertising topology and the exclamation topology, we note that Meezan and the ILP are able to maintain the throughput of each spout to the average produced by a single spout. However, we notice that as RCRR places instances in bins in a round robin fashion,

it instances together, causing the stream manager to become a bottleneck. We notice thus that the average throughput per spout with RCRR falls up to 43% in the case of the Advertising Topology and the Exclamation topology.

In the Word Count Topology, we note that RCRR does as well as Meezan. This is because the RCRR is given a very high number of VMs to use (from the ILP, which essentially packs one operator per VM). Because of the large number of VMs available that RCRR must use, it packs one operator per VM. As a result, the stream manager is not bottlenecked and its throughput levels remain high on average.

4.5 RELATED WORK

4.5.1 Stream Processing as a Service

Amazon Kinesis [2] offers itself as a Streaming as a Service Platform. It enables easy collection, processing and analysis of real-time streaming data. However, the system leaves performance tuning of the job up to the user. In Amazon Kinesis, each data stream is a function of the number of shards that the user specifies for the stream. The total capacity of the stream is the sum of the capacities of its shards. As the input data rate changes, the user needs to increase or decrease the number of shards to maintain throughput. Our goal is to remove this burden from the user's shoulders.

4.5.2 Optimal Resource Allocation in Distributed Systems

Perforator [140] is one of the first systems that tackles the resource allocation problem for DAGs of batch-compute jobs. It models, first, the estimated size of input data that the job must process and second, the job's performance, by analyzing results of hardware calibration queries, and using them to ascertain the parallelism of tasks in the execution frameworks. Ernest [158] is a performance prediction framework for batch jobs, that uses optimal experiment design. This is a statistical technique that allows its users to collect as few training points as possible, that indicate how short runs of each incoming jobs perform on specific hardware. These training points can be used to model the performance of these jobs on specific hardware, with larger input sizes, that can be used to predict near-optimal resource allocations for them. Cherrypick [49] is a system that leverages Baysian Optimization, a method for optimizing blackbox functions, to find near-optimal cloud configurations that minimize cloud usage cost and guarantee application performance for general batch-compute applications. The main idea of Cherrypick works as follows: start with a few initial cloud configurations, run them, and input the configuration details and job completion time into the performance model. The system then dynamically pick the next cloud configuration to run based on

the performance model and feed the result back to the performance model. The system stops when it has enough confidence a good configuration has been found.

The problem of finding optimal resource allocation for jobs has also been explored in other distributed systems such as systems for deep learning. FlexFlow [98] is a deep learning engine that employs a guided randomized search to quickly find a parallelization strategy of a deep learning training job for a specific parallel machine. In order to do so, FlexFlow first borrows the idea of measuring the performance of an operator once for each configuration from OptCNN [97], which is a similar approach to Meezan, and feeds these measurements into a task graph that models both the architecture of a DNN model and the network topology of a cluster.

Fine-grained resource allocation is also studied for micro-service architectures. MONAD [132] is a self-adaptive infrastructure for micro-services that are meant to run heterogeneous scientific workflows. Using fine-grained scheduling at the task-level, MONAD improves the flexibility of workflow composition and execution, It also shares tasks between multiple workflows. Furthermore, it uses feedback control with neural network-based system identification to provide resource adaptation without in-depth knowledge of workflow structures.

The problem of minimizing deployment cost exists in many domains, including serverless computing, that is used to process data across the edge and in data centers. Costless [73] presents an algorithm that optimizes the price of serverless applications on Amazon AWS Lambda. The paper describes factors that affect the price of a deployment: (1) fusing a sequence of functions, (2) splitting functions across edge and cloud resources, and (3) allocating the memory for each function. Costless runs an efficient algorithm to explore different function fusion-placement solutions and choose one that optimizes the application's price while keeping the latency under a certain threshold.

4.5.3 Video Stream Processing

Video stream processing jobs require additional effort to minimize the massive computational cost of performing computer vision tasks that Meezan does not take into account so far. For example, one work [94] asks whether video analytics can be scaled in a way that cost grows sublinearly, as more video cameras are deployed and inference accuracy of processing remains stable. To achieve their goal, the authors observe that video feeds from wide-area camera deployments demonstrate significant content correlations to other geographically proximate feeds, both spatially and temporally. These correlations can be harnessed to dramatically reduce the size of the inference search space, decreasing both workload and false positive rates in multi-camera video analytics.

This problem is also explored for environments where video stream processing jobs are deployed on a hierarchy of clusters. VideoEdge [91] is a system that uses the concept of "dominant" demand to identify the best tradeoff between multiple resources and accuracy, and narrows the search

space for placement on a hierarchical set of clusters, by identifying a "Pareto band" of promising configurations. The system also balances the resource benefits and accuracy penalty of merging queries.

4.5.4 Performance Modelling of Message Brokers

Balasubramanian [56] model a scenario in publish-subscribe systems where the publishers, the broker, and the subscribers are in different administrative domains and have no guarantees with respect to the resources available to each service. Within this context, they focus on dynamically adapting the amount of data that the publishers send to the brokers to prevent backpressure at the brokers. Nguyen [131] model publish-subscribe systems as multiple-class open queueing networks to derive measures of system performance. Using these measures, they solve objective functions that have the goal of either minimizing the latency for processing an input which translates to finding the optimal number of consumers in the system or finding the least end-to-end latency that can be provided while minimizing the total resource cost of consumers in the system. The former problem is the same as calculating the parallelism of downstream bolts in a continuous-operator stream processing topology.

To the best of our knowledge, these are the only work that experimentally evaluates the performance of recent message brokers such as Apache Kafka [111].

4.5.5 Backpressure in networks

The slowdown of upstream operators in Heron jobs because of a bottlenecked stream manager can be likened to the backpressure exerted in networks due to congestion control [50]. However, the two scenarios have different solutions because the causes of their bottlenecks are different: congestion in networks is caused by limited bandwidth [134] (if the network configuration is correct and well-designed), while stream manager slowdown is caused by a lack of computational resources. Therefore, Meezan removes the stream manager bottleneck by ensuring that each stream manager is not oversubscribed, and only receives as much data as it is able to transfer per unit time. Meezan's scope does not include handling network-related backpressure.

4.5.6 Scaling in Stream Processing Systems

A great deal of research considers the problem of scaling resources in and out to meet performance goals in stream processing systems. Drizzle [157] adapts the size of the batch in micro-batch stream processing systems such as Spark Streaming [168] to provide low latency and high throughput. DRS

[76] has applied queueing theory to create performance models for continuous-operator stream processing systems that do not have intermediate message brokers such as Apache Storm [154]. Tri Minh et al. [155] consider the performance of stream processing systems in terms of back-pressure and expected utilization. They focus on a queueing theory-based approach that is used to predict the throughput and latency of stream processing while supporting system stability in the face of bursty input.

Dhalion [75] proposes the use of self-regulating stream processing systems and provides a prototype with some capabilities that can diagnose problems e.g., backpressure in Heron topologies and can take actions to mitigate the problem. It's corrective actions are heuristics, however, DS2 [102] provides scaling methods that are based on precise processing and output rates of individual dataflow operators. Henge [104] supports SLO-driven multi-tenancy in clusters of limited resources.

Authors of [59] use a hill-climbing algorithm that uses a new heuristic sampling approach based on Latin Hypercube to automate the process of tuning some of the configuration parameters of continuous-operator systems like Apache Storm [154]. However, the work is not able to cover a majority of the parameters that need to be tuned for optimal performance.

Trevor [58] is parallel work that focuses on auto-configuring stream processing jobs especially those run on Apache Heron. Like [103], it builds a simple model that compares an instance's input rate with its output rate to model the instance's behavior and uses heuristics to ensure that each stream manager per container is not bottle-necked. However, it does not aim to minimize the amount of resources used by job or lower the cost of running the job for the user.

4.6 CONCLUSION

Meezan is a scheduler for stream processing systems that provides users with a range of possible deployments for their jobs. Each deployment has a different price and a different performance guarantee. The most performant choice is associated with the highest cost and vice versa. We compare Meezan with a multi-objective optimization framework that given a job DAG and a fixed set of VMs, optimizes the total cost of deployment and minimizes the number of containers used for a job. Meezan is able to ensure that the choice of deployment it provides users with is within 20.85% of cost found by the optimization problem. Additionally, as it is able to open up bins of its choice while the optimization framework is limited to a fixed set of bins, it is able to reduce resource fragmentation by up to 27% over the framework's solution. While packing, Meezan ensures that it does not allow jobs to be bottlenecked by components that are transparent to the job developer, and ensures that as jobs scale up, their average throughput increases in proportion to their scale.

Chapter 5: Conclusion and Future Work

Even though stream processing systems have become increasingly prevalent with the increased use of applications that require low-latency responses, they still face several challenges in terms of achieving performance goals in different scheduling environments. In multi-tenant environments with limited shared resources, jobs have to be assigned resources after carefully considering their job complexity, priorities and performance requirements. These resource allocations must change over time as jobs receive input at varying rates. On the other hand, while scheduling jobs onto public cloud services such as Amazon AWS [3] or Microsoft Azure [30] where resources are virtually unlimited, we must carefully determine the most optimal VM choices in terms of cost and resources, that will help us achieve our performance goals. Such problems motivate the need for both online and predictive mechanisms that determine optimal resource allocations for jobs before they are launched, and continue to adapt their allocations to dynamically changing conditions.

In this thesis, we present practical techniques for achieving performance goals or Service Level Objectives (SLOs) [38] in stream processing systems that are used by a variety of users in different data center environments.

1. Henge [104] is an online scheduler for stream processing systems that addresses the challenge of adapting job resource allocations in a multi-tenant environment with limited resources to maximize cluster utility. It does so by ensuring that the SLOs of higher priority jobs that provide higher utility to the organization are met first, and if resources are available, ensures that the SLOs of lower priority jobs are also met. Henge reduces intrusiveness by making the least number of modifications it can to each job. In addition, we show that Henge converges provably if job input rates stabilize i.e. once input rates of all jobs stabilize, Henge is always able to find a resource allocation for all jobs in the cluster that maximize cluster utility.

2. Caladrius [103] is a scheduler for stream processing jobs that is built in collaboration with Twitter that predicts changes in the input rates of a job and scales it out (or in) preemptively to ensure that the job's throughput SLOs are always met, while reducing resource wastage. In order to make predictions of future input rates, Caladrius uses Facebook's Prophet [34], a time-series forecasting tool that is optimized for predicting observations that depict strong human-scale seasonalities, which our experience at Twitter shows that stream processing jobs most often do. Given these forecasted changes, Caladrius models components in the job graphs and recalculates their parallelism levels to ensure that they are not bottlenecked because of predicted increases in input rate. Based on the job's new parallelism level, Caladrius derives a new resource allocation for the job to which it can be scaled out preemptively to ensure that the job does not miss its SLO at any point in time.

3. Meezan allows novice users of stream processing jobs to select a resource allocation of their stream processing job on public cloud services that will meet their throughput-based performance goals while reducing cost of deployment. In Meezan, we present two contributions: 1) we present a case study that models how dedicated VMs are priced on Amazon and Azure, and using insights from our study, 2) we propose a novel packing algorithm that minimizes the cost of job deployments on these platforms, while ensuring that the job's throughput SLO is met. A fundamental contribution of our packing algorithm is that it takes components of job graphs into account that are transparent to the job developer but are fundamental to the job's function, such as message brokers that are responsible for data transmission among the job's components.

The steps needed to deploy jobs per environment are described in each system's respective chapter. To production-ize these systems, some engineering work is required to hook them up with existing company infrastructure.

This thesis lays the ground work for several future directions:

Message Brokers as In-Network Components: In recent years, the academic community has proposed using in-network computation to implement a vast variety of functionality that is normally performed by systems [138]. This includes concurrency control [96, 115], aggregation primitives [143], consensus protocols [67, 68, 116, 137], and query processing operators [83, 113]. Other work offloads entire applications to programmable devices, including key-value stores [153], network protocols like DNS [153], and even industrial feedback control [142].

We note through Meezan that Heron topologies have a stream manager that is a great candidate for in-network processing. We observe that the stream manager is responsible for receiving data from local operators and passing it on to other destination operators that are either local or are remote. If the operators are placed remotely, a stream manager simply passes the data to the stream manager that is local to the destination operator, and that stream manager passes the data onto to the local operator. Therefore, we observe that the role of the stream manager is very much like that of a network switch. It is also a good candidate for in-network processing as it is not performing complex computation on data: it has been designed to simplify the role of operators, so that operators do not maintain connections with remote operators and simply pass the data to the stream manager. Additionally, the stream manager also must only maintain connections with local operators and remote stream managers, instead of maintaining connections with all operators. The essential work the stream manager is doing is determining which data packet should be sent to which destination, and batching it together to be sent to the recipient. By moving this task to the network, we can save on the cost of moving data from the transport layer to the application layer, where it has to be deserialized to be read. This can potentially lead to significant reductions on end-to-end latency of a

tuple as it moves throughout the topology, especially as we note that there can be multiple stream managers in its path.

Satisfying Global Latency SLOs: In this thesis, we have defined latency SLOs with respect to the amount of time it takes a tuple to be processing fully in a stream processing topology running in a data center. However, most stream processing applications process data that arrives into the data centers from across the world and require that responses that should be sent received at the sender within a few seconds. Such "global" latency goals can be very business-critical. For example, if a ride-sharing app is not able to send a response back to a rider with a driver match in a few seconds, it risks losing the rider to a competitor [28]. However, in situations where requests and responses are sent over the WAN, network congestion can create difficulties in promising low latencies [60, 100]. A possible future direction of this thesis is to derive minimum and maximum latency SLOs that can be guaranteed to the users and can be met with a defined degree of reliability.

Stream Processing as SIMD Applications: Stream processing jobs are applications that can be naturally run on SIMD (Single Instruction, Multiple Data [37]) architectures, as their job operators involve applying the same instruction to multiple pieces of incoming data. This means that we can easily use GPUs to speed up the performance of stream processing applications to meet SLOs, especially those running on micro-batch streaming architectures [69] such as Spark Streaming [168], which attempt to process small batches of data at a time. An interesting direction would involve discovering the extent to which we can automatically translate a given application's code that is meant to run on a CPU to one that would run as a SIMD application. For instance, operations such as joins would not be sped up by using GPUs.

Defining Fairness in Multi-Tenant Stream Processing Clusters: Defining fair-shares of tasks in a distributed systems is challenging and some works have explored fairness in batch data analytics systems [80, 81]. This task is challenging because although fairness requires an "equal" share of resources across jobs, one job's share might be an excess for it, whereas another job's share might not even be enough for a single operator of the job. However, fairness is important, especially in stream processing as jobs are long-running and even some amount of starvation can lead to backlogs of data.

In Henge, we define fairness in accordance with a job's priority. In batch computation, several fairness-related measures have been developed such as dominant resource fairness [80], which are not directly applicable as a stream processing jobs face more dynamism and their dominant resource share might change over time. Therefore, a possible future direction is to express fairness based on performance goals – for example, a fair allocation would be where all jobs of the same

priority achieve the same percentage of their performance goal – and to describe it's fair-division properties [61].

Generalizing Workloads, Job Components, and Resources:

Machine Learning Inference Workloads: Stream processing jobs can mirror the behavior of other long running jobs such as machine learning inference jobs. However, the computation in machine learning DAGs can be quite different from the general operators used in stream processing. In addition, they can also require the usage of more heterogeneous resources such as GPUs and TPUs [14]. A possible future direction is to meet define and achieve SLOs of ML inference jobs in difference cloud environments. Although this topic has begun to gain traction very recently [82,171], an interesting direction is to adaptively move inference components from data center environments to edge devices [144] in order to reduce end-to-end latency.

External Bottlenecks: In the same vein, another possible future direction is to evaluate systems such as Apache Samza [6] and Flink [4] that utilize message brokers e.g., Apache Kafka [111], in place of stream managers in stream processing systems. As these message brokers also write data to disk, they can be IO bound, and can form potential bottlenecks in stream processing jobs that have strict, low latency SLOs. A possible direction of future work is to determine the optimal number of partitions and message brokers for a job to achieve both its latency or throughput SLO.

Dynamically Changing DAGs: Within this thesis, we focus on stream processing DAGs that do not change at runtime. However, in production deployments, it is possible that operators are dynamically added to jobs to deal with challenges at runtime, such as unexpectedly high input and failures. It is also common that such dynamic additions are short-lived. An interesting direction to explore is A) when are such dynamic changes necessitated and when should they be added, and B) how do allocate these operators to maximize resource utilization and reduce deployment cost.

Theoretical Guarantees An interesting obstacle for using Meezan in production is that users have to be convinced of its efficacy in order to allow it to make decisions based for them, that cost them real money. For example, given that Netflix has thousands of data shards (streams) coming into AWS Kinesis (a streaming platform on AWS) daily [33], we can estimate that they spend ($0.36 (cost per day) $\times$ 1000 (streams) $\times$ 30 (days in a month)) = $10,800 per month for only a thousand streams [9]. As these numbers are quite large, it is natural that user's would like assurance that Meezan's packing policy minimizes cost of deployment. In order to overcome this obstacle, a possible future work is to derive theoretical bounds for the best and worst cases that Meezan can provide.

References

[1] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Last Visited: Sunday 12th July, 2020.

[2] Amazon Kinesis. https://aws.amazon.com/kinesis/data-streams/faqs/. Last Visited: Sunday 12th July, 2020.

[3] Amazon Web Services. https://aws.amazon.com/. Last Visited: Sunday 12th July, 2020.

[4] Apache Flink. http://flink.apache.org/. Last Visited: Sunday 12th July, 2020.

[5] Apache Flume. https://flume.apache.org/. Last Visited: Sunday 12th July, 2020.

[6] Apache Samza. http://samza.apache.org/. Last Visited: 03/2016.

[7] Apache Storm. http://storm.apache.org/. Last Visited: Sunday 12th July, 2020.

[8] Apache Zookeeper. http://zookeeper.apache.org/. Last Visited: Sunday 12th July, 2020.

[9] AWS Kinesis Pricing. https://aws.amazon.com/kinesis/data-streams/pricing/. Last Visited: Sunday 12th July, 2020.

[10] CPU Load. http://www.linuxjournal.com/article/9001. Last Visited: Sunday 12th July, 2020.

[11] EC2 Instance Types. https://www.ec2instances.info/?selected=g4dn.2xlarge. Last Visited: Sunday 12th July, 2020.

[12] Elasticity in Spark Core. http://www.ibmbigdatahub.com/blog/explore-true-elasticity-spark/. Last Visited: Sunday 12th July, 2020.

[13] EPA-HTTP Trace. http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html. Last Visited: Sunday 12th July, 2020.

[14] Google Cloud TPU. https://cloud.google.com/tpu. Last Visited: Sunday 12th July, 2020.

[15] Heron Github Issue 1125. https://github.com/apache/incubator-heron/issues/1125. Last Visited: Sunday 12th July, 2020.

[16] Heron Github Issue 119. https://github.com/apache/incubator-heron/issues/119. Last Visited: Sunday 12th July, 2020.

[17] Heron Github Issue 1389. https://github.com/apache/incubator-heron/issues/1389. Last Visited: Sunday 12th July, 2020.

[18] Heron Github Issue 1577. https://github.com/apache/incubator-heron/issues/1577. Last Visited: Sunday 12th July, 2020.

[19] Heron Github Issue 1888. https://github.com/apache/incubator-heron/issues/1888. Last Visited: Sunday 12th July, 2020.

[20] Heron Github Issue 1947. https://github.com/apache/incubator-heron/issues/1947. Last Visited: Sunday 12th July, 2020.

[21] Heron Github Issue 2454. https://github.com/apache/incubator-heron/issues/2454. Last Visited: Sunday 12th July, 2020.

[22] Heron Github Issue 266. https://github.com/apache/incubator-heron/issues/266. Last Visited: Sunday 12th July, 2020.

[23] Heron Github Issue 2803. https://github.com/apache/incubator-heron/issues/2803. Last Visited: Sunday 12th July, 2020.

[24] Heron Github Issue 3396. https://github.com/apache/incubator-heron/issues/3396. Last Visited: Sunday 12th July, 2020.

[25] Heron Github Issue 3494. https://github.com/apache/incubator-heron/issues/3494. Last Visited: Sunday 12th July, 2020.

[26] Heron Github Issue 3504. https://github.com/apache/incubator-heron/issues/3504. Last Visited: Sunday 12th July, 2020.

[27] Heron Trouble Shooting. https://apache.github.io/incubator-heron/docs/developers/troubleshooting/. Last Visited: Sunday 12th July, 2020.

[28] Introducing AthenaX, Uber Engineering's Open Source Streaming Analytics Platform. https://eng.uber.com/athenax/. Last Visited: Sunday 12th July, 2020.

[29] Kafka Streams. https://kafka.apache.org/documentation/streams/. Last Visited: Sunday 12th July, 2020.

[30] Microsoft Azure. https://azure.microsoft.com/en-us/. Last Visited: Sunday 12th July, 2020.

[31] Microsoft Azure: Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Last Visited: Sunday 12th July, 2020.

[32] Multi-Objective Optimization. https://en.wikipedia.org/wiki/Multi-objective_optimization. Last Visited: Sunday 12th July, 2020.

[33] Netflix Streaming Use Cases. https://aws.amazon.com/solutions/case-studies/netflix-kinesis-streams/. Last Visited: Sunday 12th July, 2020.

[34] Prophet: Forecasting at Scale. https://research.fb.com/blog/2017/02/prophet-forecasting-at-scale/. Last Visited: Sunday 12th July, 2020.

[35] S4. http://incubator.apache.org/s4/. Last Visited: 03/2016.

[36] SDSC-HTTP Trace. http://ita.ee.lbl.gov/html/contrib/SDSC-HTTP.html. Last Visited: Sunday 12th July, 2020.

[37] SIMD. https://en.wikipedia.org/wiki/SIMD. Last Visited: Sunday 12th July, 2020.

[38] SLOs. https://en.wikipedia.org/wiki/Service_level_objective. Last Visited: Sunday 12th July, 2020.

[39] Storm 0.8.2 Release Notes. http://storm.apache.org/2013/01/11/storm082-released.html/. Last Visited: Sunday 12th July, 2020.

[40] Storm Applications. http://storm.apache.org/Powered-By.html. Last Visited: Sunday 12th July, 2020.

[41] Storm MultiTenant Scheduler. http://storm.apache.org/releases/2.0.0-SNAPSHOT/SECURITY.html. Last Visited: Sunday 12th July, 2020.

[42] Uber Streaming Use Cases. https://www.youtube.com/watch?v=YUBPimFvcN4. Last Visited: Sunday 12th July, 2020.

[43] Why Multitenancy Matters In The Cloud. https://en.wikipedia.org/wiki/Multitenancy#Cost_savings. Last Visited: Sunday 12th July, 2020.

[44] Zillow Streaming Use Cases. https://aws.amazon.com/solutions/case-studies/zillow-zestimate/. Last Visited: Sunday 12th July, 2020.

[45] Daniel Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, C Erwin, Eduardo Galvez, M Hatoun, Anurag Maskey, Alex Rasin, et al. Aurora: A Data Stream Management System. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 666–666. ACM, 2003.

[46] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research*, volume 5, pages 277–289, 2005.

[47] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proceedings of the VLDB Endowment*, volume 6, pages 1033–1044. VLDB Endowment, 2013.

[48] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[49] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

[50] Mark Allman, Vern Paxson, Wright Stevens, et al. Tcp congestion control. 1999.

[51] Tanvir Amin. Apollo Social Sensing Toolkit. http://apollo3.cs.illinois.edu/datasets.html, 2014. Last Visited: Sunday 12th July, 2020.

[52] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive Online Scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pages 207–218. ACM, 2013.

[53] Apache Software Foundation . Apache Kafka Supports 200K Partitions Per Cluster. https://blogs.apache.org/kafka/entry/apache-kafka-supports-more-partitions. Last Visited: Sunday 12th July, 2020.

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

[55] Masoud Saeida Ardekani and Douglas B Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–381, 2014.

[56] Sowmya Balasubramanian, Dipak Ghosal, Kamala Narayanan Balasubramanian Sharath, Eric Pouyoul, Alex Sim, Kesheng Wu, and Brian Tierney. Auto-tuned publisher in a pub/sub system: Design and performance evaluation. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pages 21–30. IEEE, 2018.

[57] Cagri Balkesen, Nesime Tatbul, and M Tamer Özsu. Adaptive Input Admission and Management for Parallel Stream Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pages 15–26. ACM, 2013.

[58] Manu Bansal, Eyal Cidon, Arjun Balasingam, Aditya Gudipati, Christos Kozyrakis, and Sachin Katti. Trevor: Automatic configuration and scaling of stream processing pipelines. *arXiv preprint arXiv:1812.09442*, 2018.

[59] Muhammad Bilal and Marco Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 189–200. ACM, 2017.

[60] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[61] Steven J Brams and Alan D Taylor. *Fair Division: From cake-cutting to dispute resolution*. Cambridge University Press, 1996.

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

[63] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale-Out and Fault Tolerance in Stream Processing using Operator State Management. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 725–736. ACM, 2013.

[64] Javier Cervino, Evangelia Kalyvianaki, Joaquin Salvachua, and Peter Pietzuch. Adaptive Provisioning of Stream Processing Systems in the Cloud. In *Proceedings of the 28th International Conference on Data Engineering Workshops*, pages 295–301. IEEE, 2012.

[65] Cloudera. Tuning YARN — Cloudera. http://www.cloudera.com/documentation/enterprise/5-2-x/topics/cdh_ig_yarn_tuning.html, 2016. Last Visited Sunday 12th July, 2020.

[66] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-Based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[67] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. Network hardware-accelerated consensus. *arXiv preprint arXiv:1605.05619*, 2016.

[68] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Noa Zilberman, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. Technical report, Research Report 2018-01. USI. http://www. inf. usi. ch/research_publication. htm, 2018.

[69] Tathagata Das, Matei Zaharia, and Patrick Wendell. Diving into Apache Spark Streaming's Execution Model. https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html. Last Visited: Sunday 12th July, 2020.

[70] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels.

Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[71] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[72] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[73] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[74] Huifang Feng and Yantai Shu. Study on network traffic prediction techniques. In *Wireless Communications, Networking and Mobile Computing, 2005. Proceedings. 2005 International Conference on*, volume 2, pages 1041–1044. IEEE, 2005.

[75] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[76] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *Proceedings of the 35th International Conference on Distributed Computing Systems*, pages 411–420. IEEE, 2015.

[77] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1123–1134. ACM, 2008.

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

[79] Javad Ghaderi, Sanjay Shakkottai, and Rayadurgam Srikant. Scheduling storms and streams in the cloud. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 439–440. ACM, 2015.

[80] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 24–24, 2011.

[81] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015.

[82] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 109–120, 2017.

[83] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371, 2018.

[84] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-Aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 13–22. ACM, 2014.

[85] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-Scaling Techniques for Elastic Data Stream Processing. In *Proceedings of the 30th International Conference on Data Engineering Workshops*, pages 296–302. IEEE, 2014.

[86] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

[87] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[88] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[89] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, 2011.

[90] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 22–22, 2011.

[91] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. VideoEdge: Processing Camera Streams using Hierarchical Clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[92] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[93] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 431–442. ACM, 2006.

[94] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling Video Analytics Systems to Large Camera Deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019.

[95] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[96] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 26–32, 2018.

[97] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In *Proceedings of Machine Learning Research. PMLR 2018*, 2018.

[98] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *SysML 2019*, 2019.

[99] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Scalable Adaptation of Video Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

[100] Cheng Jin, David X Wei, and Steven H Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

[101] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, page 117, 2016.

[102] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[103] Faria Kalim, Thomas Cooper, Huijun Wu, Yao Li, Ning Wang, Neng Lu, Maosong Fu, Xiaoyao Qian, Hao Luo, Da Cheng, et al. Caladrius: A Performance Modelling Service for

Distributed Stream Processing Systems. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1886–1897. IEEE, 2019.

[104] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven Multi-Tenant Stream Processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 249–262, New York, NY, USA, 2018. ACM.

[105] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream Query Planning with Reuse. In *Proceedings of the 27th International Conference on Data Engineering*, pages 840–851, April 2011.

[106] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload Management in Data Stream Processing Systems with Latency Guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing (Feedback Computing)*, 2012.

[107] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. Themis: Fairness in Federated Stream Processing under Overload. In *Proceedings of the 2016 International Conference on Management of Data*, pages 541–553. ACM, 2016.

[108] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. Balancing Load in Stream Processing with the Cloud. In *Proceedings of the 27th International Conference on Data Engineering Workshops*, pages 16–21. IEEE, 2011.

[109] T. Knauth and C. Fetzer. Scaling Non-Elastic Applications Using Virtual Machines. In *Proceedings of the IEEE International Conference on Cloud Computing,*, pages 468–475, July 2011.

[110] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[111] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[112] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 239–250. ACM, 2015.

[113] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[114] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting Scalable Analytics with Latency Constraints. In *Proceedings of the VLDB Endowment*, volume 8, pages 1166–1177. VLDB Endowment, 2015.

[115] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[116] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 467–483, 2016.

[117] Teng Li, Jian Tang, and Jielong Xu. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4):353–364, 2016.

[118] Yao Liang. Real-time vbr video traffic prediction for dynamic bandwidth allocation. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 34(1):32–47, 2004.

[119] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: An Elastic and Highly Available Streaming Service in the Cloud. In *Proceedings of the Joint EDBT/ICDT Workshops*, pages 55–60. ACM, 2012.

[120] Yuanqiu Luo and Nirwan Ansari. Limited sharing with traffic prediction for dynamic bandwidth allocation and qos provisioning over ethernet passive optical networks. *Journal of Optical Networking*, 4(9):561–572, 2005.

[121] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 589–603, 2015.

[122] Markets and Markets. Streaming analytics market worth 35.5 Billion USD by 2024. https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html. Last Visited: Sunday 12th July, 2020.

[123] Markets and Markets. Video streaming market worth 7.5 Billion USD by 2022. https://www.marketsandmarkets.com/Market-Reports/video-streaming-market-181135120.html. Last Visited: Sunday 12th July, 2020.

[124] Mei, Yuan and Cheng, Luwei and Talwar, Vanish and Levin, Michael Y and Jacques-Silva, Gabriela and Simha, Nikhil and Banerjee, Anirban and Smith, Brian and Williamson, Tim and Yilmaz, Serhat and others. Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[125] Alok Misra. Why Multitenancy Matters In The Cloud. https://www.informationweek.com/cloud/why-multitenancy-matters-in-the-cloud/d/d-id/1087206. Last Visited: Sunday 12th July, 2020.

[126] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, pages 245–256, 2003.

[127] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[128] Mor Naaman, Amy Xian Zhang, Samuel Brody, and Gilad Lotan. On the Study of Diurnal Urban Routines on Twitter. In *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media*, 2012.

[129] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proceedings of the 31st International Conference on Data Engineering (ICDE)*, pages 137–148, April 2015.

[130] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. When Two Choices are Not Enough: Balancing at Scale in Distributed Stream Processing. In *Proceedings of the 32nd International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, May 2016.

[131] Phuong Nguyen and Klara Nahrstedt. Resource Management for Elastic Publish Subscribe Systems: A Performance Modeling-Based Approach. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 561–568. IEEE, 2016.

[132] Phuong Nguyen and Klara Nahrstedt. Monad: Self-adaptive micro-service infrastructure for heterogeneous scientific workflows. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 187–196. IEEE, 2017.

[133] Ousterhout, Kay and Canel, Christopher and Ratnasamy, Sylvia and Shenker, Scott. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[134] C. M. Pazos, J. C. Sanchez Agrelo, and M. Gerla. Using back-pressure to improve tcp performance with many flows. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 2, pages 431–438 vol.2, 1999.

[135] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[136] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 49–49. IEEE, 2006.

[137] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 43–57, 2015.

[138] Dan RK Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 209–215, 2019.

[139] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.

[140] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. PerfOrator: Eloquent Performance Models for Resource Optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 415–427, 2016.

[141] Navaneeth Rameshan, Ying Liu, Leandro Navarro, and Vladimir Vlassov. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy. In *Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 233–244. IEEE, 2016.

[142] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 14–19, 2018.

[143] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[144] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.

[145] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards An Elastic Stream Computing Platform for the Cloud. In *Proceedings of the 4th International Conference on Cloud Computing*, pages 348–355. IEEE, 2011.

[146] Scott Schneider, Henrique Andrade, Buğgra Gedik, Alain Biem, and Kun-Lung Wu. Elastic Scaling of Data Parallel Operators in Stream Processing. In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE, 2009.

[147] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, volume 12, pages 349–362, 2012.

[148] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. In *Proceedings of the VLDB Endowment*, volume 8, pages 245–256. VLDB Endowment, November 2014.

[149] Nesime Tatbul, Yanif Ahmad, Uğur Çetintemel, Jeong-Hyon Hwang, Ying Xing, and Stan Zdonik. Load Management and High Availability in the Borealis Distributed Stream Processing Engine. *GeoSensor Networks*, pages 66–85, 2006.

[150] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[151] Yahoo! Storm Team. Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, 2015. Last Visited: Sunday 12th July, 2020.

[152] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[153] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. LaKe: An Energy Efficient, Low Latency, Accelerated Key-Value Store. *arXiv preprint arXiv:1805.11344*, 2018.

[154] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm @ Twitter. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 147–156. ACM, 2014.

[155] Tri Minh Truong, Aaron Harwood, Richard O Sinnott, and Shiping Chen. Performance Analysis of Large-Scale Distributed Stream Processing Systems on the Cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 754–761. IEEE, 2018.

[156] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[157] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[158] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 363–378, 2016.

[159] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, page 14. ACM, 2012.

[160] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[161] Wikipedia. Pareto Efficiency — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Pareto_efficiency&oldid=741104719, 2016. Last Visited Sunday 12th July, 2020.

[162] Gerhard J Woeginger. There is no asymptotic ptas for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.

[163] Kun-Lung Wu, Kirsten W Hildrum, Wei Fan, Philip S Yu, Charu C Aggarwal, David A George, Buğra Gedik, Eric Bouillet, Xiaohui Gu, Gang Luo, et al. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1185–1196. VLDB Endowment, 2007.

[164] Yingjun Wu and Kian-Lee Tan. Chronostream: Elastic Stateful Stream Computation in the Cloud. In *Proceedings of the 31st International Conference on Data Engineering*, pages 723–734. IEEE, 2015.

[165] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[166] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, pages 535–544. IEEE, 2014.

[167] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE, 2016.

[168] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[169] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[170] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. AWStream: Adaptive Wide-Area Streaming Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252, 2018.

[171] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 1049–1062, 2019.

[172] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 377–392, 2017.