Henge: Intent-driven Multi-Tenant Stream Processing

Faria Kalim
University of Illinois at Urbana Champaign
kalim2@illinois.edu

Le Xu
University of Illinois at Urbana Champaign
lexu1@illinois.edu

Sharanya Bathey
Amazon.com Services, Inc.
sharanyb@amazon.com

Richa Meherwal
phData, Inc.
richa@phdata.io

Indranil Gupta
University of Illinois at Urbana Champaign
indy@illinois.edu

ABSTRACT

We present Henge, a system to support intent-based multi-tenancy in modern distributed stream processing systems. Henge supports multi-tenancy as a first-class citizen: everyone in an organization can now submit their stream processing jobs to a single, shared, consolidated cluster. Secondly, Henge allows each job to specify its own intents (i.e., requirements) as a Service Level Objective (SLO) that captures latency and/or throughput needs. In such an intent-driven multi-tenant cluster, the Henge scheduler adapts continually to meet jobs' respective SLOs 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 in the system. Henge is integrated into Apache Storm and we present experimental results using both production jobs from Yahoo! and real datasets from Twitter.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**;

KEYWORDS

Stream Processing, Multi-Tenancy, Resource Management, Intents, Service Level Objectives

ACM Reference Format:
Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. 2018. Henge: Intent-driven Multi-Tenant Stream Processing. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 14 pages.
https://doi.org/10.1145/3267809.3267832

1 INTRODUCTION

Modern distributed stream processing systems use a cluster to process continuously-arriving data streams in real time, from Web data to social network streams. Multiple companies use Apache Storm [4] (Yahoo!/Oath, Weather Channel, Alibaba, Baidu, WebMD, etc.), Twitter uses Heron [54], LinkedIn relies on Samza [3] and others use Apache Flink [1]. These systems provide high-throughput and low-latency processing of streaming data from advertisement pipelines (internal use at Yahoo!), social network posts (LinkedIn, Twitter), geospatial data (Twitter), etc.

While stream processing systems for clusters have been around for decades [18, 35], neither classical nor modern distributed stream processing systems support *intent-driven multi-tenancy*. We tease apart these two terms. First, multi-tenancy allows multiple jobs to share a single consolidated cluster. Because this capability is lacking in stream processing systems today, many companies (e.g., Yahoo! [6]) 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, 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. In industry terms, multi-tenancy reduces capital and operational expenses (Capex & 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 [73], storage systems [81], batch processing [80], and others [58], yet it remains a vital need in modern stream processing systems.

Secondly, 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 continually monitoring their job and guessing how many resources it needs. Modern distributed stream processing systems are very primitive and do not admit intents.

We allow each job in a multi-tenant environment to specify its own intent as a Service Level Objective (SLO) [44]. It is critical that the metrics in an SLO 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 and scheduler. For instance, SLO metrics can capture latency and throughput expectations. SLOs should not include internal metrics like queue lengths or CPU utilization as these can vary depending on the software, cluster, and job mix[1].

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267832

[1] However, these latter metrics can be monitored and used internally by the scheduler for self-adaptation.

Business	Use Case	SLO Type & Value
Bloomberg	High Frequency Trading Updating top-k recent news articles on website Combining updates into one email sent per subscriber	Latency < Tens of ms Latency < 1 min. Throughput > 40K messages/s [64]
Uber	Determining price of a ride on the fly, identifying surge periods Analyzing earnings over time	Latency < 5 s Throughput > 10K rides/hr. [16]
The Weather Channel	Monitoring natural disasters in real-time Processing collected data for forecasts	Latency < 30 s Throughput > 100K messages/min. [10]
WebMD	Monitoring blogs to provide real-time updates Search indexing related websites	Latency < 10 min. Throughput: index new sites at the rate found
E-Commerce Websites	Counting ad-clicks Processing logs at Alipay	Latency: update click counts every second Throughput > 6 TB/day [15]

Table 1: Stream Processing: Use Cases and Possible SLO Types.

Schedulers	Jobs	Multi-Tenant?	User-Facing SLOs?
Mesos [41]	General	✓	✗ Uses Reservations: CPU, Mem, Disk, Ports
YARN [80]	General	✓	✗ Uses Reservations: CPU, Mem, Disk
Aurora [17]	Streaming	✓	✓ For Single Node Environment Only
Borealis [18]	Streaming	✗	✓ Latency, Throughput, Others
R-Storm [66]	Streaming	✗	✗ Schedules based on: CPU, Mem, Bandwidth
Henge	**Streaming**	✓	**✓ Latency, Throughput, Hybrid**

Table 2: Henge vs. Existing Multi-Tenant Schedulers.

We believe lay users should not have to grapple with such complex metrics.

While there are myriad ways to specify SLOs (including potentially declarative languages paralleling SQL), our paper is best seen as one contributing *mechanisms* that are pivotal for building a truly intent-based stream processing system. Our latency SLOs and throughput SLOs are immediately useful. Time-sensitive jobs (e.g., those related to an ongoing ad campaign) are latency-sensitive and will specify latency SLOs, while longer running jobs (e.g., sentiment analysis of trending topics) will have throughput SLOs. Table 1 summarizes several real use cases spanning different SLO requirements.

We present Henge, the first scheduler to support both multi-tenancy and per-job intents (SLOs) for modern stream processing engines. Henge has to tackle several challenges. In a cluster of limited resources, Henge needs to continually adapt to meet jobs' SLOs, both under natural system fluctuations, and input rate changes due to diurnal patterns or sudden spikes. While attempting to maximize the system's overall SLO achievement, Henge needs to prevent non-performing topologies from hogging cluster resources. It needs to scale with cluster size and jobs, and work well under failures.

Table 2 compares Henge with multi-tenant schedulers that are generic (Mesos, Yarn), as well as those that are stream processing-specific (Aurora, Borealis and R-Storm). Generic schedulers largely use reservation-based approaches to specify intents. A reservation is an explicit request to hold a specified amount of cluster resources for a given duration [29]. Besides not being user-facing, reservations are known to be hard to estimate even for a job with a static workload [45], let alone the dynamic workloads prevalent in streaming applications. Classical stream processing systems are either limited to a single node environment (Aurora), or lack multi-tenant implementations (e.g., Borealis has a multi-tenant proposal, but no associated implementation). R-Storm [66] is resource-aware Storm that adapts jobs based on CPU, memory, and bandwidth, but does not support user-facing SLOs.

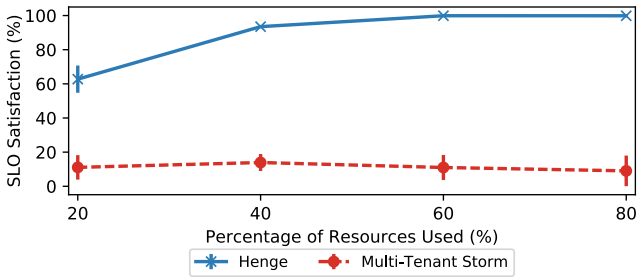


Fig. 1: Benefits of enabling Henge in Apache Storm.

Fig. 1 depicts the twin benefits from Henge: dollar cost savings from multi-tenancy, and higher SLO satisfaction. Concretely, in the workload considered (5 jobs), x=100% represents the minimum resources needed in a single-tenant version of Storm (each job gets its own cluster) to reach 100% utility, i.e., satisfy all SLOs. First, compared to single-tenant Storm, Henge can save 60% of resource costs (x=40%) and still reach 93.5% utility. It can satisfy 100% of SLOs at resource savings of 40% (x=60%). Second, compared to multi-tenant vanilla Storm, Henge achieves significantly higher SLO satisfaction (e.g., 6.7× better at x=40%). Considering that the streaming analytics market is expected to grow to $13.7 billion by 2021 [59], the 40%-60% dollar cost savings from Henge will have a significant impact on profit margins.

This paper makes the following contributions:

- We present Henge, the first scheduler for intent-based multi-tenancy in distributed stream processing.
- We define and calculate a new throughput SLO metric that is input-rate-independent, called "juice".
- We define SLOs via utility functions.
- We describe Henge's implementation in Storm [4].

- We evaluate Henge using workloads from Yahoo! production Storm topologies, and Twitter datasets.

2 HENGE SUMMARY

We now summarize the novel ideas in Henge.

Juice: As input rates vary over time, specifying a throughput SLO as an absolute value is impractical. We define a new *input rate-independent* metric for throughput SLOs called *juice*. We show how Henge calculates juice for arbitrary topologies (Section 6).

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 processed by the job. Conversely, a value less than 1 indicates that tuples are building up in queues, waiting to be processed.

Throughput SLOs 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 input tuples may be filtered or modified: thus, they affect results but are never outputted.

SLOs: A job's SLO can capture either latency or juice (or a combination of both). The SLO has: a) a threshold (min-juice or max-latency), and b) a job priority. Henge combines these via a user-specifiable *utility function*, inspired by soft real-time systems [52]. The utility function maps current achieved performance (latency or juice) to a value that represents the current benefit to the job. Thus, the function captures the developer intent that a job attains full "utility" if its SLO threshold is met and partial utility if not. Our utility functions are monotonic: the closer the job is to its SLO threshold, the higher its achieved maximum possible utility (Section 4).

State Space Exploration: Moving resources in a live cluster is challenging. It entails a state space exploration where every step has both: 1) a significant realization cost, as moving resources takes time and affects jobs, and 2) a convergence cost, since the system needs time to converge to steady state after a step. Henge adopts a *conservatively online* approach where the next step is planned, executed in the system, then the system is allowed to converge, and the step's effect is evaluated. Then, the cycle repeats. This conservative exploration is a good match for modern stream processing clusters because they are unpredictable and dynamic. Offline exploration (e.g. simulated annealing) is time consuming and may make decisions on a cluster using stale information (as the cluster has moved on). Conversely, an aggressive online approach will over-react to changes, and cause more problems than it solves.

The primary actions in our state machine (Section 5) 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 good configuration). Henge gives jobs additional resources proportionate to how congested they are. Highly intrusive actions like reduction are kept small in number and frequency.

Maximizing System Utility: Via these actions, Henge attempts to continually improve each individual job and converge it to its maximum achievable utility. Henge is amenable to different goals for the cluster: either maximizing the minimum utility across jobs, or maximizing the total achieved utility across all jobs. While the former focuses on fairness, the latter allows more cost-effective use of the cluster, which is especially useful since revenue is associated with total utility of all jobs. Thus, Henge adopts the goal of maximizing total achieved utility summed across all jobs. Our approach creates a weak form of Pareto efficiency [83]; in a system where jobs compete for resources, Henge transfers resources among jobs only if this will cause the total cluster's utility to rise.

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 topologies that: a) are farthest from their SLOs, and b) continue to show utility improvements due to recent Henge actions. This spreads resources across all wanting jobs and mitigates starvation and resource hogging.

3 BACKGROUND

We describe our system model, fitting modern stream processing engines like Storm [79], Heron [54], Flink [1] and Samza [3]. Stream processing jobs are long-running. A stream processing job can be logically interpreted as a *topology*, i.e., a directed acyclic graph of *operators* (called *bolts* in Apache Storm)[2]. An operator is a logical processing unit that applies user-defined functions on a stream of *tuples*. Source operators (called spouts) pull input tuples from an external source (e.g., Kafka [53]), while sink operators spew output tuples (e.g., to affect dashboards). The sum of all spouts' input rates is the *input rate of the topology*, while the sum of all sinks' output rates is the *output rate of the topology*. Each operator is parallelized via multiple *tasks*. Fig. 2 shows an example topology.

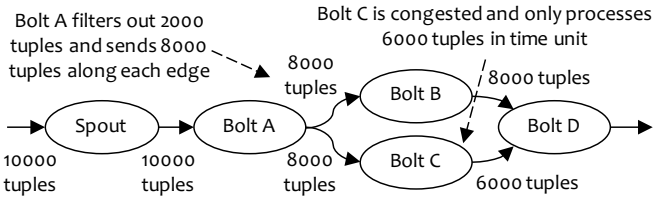


Fig. 2: Sample Storm topology. *Tuples processed per unit time. Edge labels indicate number of tuples sent out by the parent operator to the child. (Congestion described in Section 5.)*

A topology is run on worker processes executing on each machine in the cluster. Each worker process runs *executors* (threads) which run tasks specific to one operator. Each thread processes streaming data, one tuple at a time, forwarding output tuples to its child operators. Tuples are shuffled among threads (our system is agnostic).

Definitions: An outputted tuple *O*'s *latency* is the time between it being outputted at the sink, and the arrival time of the latest input tuple that directly influenced it. A *topology's latency* is then the average latency of tuples it outputs over a window of time. A *topology's throughput* is the number of tuples processed per unit time.

4 SLOS AND UTILITY FUNCTIONS

A *Service Level Objective (SLO)* [13] in Henge is user-facing and can be set without knowledge of internal cluster details. An SLO specifies: a) an SLO *threshold* (min-throughput or max-latency);

[2]We use the terms job and topology interchangeably in this paper.

and b) a job priority. Henge girds these requirements together into a *utility function*.

Utility functions are commonplace in distributed systems [57, 70, 89, 90]. In Henge, each job can have its own utility function—the user can either reuse a stock utility function (like the knee function we define below), or define their own. Henge's utility function for a job maps the current performance of the job to a *current utility* value. Utility functions are attractive as they abstract away the type of SLO metric that each topology has. Our approach allows Henge to manage in a compatible manner a cluster with jobs of various SLO types, SLO thresholds, priorities, and utility functions.

Currently, Henge supports both latency SLOs and throughput SLOs (and hybrids thereof). (While Section 6 discusses juice as an input-rate-independent throughput metric, here we assume an arbitrary throughput metric.)

A utility function can also capture partial utility when the SLO requirement is not being met. Our only requirement for utility functions is that they must be *monotonic*. This captures more utility closer to the SLO threshold. For a job with a latency SLO, the utility function must be monotonically non-increasing as latency rises. For a job with a throughput SLO, it must be monotonically non-decreasing as throughput rises.

A utility function typically has a *maximum utility*, achieved when the SLO threshold is met, e.g., a job with an SLO threshold of 100 ms achieves maximum utility only if its latency is below 100 ms. To stay monotonic, as latency grows above 100 ms, the utility function can drop or plateau, but not rise.

The maximum utility value is based on (e.g., proportional to) job priority. For instance, in Fig. 3a, topology T2 has twice the priority of T1, and thus has twice the maximum utility (20 vs. 10). We assume that priorities are fixed at submission time—although Henge can generalize (Section 5.5).

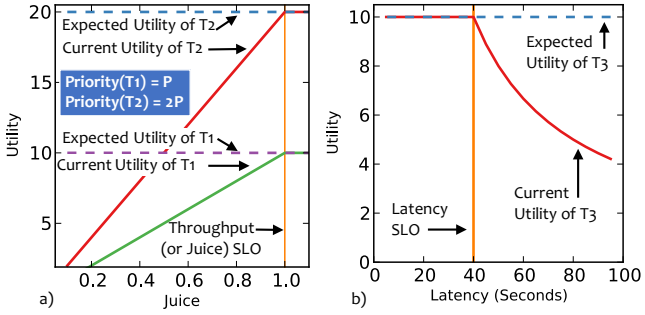


Fig. 3: Knee Utility functions for: *(a) throughput, (b) latency.*

Given these requirements, Henge allows a variety of utility functions: linear, piece-wise linear, step functions, lognormal, etc. Utility functions may not be continuous.

Users can pick any utility functions that are monotonic. For concreteness, our Henge implementation uses a piece-wise linear utility function called a *knee* function. A knee function (Fig. 3) has two parts: a plateau after the SLO threshold, and a sub-SLO for when the job does not meet the threshold. Concretely, the achieved utility for jobs with throughput and latency SLOs respectively, are:

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

$$\frac{\texttt{Current Utility}}{Job\ Max\ Utility} = min(1, \frac{SLO\ Latency\ Threshold}{Current\ Latency}) \quad (2)$$

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. 3 shows an example of a throughput SLO (Fig. 3a) and a latency SLO (Fig. 3b).

A utility function approach can also be used in Service Level Agreements (SLAs), but these are beyond our scope.

5 HENGE STATE MACHINE

Henge uses a state machine for the entire cluster (Fig. 4). A cluster is *Converged* if and only if either: a) all topologies have reached their max utility (i.e., satisfy their SLO thresholds), or b) Henge recognizes that no further actions will improve any topology's performance, and thus it reverts to the last best configuration. The last clause guarantees that if external factors remain constant–topologies are not added and input rates stay constant–Henge will converge. All other cluster states are *Not Converged*.

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

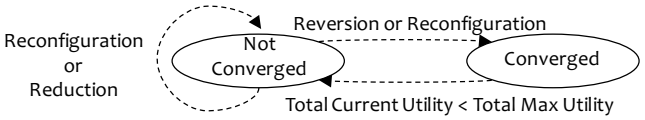


Fig. 4: Henge's State Machine for the Cluster.

5.1 Reconfiguration

In the Not Converged state, a Reconfiguration action gives resources to topologies missing their SLO. Reconfigurations occur in *rounds*, executed periodically (currently 10 s). In each round, Henge sorts all topologies missing their SLOs, in descending order of their maximum utility, with ties broken by preferring lower current utility. It then greedily picks the head of this sorted queue to allocate resources to. The performance of a chosen topology can be improved by giving more resources to all of its operators that are *congested* i.e., those that have insufficient resources, and thus are bottlenecks.

Measuring Congestion: We use operator *capacity* [14] to measure congestion. (Henge is amenable to using other congestion metrics, e.g., input queue sizes or ETP [87].) Operator capacity captures the fraction of time that an operator spends processing tuples in 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 tasks of a topology operator. Its capacity is calculated as:

$$Capacity_E = \frac{Executed\ Tuples_E \times Execute\ Latency_E}{Unit\ Time} \quad (3)$$

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.

Dividing the numerator by the time window tells us what fraction of time the executor was busy processing tuples. The operator capacity is then the maximum capacity across all of its executors. Henge considers an operator to be congested if its capacity is above the threshold of $Operator\ Capacity\ Threshold = 0.3$. This low value increases the pool of possibilities, as more operators become candidates for receiving resources.

Allocating Thread Resources: Henge allocates each congested operator an additional number of threads that is proportional to its congestion level based on the following equation:

$$\left(\frac{Current\ Operator\ Capacity}{Operator\ Capacity\ Threshold} - 1\right) \times 10 \quad (4)$$

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

A topology may improve only marginally after receiving more resources in a reconfiguration. If a job's utility increases by less than a factor $(1 + \Delta)$, Henge retains the reconfiguration but skips this particular topology in the near future rounds. $\Delta = 5\%$ in our implementation. This topology may have plateaued in terms of benefiting from more threads. As the cluster is dynamic, the topology's blacklisting is allowed to expire after a while (1 hour in our settings), after which the topology again becomes a candidate for reconfiguration.

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

5.2 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 all of these conditions are true: (a) the cluster is congested (described below), (b) there is at least one SLO-satisfying topology, and (c) there is no past history of a Reduction action. Note that (c) implies that there is at most one Reduction (until an external factor such as workload changes) cluster-wide.

CPU load is defined as the number of processes that are running or runnable on a machine [7]. A machine's load should be less than or equal to the number of available cores, ensuring maximum utilization and no over-subscription. Thus, Henge considers a *machine to be congested* if its CPU load exceeds its number of cores. Henge considers a *cluster to be congested* when a majority of its machines are congested.

If a Reconfiguration drops utility and results in a congested cluster, Henge executes Reduction to reduce congestion. For all topologies *meeting* their SLOs, Henge finds all their un-congested operators (except spouts) and reduces their parallelism by a large amount (80% in our settings). If this results in further SLO misses, such topologies will be candidates in future reconfigurations to improve their SLO.

Henge limits Reduction to once per cluster to minimize intrusion; this is reset if external factors change (new jobs are added or input rate changes etc.).

Like backoff mechanisms [42], massive reduction is the only way to free many resources at once for reconfigurations. Fewer threads also make the system more efficient because they reduce the number of OS context switches.

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 is futile. This is a typical "lockout" case, and Henge resolves it by Reversion.

5.3 Reversion

If a Reconfiguration drops utility and a Reduction is not possible (meaning that at least one of the conditions (a)-(c) in Section 5.2 is false), 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 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 new workload, Henge forgets all history of past actions and moves to the Not Converged state. In future reversions, such forgotten states will not be available. This reset allows Henge to start its state space search afresh.

5.4 Proof of Convergence

We prove the convergence of Henge. As is usually necessary in proofs of real implementation, we make some assumptions. Our implementation and subsequent evaluation in Section 8 removes all assumptions, and demonstrates that Henge works well in practice.

Recall that a *converged state* is one where Henge can take no further actions (Fig. 4). We define a *well-behaved cluster* as one where: (a) a Reduction action on a job (Section 5.2) does not cause its SLO to be violated (recall that reductions are only done on jobs already satisfying their SLO), and (b) between consecutive Henge steps, the performance of each topology (throughput and/or latency), measured via its utility, remains stable.

THEOREM 1. *Given any initial state on a well-behaved cluster with no job arrivals or departures, and sufficiently long black-listing, Henge converges in finite steps.*

PROOF. Consider a cluster with N jobs (topologies), $J_1 \ldots J_N$. Henge ensures that in each step of its state machine, at least one topology is affected. This effect may be one of four kinds—(i) job J_i is reconfigured (Section 5.1) and its utility improves by at least [3] a factor of $(1 + \Delta)$, or (ii) J_i is reconfigured and its utility improves by less than a factor of $(1 + \Delta)$, or (iii) the cluster is subjected to a

[3] See definition of Δ in Section 5.1. In our implementation $\Delta = 5\%$.

reduction (Section 5.2), or (iv) the cluster is subjected to a reversion (Section 5.3).

Consider the set $\mathcal{S}$ of topologies *not* satisfying their SLO. We claim that the cardinality of $\mathcal{S}$ is *monotonically non-increasing* as long as steps under only cases (i)-(iii) are applied. We explain why. A case (i) step can only remove from (never add to) $\mathcal{S}$, if and only if this step causes J_i to meet its SLO. A case (ii) step black-lists J_i (Section 5.1) and removes it from $\mathcal{S}$—as we assumed black-listing lasts long enough, the monotonic property holds. Due to our well-behaved cluster assumption (a), the monotonic property holds after a reduction operation (case (iii)). Lastly, in between consecutive steps, cardinality of $\mathcal{S}$ does not increase due to our well-behaved cluster assumption (b).

Now consider case (i). For job i, let $Init\_Util(i)$ be its initial utility, and $Max\_Util(i)$ be its maximum achievable utility (at SLO threshold). Let:

$$F = Max_{i=1}^{N} \left\{ \frac{Max\_Util(i)}{Init\_Util(i)} \right\}$$

The initial number of jobs missing their SLOs is $\leq N$. For a topology J_i that converges to its SLO, J_i is subjected to at most $log_{(1+\Delta)}(F)$ reconfigurations under case (i) (until it meets its SLO). For a topology J_i that becomes black-listed, it is subject to strictly $< log_{(1+\Delta)}(F)$ reconfigurations under case (i), along with an additional (last) reconfiguration under case (ii). Together, any job that either meets its SLOs or becomes black-listed, is subject to at most $log_{(1+\Delta)}(F)$ steps under cases (i) or (ii).

Cluster-wide, there is at most one case (iii) reduction step (Section 5.2). Finally, reversion in case (iv) is, if present, always the last step in the entire cluster.

Putting these together, the total number of steps required for Henge to converge is $\leq (N \cdot log_{(1+\Delta)}(F) + 1 + 1)$ steps. Thus, Henge converges in finite steps. □

Clause (b) in the well-behaved cluster assumption can be relaxed to restate the proof (and theorem), by amending F to also include the max of $\frac{Max\_Util(i)}{Red\_Util(i)}$ for all jobs J_i whose SLO is violated due to a reduction, where $Red\_Util(i)$ is J_i's utility right after its reduction.

5.5 Discussion

Online vs. Offline State Space Search: Henge uses an online state space search. In fact, our early attempt at designing Henge was to perform offline state space exploration, by using analytical models to find relations of SLO metrics to job resource allocations.

However, the offline approach turned out to be impractical. Analysis and prediction is complex and inaccurate for stream processing systems, which are very dynamic in nature. (This phenomenon has also been observed in other distributed scheduling domains, e.g., [24, 45, 65].) Fig. 5 shows two identical runs of the same Storm job on 10 machines, where the job is given 28 extra threads at t=910 s. Latency drops to a lower value in run 2, but stays high in run 1. This is due to differing CPU resource consumptions across the runs. 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. This motivated us to adopt Henge's conservative online approach.

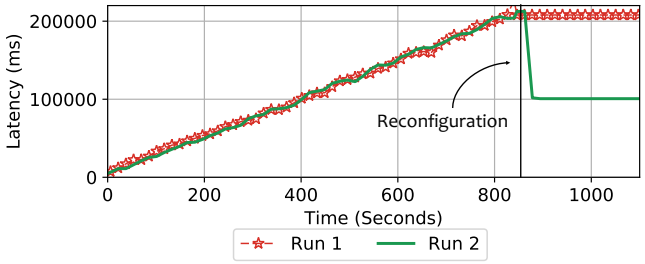


Fig. 5: Unpredictability in Modern Stream Processing Engines: *Two runs of the same topology (on 10 machines) being given the same extra computational resources (28 threads, i.e., executors) at 910 s, react differently.*

Tail Latencies: Henge can also support SLOs expressed as tail latencies (e.g., 95th or 99th percentile). Utility functions are then expressed in terms of tail latency.

Statefulness, Memory Bottlenecks: In most production use cases, we have observed that operators are stateless and CPU-bound, and our exposition so far is thus focused. Even so, Henge gracefully handles stateful operators and memory-pressured nodes (Sections 8.3, 8.5). Stateful jobs can rebuild state after a reconfiguration as we periodically checkpoint state to an external database. The orthogonal problem of dynamically reconfiguring stateful operators is beyond our scope and is an open direction [26].

"Magic" Parameters: Henge uses five parameters: capacity threshold (Section 5.1), stability rounds (Section 5.1), improvement Δ in utility for reconfiguration (Section 5.1), percentage reduction in executors (Section 5.2), and percentage utility drop to move out of converged state (Section 5.3). All of these parameters only affect Henge's convergence speed, not correctness.

Four of these five parameters have little effect on performance as long as their values stay small. The percentage reduction in executors may require minimal profiling. In our cluster, reducing a topology's executors by 90% caused SLO misses, while a 70% setting did not alleviate CPU load much. Thus, we set this parameter to 80% in our implementation, for all experimental workloads.

Henge as Indicator of Resource Crunches: While we do not perform admission control, Henge can nonetheless be used to identify topologies that are overly hungry (e.g., high priority jobs that are touched by many reconfiguration steps), and these could be black-listed or removed. Additionally, Henge could be used as an indicator that the cluster is oversubscribed and that additional cluster resources are needed, e.g., many SLO violations, cluster utility rarely approaches max utility, etc.

Setting Priorities: A consequence of priorities is that Henge will prefer higher priority jobs at the expense of penalizing lower priority jobs, especially when the cluster is congested. We recommend that any jobs that wish to absolutely avoid starvation come associated with medium to high priorities. This is feasible as we envision Henge to be used internally in companies where job priorities are set either consensually or by upper management.

Dynamic Priorities: While we have assumed for simplicity that a job's priority is fixed at its submission time (Section 4), Henge would still work if priorities were changed on the fly. Henge's state

machine and utility function approach are compatible with dynamic priorities.

6 JUICE: DEFINITION AND ALGORITHM

As stated in Section 1, we wish to design a metric for throughput SLOs that is independent of input rate. Henge uses a new metric called *juice*. Juice defines the fraction of input data that is processed by the topology per unit time. It lies in the interval [0, 1]. A value of 1.0 means all the input data that arrived in the last time unit has been processed. Thus, users can set throughput requirements as percentages of input rate (Section 4). Henge then attempts to maintain this even as input rates change.

Any algorithm that calculates juice should be:

1. Code Independent: 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 Independent: It should be input-rate independent.

3. Topology Independent: It should be independent of the shape and structure of the topology. It should be correct in spite of duplication, merging, and splitting of tuples.

Juice Intuition: Juice is formulated to reflect the *global* processing efficiency of a topology. An operator's contribution to juice is the proportion of input passed in originally *from the source (i.e., from all spouts)* that it processed in a given time window. This is 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). *Source input* tuples are those that arrive at a spout in unit time. For each operator o in a topology that has n parents, we define T_o^i as the sum 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 spout 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:

$$J_o^s = \sum_{i=1}^{n} \left(J_i^s \times \frac{E_o^i}{T_o^i} \right) \tag{5}$$

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

In eq. 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 are 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 spouts:

$$\frac{\sum_{Sinks\ s_i,\ Spouts\ s_j} (J_{s_i}^{s_j})}{Total\ Number\ of\ Spouts} \tag{6}$$

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

Example 1: Consider Fig. 2 in Section 3. $J_A^s = 1 \times \frac{10K}{10K} = 1$ 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.

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 and 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 (Topology Juice with Duplication, Split, and Merge): Due to space, we refer the reader to our tech report [46] for the example.

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 10 s. This needs only loose time synchronization across nodes. Second, collecting data in sub-windows implies that when short-lived bursts occur, Henge does not over-react. and instead adjusts resources in a conservative manner. Concretely, it reacts only if the burst is long-lived and stretches across multiple sub-windows. Third, eq. 6 treats all processed tuples equally–instead, a weighted sum could be used instead to capture differing priorities among sinks. Fourth, processing guarantees (exactly, at least, at most once) are orthogonal to juice. Our experiments use non-acked Storm (at most once semantics), but Henge also works with acked Storm (at least once semantics).

7 IMPLEMENTATION

We integrated Henge into Apache Storm [4]. Henge involves 3800 lines of Java code. It is an implementation of the predefined IScheduler interface. The scheduler runs in the Storm Nimbus daemon, and is invoked by Nimbus every 10 seconds. We updated Storm Config to let users set SLOs and utility functions for their topologies.

Fig. 6 shows Henge's architecture. The Decision Maker implements the Henge state machine (Section 5). The Statistics Module continuously calculates cluster and per-topology metrics e.g., the number of tuples processed per task of an operator per topology, end-to-end tuple latencies, and the CPU load per node. This information is used to produce 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 past states so that reversion can be performed.

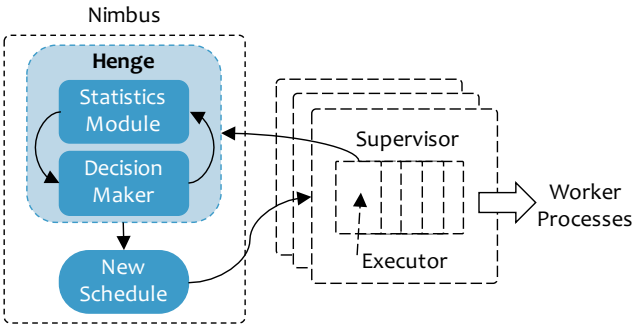


Fig. 6: Henge Implementation in Storm

8 EVALUATION

We evaluate Henge with several workloads, topologies, and SLOs. Further results are in our tech report [46].

Experimental Setup: By default, our experiments used the Emulab cluster [82], with machines (2.4 GHz, 12 GB RAM) running Ubuntu 12.04 LTS, over a 1 Gbps network. A machine runs Zookeeper [5] and Nimbus. Workers (Java processes running executors) are allotted across 10 machines (we also evaluate scalability).

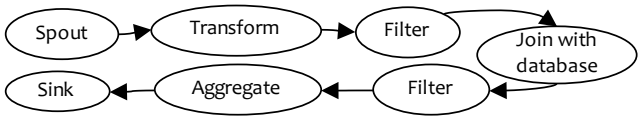


Fig. 7: PageLoad Topology from Yahoo!.

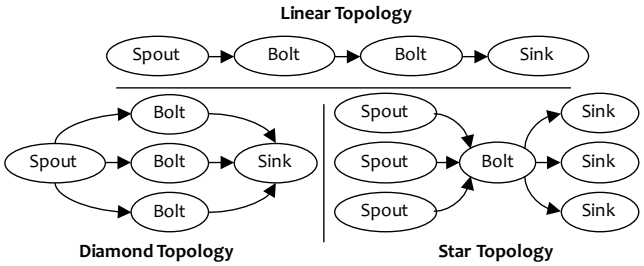


Fig. 8: Three Microbenchmark Storm Topologies.

Topologies: We use a combination of production and microbenchmark topologies. Production topologies include the "PageLoad" topology from Yahoo! Inc. [87] (shown in Fig. 7), and WordCount. Its operators are the most commonly used in production: filtering, transformation, aggregation. For some evaluations, we also use microbenchmark topologies (shown in Fig. 8) that are possible sub-parts of larger topologies [87]; these are used in Section 8.4.1, Fig. 16, and Section 8.5.

These topologies are representative of today's workloads. All topologies for modern stream processing engines that we have encountered in production, and in literature [6, 77], are linear (Fig. 7) or at best linear-like (like Fig. 8). While Henge is built to handle arbitrarily complex topologies (as long as they are DAGs), for practicality, we use the topologies just discussed.

The latency SLO thresholds in our experiments are in order of milliseconds. Thus, we subject Henge to more constraints and stress than the values listed in Table 1.

In each experimental run, we initially let topologies 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 a knee utility function (Section 4). A knee function is parameterized by the SLO threshold and the maximum utility value. Hence, below we use "SLO" as a shorthand for the SLO threshold, and specify the max utility value.

Storm Configuration: In experiments where we compare Henge with Storm, we used Storm's default scheduler and began the Storm run with identical configuration (e.g., cluster size, number of executors) as the Henge run.

8.1 Henge Policy and Scheduling

8.1.1 *Meeting SLOs.*

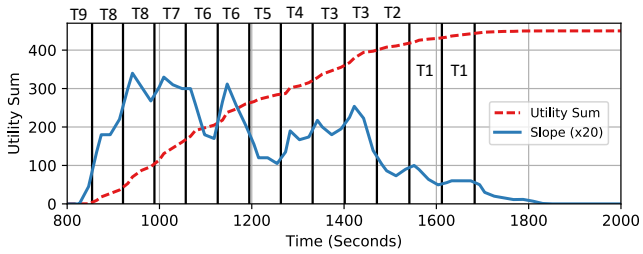


Fig. 9: Maximizing Cluster Utility: *Red (dotted) line is total system utility. Blue (solid) line is magnified slope of the red line. Vertical lines are reconfigurations annotated by the job touched. Henge reconfigures higher max-utility jobs first, leading to faster increase in system utility.*

Maximizing Cluster Utility: To maximize total cluster utility, Henge greedily prefers to reconfigure those jobs first which have a higher max achievable utility (among those missing their SLOs). In Fig. 9, 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 jobs is 60 ms. Henge first picks T9 (highest max utility of 90), leading to a sharp increase in total cluster utility at 950 s. Then, it continues in this greedy way. We observe some latent peaks when jobs 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 jobs meet their SLO in 15 minutes (900 s to 1800 s).

Hybrid SLOs: We evaluate a topology with a hybrid SLO that has separate thresholds for latency and juice, and two corresponding utility functions (Section 4) with identical max utility values. The job's utility is then the average of these two utilities.

Fig. 10 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 of each topology is 35. Henge takes about 13 minutes (t=920 s to t=1710 s) to reconfigure all topologies successfully to meet their

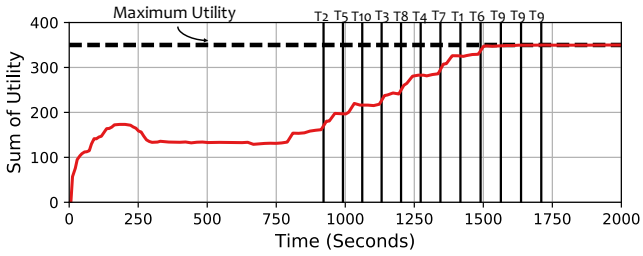


Fig. 10: Hybrid SLOs: *Henge Reconfiguration.*

SLOs. 9 out of 10 topologies required a single reconfiguration, and one (T9) required 3 reconfigurations.

Is convergence fast enough? Stream processing jobs run for long periods. Practically, Henge's convergence times are relatively quite small. We ran 10 topologies over 48 hours with a diurnal workload (details in next section). The *total* time taken by Henge's actions, summed across all topologies, was 2.4 hours. This is only 0.5% of the total runtime summed across all topologies. The longest convergence time (for any topology) was 16 minutes, which is only 0.56% of the total runtime.

8.1.2 Handling Dynamic Workloads.

A. Spikes in Workload: Fig. 11 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 ×, starting from 3600 s. The spike lasts until 7200 s when A's spike (also 2 ×) begins. Each topology's SLO is 80 ms with max utility is 35. Fig. 11 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 hides the effect of the input rate spike from the user.

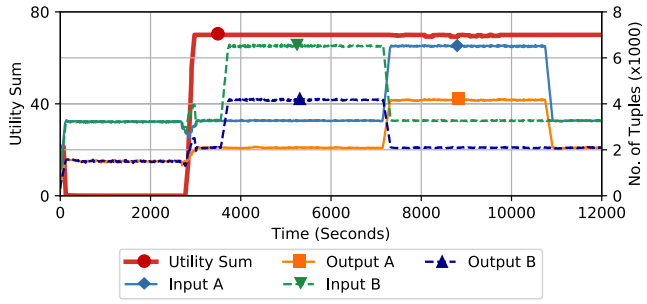


Fig. 11: Spikes in Workload: *Left y-axis shows total cluster utility (max possible is* $35 \times 2 = 70$*). Right y-axis shows the variation in workload as time progresses.*

B. Diurnal Workloads: Diurnal workloads are common for stream processing, e.g., in e-commerce websites [30] and social media [61]. We generated a diurnal workload based on the distribution of the SDSC-HTTP [12] and EPA-HTTP traces [9], injecting them into PageLoad topologies. 5 jobs run with the SDSC-HTTP trace and concurrently, 5 other jobs run with the EPA-HTTP trace. All jobs have max-utility=35, and a latency SLO of 60 ms.

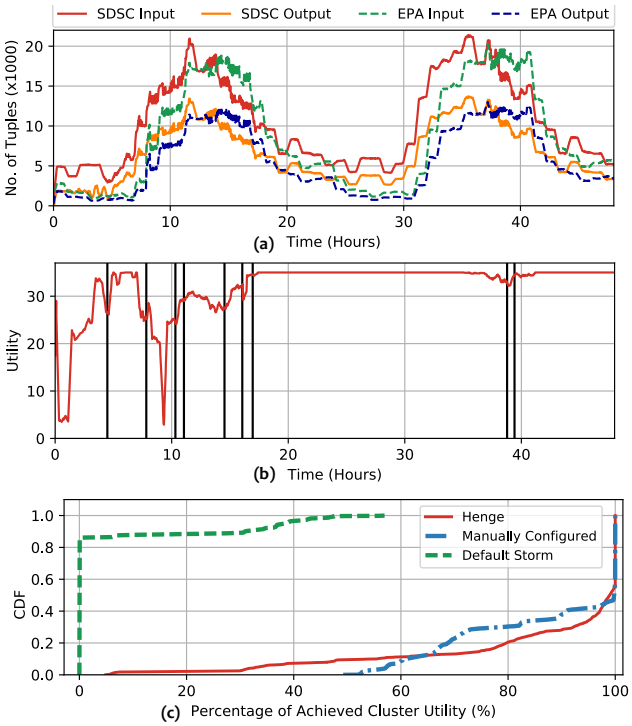


Fig. 12: Diurnal Workloads: *a) Input and output rates vs time, for two diurnal workloads. b) Utility of job (reconfigured by Henge) with EPA workload, c) CDF of SLO satisfaction for Henge, default Storm, & manual configuration. Henge adapts during first cycle and fewer reconfigurations are needed later.*

Fig. 12 shows the result of running 48 hours of the trace (each hour is mapped to 10 mins). In Fig. 12a, workloads increase from hour 7 of day 1, reach their peak by hour $13\frac{1}{3}$, and then fall. Henge reconfigures all 10 jobs, reaching 89% of max cluster utility by hour 15.

Fig. 12b shows a topology running the EPA workload (other topologies exhibited similar behavior). Observe how Henge reconfigurations from hour 8 to 16 adapt to the fast changing workload. This results in fewer SLO violations during the second peak (hours 32 to 40). Thus, Henge tackles diurnal workloads without extra resources.

Fig. 12c shows the CDF of SLO satisfation for three systems. Default Storm gives 0.0006% SLO satisfaction at the median, and 30.9% at the 90th percentile (meaning that 90% of the time, default Storm provided 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 preferable over manual configurations. We manually configured all topologies to meet their SLOs at median load. They 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 later.

Henge has an average of 88.12% SLO satisfaction rate, while default Storm and manually configured topologies provide an average of 4.56% and 87.77% respectively. Thus, Henge gives 19.3× better

SLO satisfaction than default Storm and does better than manual configuration.

8.2 Production Workloads

We configured the sizes of 5 PageLoad topologies based on a Yahoo! Storm production cluster and Twitter datasets [20], shown in Table 3. We use 20 nodes with 14 worker processes. For each topology, we use an input rate proportional to its number of workers. T1-T4 run sentiment analysis on Twitter workloads [20]. T5 processes logs at a constant rate. Each topology has a latency SLO threshold of 60 ms and max utility 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

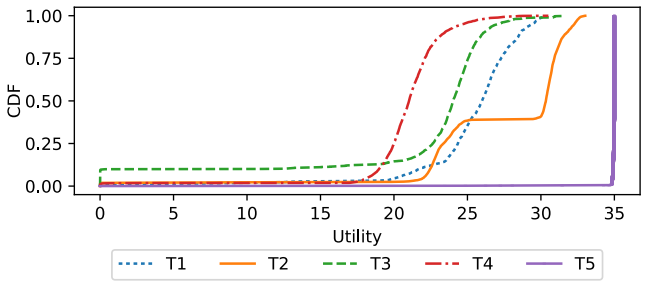
Table 3: Job and Workload Distributions in Experiments: *Derived from Yahoo! production clusters, using Twitter Datatsets for T1-T4. (Results in Figure 13.)*

This is an extremely constrained cluster where not all SLOs can be met. Yet, Henge improves cluster utility. Fig. 13a 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 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).

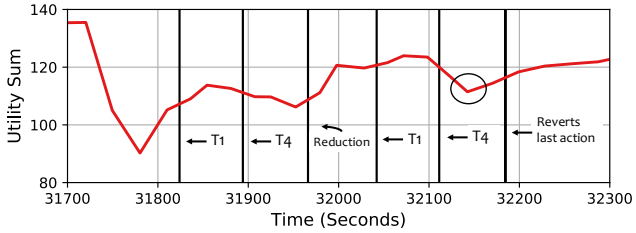
Reversion: Fig. 13b depicts Henge's reversion. At 31710 s, the system utility drops due to natural system fluctuations. This forces Henge to reconfigure two topologies (T1, T4). Since system utility continues to drop, Henge is forced to reduce a topology (T5), which satisfies its SLO before and after 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.

8.2.1 Reacting to Natural Fluctuations.

Natural fluctuations can occur in a cluster due to load variations that arise from interfering processes, disk IO, page swaps, etc. Fig. 14 shows such a scenario. We run 8 PageLoad topologies, 7 of which have an SLO of 70 ms, and the 8th's 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). Due to the significant drop in cluster utility, Henge reduces two SLO-achieving topologies. The reduction lets other topologies recover within 20 minutes (by 23000 s). Henge converges the cluster to the same total utility as before the CPU spike.



(a) *CDF of fraction of total time that tenant topologies achieve given SLO. Max utility for each topology is 35.*



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

Fig. 13: Henge on Production Workloads.

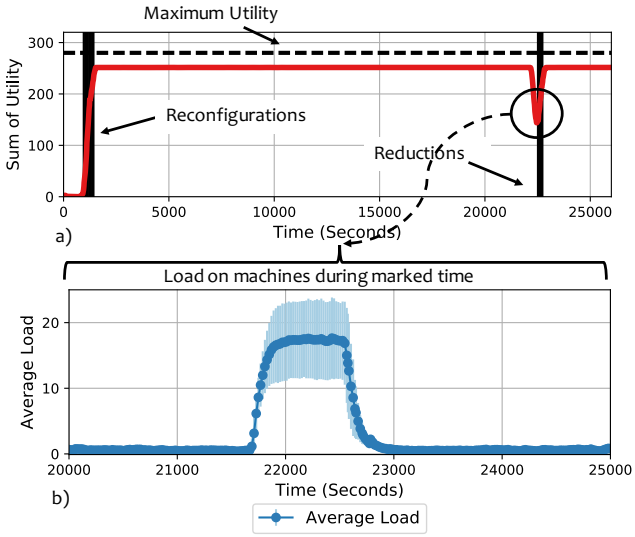


Fig. 14: Handling CPU Load Spikes: *a) Total cluster utility. b) Average CPU load on machines in CPU spike interval.*

8.3 Stateful Topologies

Henge handles stateful topologies gracefully. We ran four Word-Count topologies with identical workload and configuration as T2 in Table 3. Stateful topologies periodically checkpoint state to Redis 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 them meet their SLOs. Table 4 shows results after

Job Type	Avg. Reconfig. Rounds (Stdev)	Average Convergence Time (Stdev)
Stateful	5.5 (0.6)	1358.7s (58.1s)
Stateless	4 (0.8)	1134.2s (210.5s)

Table 4: Stateful Topologies: *Convergence Rounds and Times for a cluster with Stateful and Stateless Topologies.*

convergence. Stateful topologies take 1.5 extra reconfigurations to converge to their SLO, and 19.8% more reconfiguration time. This is due to checkpointing and recovery mechanisms, orthogonal to Henge.

8.4 Scalability and Fault-tolerance

8.4.1 *Scalability.*

Increasing the Number of Topologies: Fig. 15 stresses Henge by overloading the cluster with topologies over time. We start with 5 PageLoad jobs (latency SLO=70 ms, max utility=35), and add 20 more jobs at even hours.

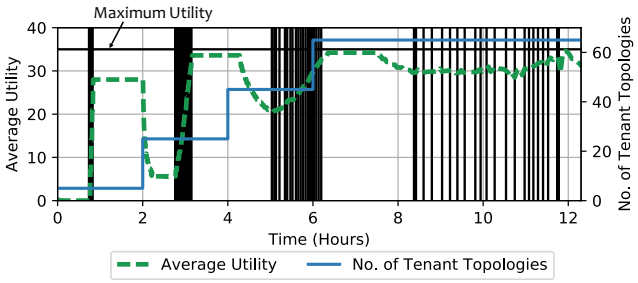


Fig. 15: Scalability w.r.t. No. of Topologies: *Cluster has 5 tenants. 20 tenants added every 2 hours until 8 hour mark. Green dotted line is average job utility. Blue solid line is number of jobs on cluster. Vertical black lines are reconfigurations.*

Henge stabilizes better when there are more topologies and a larger state space. With fewer topologies (first 2 hours) there is less state space to maneuver in, and hence the average utility stays low. 20 new tenant topologies at t=2 hours drop average utility but also open up the state space more–Henge quickly converges the cluster to the max utility value. We observe the same behavior when more jobs arrive at t=4, 6, and 8 hours.

Henge also tends to do fewer reconfigurations on older topologies than on new ones–see our tech report [46].

Increasing Cluster Size: In Fig. 16, we run 40 topologies on clusters of 10 to 40 machines (in our production use case studies, very rarely did we see cluster sizes exceeding 40 nodes). Each machines has 8-cores, 64 GB RAM, and a 10 Gbps interconnect. Topologies (with SLO, max utility) are: 20 PageLoads (80 ms, 35), 8 Diamond (juice=1.0, 5), 6 Star (1.0, 5), 6 Linear (1.0, 5).

As cluster size increases from an overloaded 10 nodes to 20 nodes, time to converge drops by 50%, and plateaus. At 10 nodes, only 40% of jobs meet SLO thresholds, and at the 5th percentile, jobs achieve only 0.3% of their max utility. At 20, 30, and 40 nodes, 5th percentile SLO satisfactions are 56.4%, 74.0% and 94.5% respectively. We

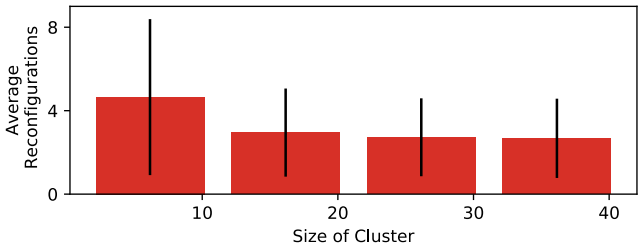


Fig. 16: Scalability w.r.t No. of Machines: *40 jobs run on cluster sizes increasing from 10 to 40 nodes. The bars show number of reconfigurations until convergence.*

point the reader to our tech report for details [46]. Overall, Henge's performance generally improves with cluster size, and overheads scale independently.

8.4.2 *Failure Recovery.*

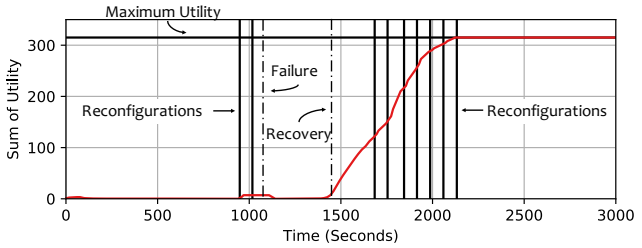


Fig. 17: Fault-tolerance: *Failure at t=1020s & recovery at t=1380 s. Henge makes no wrong decisions due to failure, and immediately converges to max system utility after recovery.*

Henge reacts gracefully to failures. In Fig. 17, we run 9 PageLoad topologies each with 70 ms SLO and 35 max utility. We introduce a failure at the worst possible time: during Henge's reconfiguration operations, at 1020 s. This severs communication between Henge and all worker nodes; Henge's Statistics module is unable to obtain fresh job information. Henge reacts conservatively by avoiding reconfiguration in the absence of data. At 1380 s, when communication is restored, Henge collects performance data for 5 minutes (until 1680 s) and then proceeds with reconfigurations until it meets all SLOs.

8.5 Memory Utilization

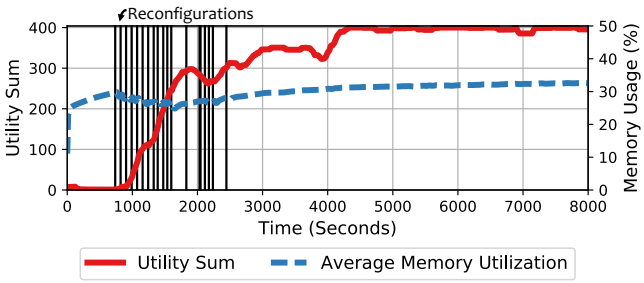


Fig. 18: Memory Utilization: ***8 jobs with joins and 30 s tuple retention.***

Fig. 18 shows a cluster with 8 memory-intensive micro-benchmark topologies (latency SLO=100 ms, max utility=50). These topologies

contain joins where tuples are retained for 30 s, creating memory pressure at some nodes. The figure shows that Henge reconfigures quickly to reach total max utility of 400 by 2444s, keeping average memory usage below 36%. Critically, the memory utilization (blue dotted line) plateaus in the converged state, showing that Henge handles memory-bound topologies gracefully.

9 RELATED WORK

Stream Processing: Among classical stream processing systems, Aurora [17] performs OS-level scheduling of threads and drops tuples in order to provide QoS for multiple queries [25]. However, Aurora's techniques do not apply or extend to a cluster environment. Borealis [18] is a distributed stream processing system that supports QoS, but is not multi-tenant. It orchestrates inflowing data, and optimizes coverage to reduce tuple-shedding. Borealis, as implemented, does not tackle the hard problems associated with multi-tenancy.

Modern stream processing systems [1–4, 11, 19, 54, 60] do not natively handle adaptive elasticity. Ongoing work [8] on Spark Streaming [88] allows scaling but does not apply to resource-limited multi-tenant clusters. [26, 68] scale out stateful operators and checkpoint, but do not scale in or support multi-tenancy.

Resource-aware elasticity in stream processing [23, 27, 34, 47, 50, 66, 71] assumes infinite resources that a tenant can scale out to. [21, 36, 43, 55, 56, 72, 84] propose resource provisioning but not multi-tenancy. Some works have focused on balancing load [62, 63, 75], optimal operator placement [38, 49, 67] and scaling out strategies [39, 40] in stream processing. These approaches can be used to complement Henge. [38–40] look at single-job elasticity, but not multi-tenancy.

Themis [48] and others [76, 91] reduce load in stream processing systems by dropping tuples. Henge does not drop tuples. Themis uses the SIC metric to evaluate how much each tuple contributes in terms of accuracy to the result. Henge's juice metric is different in that it is used to calculate processing rate and it serves as an input rate-independent throughput metric.

Dhalion [33] supports throughput SLOs for Heron, but does not generalize to varying input rates. It uses backpressure as a trigger for scaling out topologies, but as backpressure takes time to propagate (e.g., after spikes), it is less responsive than CPU load (which Henge uses).

Multi-tenant Resource Management Systems: Resource schedulers like YARN [80] and Mesos [41] can be run under stream processing systems, and manually tuned [28]. As job internals are not exposed to the scheduler, it is hard to make fine-grained decisions for stream processing jobs in an automated fashion.

Cluster Scheduling: Some scheduling work addresses resource fairness and SLO achievement [31, 32, 37, 58, 69, 73]. VM-based scaling approaches [51] do not map directly and efficiently to expressive frameworks like stream processing systems. Among multi-tenant stream processing systems, Chronostream [85] achieves elasticity through migration across nodes. It does not support SLOs.

SLAs/SLOs in Other Areas: SLAs/SLOs have been explored in other areas. Pileus [78] is a geo-distributed storage system that supports multi-level SLA requirements dealing with latency and consistency. Tuba [22] builds on Pileus and uses reconfiguration to adapt to changing workloads. SPANStore [86] is a geo-replicated storage service that automates trading off cost vs. latency, while being consistent and fault-tolerant. E-store [74] re-distributes hot and cold data chunks across cluster nodes if load exceeds a threshold. Cake [81] supports latency and throughput SLOs in multi-tenant storage settings.

10 CONCLUSION

We presented Henge, a system for intent-driven, SLO-based multi-tenant stream processing. Henge provides SLO satisfaction for jobs with latency and/or throughput SLOs. To make throughput SLOs independent of input rate and topology structure, Henge uses a new metric called juice. When jobs miss their SLO, Henge uses three kinds of actions (reconfiguration, reversion or reduction) to improve the total utility achieved cluster-wide. Evaluation with Yahoo! topologies and Twitter datasets showed 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 on a constrained cluster; iii) gracefully handles dynamic workloads; iv) scales with increasing cluster size and jobs; and v) recovers from failures.

ACKNOWLEDGEMENTS

This work was supported in part by: NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft. We would like to thank Robert Evans from the Yahoo! Storm team for workload traces and his invaluable feedback. We thank Boyang Jerry Peng and Reza Farivar for feedback and discussions on initial versions of our system. We also thank Umar Kalim, Mainak Ghosh and Sangeetha Abdu Jyothi for their invaluable input. We thank University of Utah for Emulab support.

REFERENCES

[1] Apache Flink. http://flink.apache.org/. Last Visited: Thursday 6th September, 2018.
[2] Apache Flume. https://flume.apache.org/. Last Visited: Thursday 6th September, 2018.
[3] Apache Samza. http://samza.apache.org/. Last Visited: 03/2016.
[4] Apache Storm. http://storm.apache.org/. Last Visited: Thursday 6th September, 2018.
[5] Apache Zookeeper. http://zookeeper.apache.org/. Last Visited: Thursday 6th September, 2018.
[6] Based on Private Conversations with Yahoo! Storm Team.
[7] CPU Load. http://www.linuxjournal.com/article/9001. Last Visited: Thursday 6th September, 2018.
[8] Elasticity in Spark Core. http://www.ibmbigdatahub.com/blog/explore-true-elasticity-spark/. Last Visited: Thursday 6th September, 2018.
[9] EPA-HTTP Trace. http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html. Last Visited: Thursday 6th September, 2018.
[10] How to collect and analyze data from 100,000 weather stations. https://www.cio.com/article/2936592/big-data/how-to-collect-and-analyze-data-from-100000-weather-stations.html. Last Visited: Thursday 6th September, 2018.
[11] S4. http://incubator.apache.org/s4/. Last Visited: 03/2016.
[12] SDSC-HTTP Trace. http://ita.ee.lbl.gov/html/contrib/SDSC-HTTP.html. Last Visited: Thursday 6th September, 2018.
[13] SLOs. https://en.wikipedia.org/wiki/Service_level_objective. Last Visited: Thursday 6th September, 2018.
[14] Storm 0.8.2 Release Notes. http://storm.apache.org/2013/01/11/storm082-released.html/. Last Visited: Thursday 6th September, 2018.
[15] Storm Applications. http://storm.apache.org/Powered-By.html. Last Visited: Thursday 6th September, 2018.
[16] Uber Releases Hourly Ride Numbers In New York City To Fight De Blasio. https://techcrunch.com/2015/07/22/

uber-releases-hourly-ride-numbers-in-new-york-city-to-fight-de-blasio/. Last Visited: Thursday 6th September, 2018.

[17] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., et al. Aurora: A Data Stream Management System. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2003), ACM, pp. 666–666.

[18] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research* (2005), vol. 5, pp. 277–289.

[19] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. Millwheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proceedings of the VLDB Endowment* (2013), vol. 6, VLDB Endowment, pp. 1033–1044.

[20] Amin, T. Apollo Social Sensing Toolkit. http://apollo3.cs.illinois.edu/datasets.html, 2014. Last Visited: Thursday 6th September, 2018.

[21] Aniello, L., Baldoni, R., and Querzoni, L. Adaptive Online Scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems* (2013), ACM, pp. 207–218.

[22] Ardekani, M. S., and Terry, D. B. A Self-Configurable Geo-Replicated Cloud Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 367–381.

[23] Balkesen, C., Tatbul, N., and Özsu, M. T. Adaptive Input Admission and Management for Parallel Stream Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems* (2013), ACM, pp. 15–26.

[24] Bilal, M., and Canini, M. Towards Automatic Parameter Tuning of Stream Processing Systems. In *Proceedings of the Symposium on Cloud Computing* (2017), ACM, p. 1.

[25] Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., and Stonebraker, M. Operator scheduling in a data stream manager. In *Proceedings 2003 VLDB Conference* (2003), Elsevier, pp. 838–849.

[26] Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. Integrating Scale-Out and Fault Tolerance in Stream Processing using Operator State Management. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2013), ACM, pp. 725–736.

[27] Cervino, J., Kalyvianaki, E., Salvachua, J., and Pietzuch, P. Adaptive Provisioning of Stream Processing Systems in the Cloud. In *Proceedings of the 28th International Conference on Data Engineering Workshops* (2012), IEEE, pp. 295–301.

[28] Cloudera. Tuning YARN — Cloudera. http://www.cloudera.com/documentation/enterprise/5-2-x/topics/cdh_ig_yarn_tuning.html, 2016. Last Visited Thursday 6th September, 2018.

[29] Curino, C., Difallah, D. E., Douglas, C., Krishnan, S., Ramakrishnan, R., and Rao, S. Reservation-Based Scheduling: If You're Late Don't Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.

[30] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review 41*, 6 (2007), 205–220.

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

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

[33] Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. Dhalion: Self-Regulating Stream Processing in Heron. In *Proceedings of the VLDB Endowment* (2017), ACM, p. 1.

[34] Fu, T. Z., Ding, J., Ma, R. T., Winslett, M., Yang, Y., and Zhang, Z. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *Proceedings of the 35th International Conference on Distributed Computing Systems* (2015), IEEE, pp. 411–420.

[35] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. SPADE: The System S Declarative Stream Processing Engine. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2008), ACM, pp. 1123–1134.

[36] Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2014), 1447–1463.

[37] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011), vol. 11, pp. 24–24.

[38] Heinze, T., Jerzak, Z., Hackenbroich, G., and Fetzer, C. Latency-Aware Elastic Scaling for Distributed Data Stream Processing Systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (2014), ACM, pp. 13–22.

[39] Heinze, T., Pappalardo, V., Jerzak, Z., and Fetzer, C. Auto-Scaling Techniques for Elastic Data Stream Processing. In *Proceedings of the 30th International Conference on Data Engineering Workshops* (2014), IEEE, pp. 296–302.

[40] Heinze, T., Roediger, L., Meister, A., Ji, Y., Jerzak, Z., and Fetzer, C. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (2015), ACM, pp. 276–287.

[41] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. 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)* (2011), vol. 11, pp. 22–22.

[42] Jacobson, V. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM Computer Communication Review* (1988), vol. 18, ACM, pp. 314–329.

[43] Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. 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)* (2006), ACM, pp. 431–442.

[44] Jones, C., Wilkes, J., Murphy, N., and Smith, C. Service Level Objectives. https://landing.google.com/sre/book/chapters/service-level-objectives.html. Last Visited: Thursday 6th September, 2018.

[45] Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Goiri, Í., Krishnan, S., Kulkarni, J., and Rao, S. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), p. 117.

[46] Kalim, F., Xu, L., Bathey, S., Meherwal, R., and Gupta, I. Henge: Intent-driven Multi-Tenant Stream Processing. https://arxiv.org/abs/1802.00082. Last Visited: Thursday 6th September, 2018.

[47] Kalyvianaki, E., Charalambous, T., Fiscato, M., and Pietzuch, P. Overload Management in Data Stream Processing Systems with Latency Guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing (Feedback Computing)* (2012).

[48] Kalyvianaki, E., Fiscato, M., Salonidis, T., and Pietzuch, P. Themis: Fairness in Federated Stream Processing under Overload. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 541–553.

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

[50] Kleiminger, W., Kalyvianaki, E., and Pietzuch, P. Balancing Load in Stream Processing with the Cloud. In *Proceedings of the 27th International Conference on Data Engineering Workshops* (2011), IEEE, pp. 16–21.

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

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

[53] Kreps, J., Narkhede, N., Rao, J., et al. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the NetDB* (2011), pp. 1–7.

[54] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. Twitter Heron: Stream Processing at Scale. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2015), ACM, pp. 239–250.

[55] Li, B., Diao, Y., and Shenoy, P. Supporting Scalable Analytics with Latency Constraints. In *Proceedings of the VLDB Endowment* (2015), vol. 8, VLDB Endowment, pp. 1166–1177.

[56] Loesing, S., Hentschel, M., Kraska, T., and Kossmann, D. Stormy: An Elastic and Highly Available Streaming Service in the Cloud. In *Proceedings of the Joint EDBT/ICDT Workshops* (2012), ACM, pp. 55–60.

[57] Low, S. H., and Lapsley, D. E. Optimization Flow Control. I. Basic Algorithm and Convergence. *IEEE/ACM Transactions on networking 7*, 6 (1999), 861–874.

[58] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015), pp. 589–603.

[59] Markets and Markets. Streaming analytics market worth 13.70 Billion USD by 2021. https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html. Last Visited: Thursday 6th September, 2018.

[60] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP

'13, ACM, pp. 439–455.

[61] NAAMAN, M., ZHANG, A. X., BRODY, S., AND LOTAN, G. On the Study of Diurnal Urban Routines on Twitter. In *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media* (2012).

[62] NASIR, M. A. U., MORALES, G. D. F., GARCÃ■A-SORIANO, D., KOURTELLIS, N., AND SERAFINI, M. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proceedings of the 31st International Conference on Data Engineering (ICDE)* (April 2015), pp. 137–148.

[63] NASIR, M. A. U., MORALES, G. D. F., KOURTELLIS, N., AND SERAFINI, M. When Two Choices are Not Enough: Balancing at Scale in Distributed Stream Processing. In *Proceedings of the 32nd International Conference on Data Engineering (ICDE)* (May 2016), IEEE, pp. 589–600.

[64] NOGHABI, S. A., PARAMASIVAM, K., PAN, Y., RAMESH, N., BRINGHURST, J., GUPTA, I., AND CAMPBELL, R. H. Samza: Stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1634–1645.

[65] 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).

[66] PENG, B., HOSSEINI, M., HONG, Z., FARIVAR, R., AND CAMPBELL, R. R-Storm: Resource-Aware Scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference* (2015), ACM, pp. 149–161.

[67] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)* (April 2006), pp. 49–49.

[68] QIAN, Z., HE, Y., SU, C., WU, Z., ZHU, H., ZHANG, T., ZHOU, L., YU, Y., AND ZHANG, Z. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 1–14.

[69] RAMESHAN, N., LIU, Y., NAVARRO, L., AND VLASSOV, V. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy. In *Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (2016), IEEE, pp. 233–244.

[70] RAVINDRAN, B., JENSEN, E. D., AND LI, P. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on* (2005), IEEE, pp. 55–60.

[71] SATZGER, B., HUMMER, W., LEITNER, P., AND DUSTDAR, S. Esc: Towards An Elastic Stream Computing Platform for the Cloud. In *Proceedings of the 4th International Conference on Cloud Computing* (2011), IEEE, pp. 348–355.

[72] SCHNEIDER, S., ANDRADE, H., GEDIK, B., BIEM, A., AND WU, K.-L. Elastic Scaling of Data Parallel Operators in Stream Processing. In *Proceedings of International Parallel and Distributed Processing Symposium* (2009), IEEE, pp. 1–12.

[73] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (2012), vol. 12, pp. 349–362.

[74] TAFT, R., MANSOUR, E., SERAFINI, M., DUGGAN, J., ELMORE, A. J., ABOULNAGA, A., PAVLO, A., AND STONEBRAKER, M. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. In *Proceedings of the VLDB Endowment* (Nov. 2014), vol. 8, VLDB Endowment, pp. 245–256.

[75] TATBUL, N., AHMAD, Y., ÇETINTEMEL, U., HWANG, J.-H., XING, Y., AND ZDONIK, S. Load Management and High Availability in the Borealis Distributed Stream Processing Engine. *GeoSensor Networks* (2006), 66–85.

[76] TATBUL, N., ÇETINTEMEL, U., AND ZDONIK, S. Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 159–170.

[77] TEAM, Y. S. Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, 2015. Last Visited: Thursday 6th September, 2018.

[78] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 309–324.

[79] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., ET AL. Storm @ Twitter. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2014), ACM, pp. 147–156.

[80] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[81] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., KATZ, R., AND STOICA, I. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), ACM, p. 14.

[82] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[83] WIKIPEDIA. Pareto Efficiency — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Pareto_efficiency&oldid=741104719, 2016. Last Visited Thursday 6th September, 2018.

[84] WU, K.-L., HILDRUM, K. W., FAN, W., YU, P. S., AGGARWAL, C. C., GEORGE, D. A., GEDIK, B., BOUILLET, E., GU, X., LUO, G., 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* (2007), VLDB Endowment, pp. 1185–1196.

[85] WU, Y., AND TAN, K.-L. Chronostream: Elastic Stateful Stream Computation in the Cloud. In *Proceedings of the 31st International Conference on Data Engineering* (2015), IEEE, pp. 723–734.

[86] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. Spanstore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 292–308.

[87] XU, L., PENG, B., AND GUPTA, I. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *IEEE International Conference on Cloud Engineering (IC2E)* (2016), IEEE, pp. 22–31.

[88] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.

[89] ZHANG, H., ANANTHANARAYANAN, G., BODIK, P., PHILIPOSE, M., BAHL, P., AND FREEDMAN, M. J. Live video analytics at scale with approximation and delay-tolerance. In *NSDI* (2017), vol. 9, p. 1.

[90] ZHANG, H., STAFMAN, L., OR, A., AND FREEDMAN, M. J. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 390–404.

[91] ZHAO, H. C., XIA, C. H., LIU, Z., AND TOWSLEY, D. A Unified Modeling Framework for Distributed Resource Allocation of General Fork and Join Processing Networks. In *ACM SIGMETRICS Performance Evaluation Review* (2010), vol. 38, ACM, pp. 299–310.