Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand

Le Xu*, Boyang Peng†, Indranil Gupta*
*Department of Computer Science, University of Illinois, Urbana Champaign,
{lexu1, indy}@illinois.edu
†Yahoo! Inc., jerrypeng@yahoo-inc.com

***Abstract*—The era of big data has led to the emergence of new real-time distributed stream processing engines like Apache Storm. We present Stela (STream processing ELAsticity), a stream processing system that supports scale-out and scale-in operations in an on-demand manner, i.e., when the user requests such a scaling operation. Stela meets two goals: 1) it optimizes post-scaling throughput, and 2) it minimizes interruption to the ongoing computation while the scaling operation is being carried out. We have integrated Stela into Apache Storm. We present experimental results using micro-benchmark Storm applications, as well as production applications from industry (Yahoo! Inc. and IBM). Our experiments show that compared to Apache Storm's default scheduler, Stela's scale-out operation achieves throughput that is 21-120% higher, and interruption time that is significantly smaller. Stela's scale-in operation chooses the right set of servers to remove and achieves 2X-5X higher throughput than Storm's default strategy.**

I. Introduction

As our society enters an age dominated by digital data, we have seen unprecedented levels of data in terms of volume, velocity, and variety. Processing huge volumes of high-velocity data in a timely fashion has become a major demand. According to a recent article by BBC News [25], in the year 2012, 2.5 Exabytes of data was generated everyday, and 75% of this data is unstructured. The volume of data is projected to grow rapidly over the next few years with the continued penetration of new devices such as smartphones, tablets, virtual reality sets, wearable devices, etc.

In the past decade, distributed batch computation systems like Hadoop [1] and others [2][3][4][18] have been widely used and deployed to handle big data. Customers want to use a framework that can process large dynamic streams of data on the fly and serve results with high throughput. For instance, Yahoo! uses a stream processing engine to perform for its advertisement pipeline processing, so that it can monitor ad campaigns in real-time. Twitter uses a similar engine to compute trending topics [5] in real time.

To meet this demand, several new stream processing engines have been developed recently, and are widely in use in industry, e.g., Storm [5], System S [26], Spark Streaming [27], and others [9][10][21]. Apache Storm is the most popular among these. A Storm application uses a directed graph (dataflow) of operators (called "bolts") that runs user-defined code to process the streaming data.

Unfortunately, these new stream processing systems used in industry largely lack an ability to *seamlessly* and *efficiently* scale the number of servers in an *on-demand* manner. On-demand means that the scaling is performed when the user (or some adaptive program) requests to increase or decrease the number of servers in the application Today, Storm supports an on-demand scaling request by simply unassigning all processing operators and then reassigning them in a round robin fashion to the new set of machines. This is not seamless as it interrupts the ongoing computation for a long duration. It is not efficient either as it results in sub-optimal throughput after the scaling is completed (as our experiments show later).

Scaling-out and -in are critical tools for customers. For instance, a user might start running a stream processing application with a given number of servers, but if the incoming data rate rises or if there is a need to increase the processing throughput, the user may wish to add a few more servers (scale-out) to the stream processing application. On the other hand, if the application is currently under-utilizing servers, then the user may want to remove some servers (scale-in) in order to reduce dollar cost (e.g., if the servers are VMs in AWS [12]). Supporting on-demand scale-out is preferable compared to over-provisioning which uses more resources (and money in AWS deployments), while on-demand scale-in is preferable to under-provisioning.

On-demand scaling operations should meet two goals: 1) the post-scaling throughput (tuples per sec) should be optimized and, 2) the interruption to the ongoing computation (while the scaling operation is being carried out) should be minimized. We present a new system, named Stela (STream processing ELAsticity), that meets these two goals. For scale-out, Stela carefully selects which operators (inside the application) are given more resources, and does so with minimal intrusion. Similarly, for scale-in, Stela carefully selects which machine(s) to remove in a way that minimizes the overall detriment to the application's performance.

To select the best operators to give more resources when scaling-out, Stela uses a new metric called *ETP (Effective Throughput Percentage)*. The key intuition behind ETP is to capture those operators (e.g., bolts and spouts in Storm) that are both: i) congested, i.e., are being overburdened with incoming tuples, and ii) affect throughput the most because they reach a large number of sink operators. For scale-in, we also use an ETP-based approach to decide which machine(s) to remove and where to migrate operator(s).

The ETP metric is both hardware- and application- agnostic. Thus Stela neither needs hardware profiling (which can be intrusive and inaccurate) nor knowledge of application code.

Existing work on elasticity in System S [15][23], Stream-

The first two authors contributed equally to the paper. This work was completed during the author's study in University of Illinois at Urbana-Champaign.

978-1-5090-1961-8/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2E.2016.38

IEEE computer society

Cloud (elasticity in Borealis) [16], Stormy [21] and [17] propose the use of metrics such as the congestion index, throughput, CPU, latency or network usage, etc. These metrics are used in a closed feedback loop, e.g., under congestion, System S determines *when* the parallelism (number of instances of an operator) should increase, and then does so for *all* congested operators. This is realistic only when infinite resources are available. Stela assumes finite resources (fixed number of added machines or removed machines, as specified by the user), and thus has to solve not only the "when" problem, but also the harder problem of deciding *which* operators need to get/lose resources. We compare Stela against the closest-related elasticity techniques from literature, i.e., [11].

The design of Stela is generic to any data flow system (Section II-A). For concreteness, we integrated Stela into Apache Storm. We present experimental results using micro-benchmark Storm applications, as well as production applications from industry (Yahoo! Inc. and IBM [15]). Our experiments show that Stela's scale-out operation reduces interruption time to a fraction as low as 12.5% that of Storm and achieves throughput that is about 21-120% higher than Storm's. Stela's scale-in operation performs 2X-5X better than Storm's default strategy. We believe our metric can be applied to other systems as well.

The contributions of our work are: 1) development of the novel metric, ETP, that captures the "importance" of an operator, 2) to the best of knowledge, this is the first work to describe and implement on-demand elasticity within Storm, and 3) evaluation of our system on both micro-benchmark applications and on applications used in production.

II. Stela Policy and the ETP Metric

In this section, we first define our data stream processing model. Then, we focus on Stela scale-out and how it uses the ETP metric. Finally we discuss scale-in.

A. Data Stream Processing Model and Assumptions

In this paper, we target distributed data stream processing systems that represent each application as a directed acyclic graph (DAG) of *operators*. An operator is a user-defined logical processing unit that receives one or more streams of *tuples*, processes each tuple, and outputs one or more streams of tuples. We assume operators are stateless. We assume that tuple sizes and processing rates follow an ergodic distribution. These assumptions hold true for most Storm topologies used in industry. An example of this model is shown in Figure 1. Operators that have no parents are *sources* of data injection, e.g., 1. They may read from a Web crawler. Operators with no children are *sinks*, e.g., 6. The intermediate operators (e.g., 2-5) perform processing of tuples. Each sink outputs data (e.g., to a GUI or database), and the application throughput is the sum of throughputs of all sinks in the application. An application may have multiple sources and sinks.

An *instance* (of an operator) is an instantiation of the operator's processing logic and is the physical entity that executes the operator's logic. The number of instances is correlated with the operator's parallelism level. For example, in Storm, these instances are called "executors" (Section III-A).

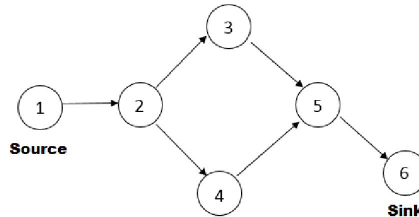


Fig. 1: An Example Of Data Stream Processing Application.

B. Stela: Scale-Out Overview

In this section, we give an overview of how Stela supports scale-out. When the user requests a scale-out with a given number of new machines Stela needs to decide which operators to give more resources to, by increasing their parallelism.

Stela first identifies operators that are congested based on their input and output rates. Then it calculates a per-operator metric called *Expected Throughput Percentage (ETP)*. ETP takes the topology into account: it captures the percentage of total [1] application throughput (across all sinks) that the operator has direct impact on, but ignores all down-stream paths in the topology that are already congested. This ensures that giving more resources to a congested operator with higher ETP will improve the effect on overall application throughput. Thus Stela increases the parallelism of that operator with the highest ETP (from among those congested). Finally Stela recalculates the updated execution speed and *Projected ETP* (given the latest scale-out) for all operators and selects the next operator to increase its parallelism, and iterates this process. To ensure load balance, the total number of such iterations equals the number of new machines added times average number of instances per machine pre-scale. We determine the number of instances to allocate a new machine as: $N_{instances} = (Total\ \#\ of instances)/(\#\ of\ machines)$, in other words $N_{instances}$ is the average number of instances per machine prior to scale-out. This ensures load balance post-scale-out. The schedule of operators on existing machines is left unchanged.

The ETP approach is essentially a greedy approach because it assigns resources to the highest ETP operator in each iteration. Other complex approaches to elasticity may be possible, including graph theory and max-flow techniques–however these do not exist in literature yet and would be non-trivial to design. While we consider these to be interesting directions, they are beyond the scope of this paper.

C. Congested Operators

Before calculating ETP for each operator, Stela determines all congested operators in the graph by calling a CONGESTIONDETECTION procedure. This procedure considers an operator to be congested if the combined speed of its input streams is much higher than the speed at which the input streams are being processed within the operator. Stela measures the input rate, processing rate and output rate of an

[1] This can also be generalized to a weighted sum of throughput across sinks.

operator as the sum of input rates, processing rates and output rates, respectively, across all instances of that operator. An application may have multiple congested operators. In order to determine the best operators that should be migrated during a cluster scaling operation, Stela quantifies the impact of scaling an operator towards the application overall throughput by using the ETP metric.

Stela continuously samples the input rate, emit rate and processing rate of each operator in the processing the topology respectively. The input rate of an operator is calculated as the sum of emit rate towards this operator from all its parents. Stela uses periodic collection every 10 seconds and calculates these rates in a sliding window of recent tuples (of size 20 tuples). These values are chosen based on Storm's default and suggested values, e.g., the Storm scheduler by default runs every 10s.

When the ratio of input to processing exceeds a threshold *CongestionRate*, we consider that operator to be congested. An operator may be congested because it's overloaded by too many tuples, or has inefficient resources, etc. When the operator's input rate equals its processing rate, it is not considered to be congested. Note that we only compare input rates and processing rates (not emit rates) – thus this applies to operators like filter, etc., which may output a different rate than the input rate. The CongestionRate parameter can be tuned as needed and it controls the sensitivity of the algorithm: lower CongestionRate values result in more congested operators being captured. For Stela experiments, we set CongestionRate to be 1.2.

D. Effective Throughput Percentage (ETP)

Effective Throughput Percentage (ETP): To estimate the impact of each operator towards the application throughput, Stela introduces a new metric called Effective Throughput Percentage (ETP). An operator's ETP is defined as the percentage of the final throughput that would be affected if the operator's processing speed were changed.

The ETP of an operator o is computed as:

$$ETP_o = \frac{Throughput_{EffectiveReachableSinks}}{Throughput_{workflow}}$$

Here, $Throughput_{EffectiveReachableSinks}$ denotes the sum of throughput of all sinks reachable from o by at least one un-congested path, i.e., a path consisting only of operators that are not classified as congested. $Throughput_{workflow}$ denotes the sum throughput of all sink operators of the entire application. The algorithm to calculate an operator's ETP is shown in Algorithm 1. This algorithm does a depth first search throughout the application DAG, and calculates ETPs via a post-order traversal. ProcessingRateMap stores processing rates of all operators. Note that if an operator o has multiple parents, then the effect of o's ETP is the same at each of its parents (i.e. it is replicated, not split).

While ETP is not a perfect measurement of post-scaling performance, it provides a good estimate. Our results in Section 4 show that using ETP is a reasonable and practical approach.

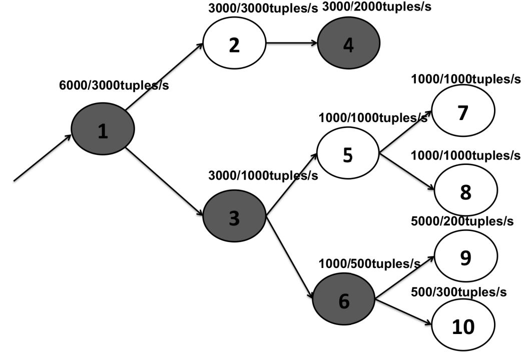


Fig. 2: A sliver of a stream processing application. Each operator is denoted by its input/execution speed. Shaded operators are congested. CongestionRate=1.

Algorithm 1 Find ETP of an operator o of the application

```
1:  function FINDETP(ProcessingRateMap)
2:      if o.child = null then
            return ProcessingRateMap.get(o)/ThroughputSum //o is a sink
3:      end if
4:      SubtreeSum ← 0;
5:      for each descendant child ∈ o do
6:          if child.congested = true then
7:              continue; // if the child is congested, give up the subtree rooted at that child
8:          else
9:              SubtreeSum+ = FINDETP(child);
10:         end if
11:     end for
            return SubtreeSum
12: end function
```

ETP Calculation Example and Intuition: We illustrate the ETP calculation using the example application in Figure 2. The processing rate of each operator is shown. In Figure 2, the operators congested are shown as shaded, i.e. operators 1, 3, 4 and 6. The total throughput of the workflow is calculated as the sum of throughput of sink operators 4, 7, 8, 9 and 10 as $Throughput_{workflow}$=4500 tuples/s.

Let us calculate the ETP of operator 3. Its reachable sink operators are 7, 8, 9 and 10. Of these only 7 and 8 are considered to be the "effectively" reachable sink operators, as they are both reachable via an un-congested path. Thus, increasing the speed of operator 3 will improve the throughput of operators 7 and 8. However, operator 6 is a non-effective reachable for operator 3, because operator 6 is already congested – thus increasing operator 3's resources will only increase operator 6's input rate and make operator 6 further congested, without improving its processing rate. Thus, we ignore the subtree of operator 6 when calculating 3's ETP. The ETP of operator 3 is: $ETP_3 = (1000+1000)/4500 = 44\%$.

Similarly, for operator 1, the sink operators 4, 7, 8, 9 and 10 are reachable, but none of them are reachable via a non-congested path. Thus the ETP of operator 1 is 0. Likewise, we can calculate the ETP of operator 4 as 44% and the ETP of

operator 6 as 11%. Thus, the priority order for Stela to assign resources to these operators is: 3, 4, 6, 1.

E. Iterative Assignment and Intuition

During each iteration, Stela calculates the ETP for all congested operators. Stela targets the operator with the highest ETP and it increases the parallelism of the operator by assigning a new instance of that operator at the newly added machine. If multiple machines are being added, then the target machine is chosen in round-robin manner. Overall this algorithm runs $N_{instances}$ iterations to select $N_{instances}$ target operators (Section II-A showed how to calculate $N_{instances}$).

Algorithm 2 depicts the pseudocode for scale-out. In each iteration, Stela constructs a *CongestedMap*, as explained earlier in Section II-C. If there are no congested operators in the application, Stela chooses a source operator as a target – this is done to increase the input rate of the entire application. If congested operators do exist, for each congested operator, Stela finds its ETP using the algorithm discussed in Section II-D. The result is sorted into $ETPMap$. Stela chooses the operator that has the highest ETP value from $ETPMap$ as a target for the current iteration. It increases the parallelism of this operator by assigning one additional random instance to it, on one of the new machines in a round robin manner.

For the next iteration, Stela estimates the processing rate of the previously targeted operator o proportionally, i.e., if the o previously had an output rate E and k instances, then o's new *projected* processing rate is $E \cdot \frac{k+1}{k}$. This is a reasonable approach since all machines have the same number of instances and thus proportionality holds. Note that even though this may not be accurate, we find that it works in practice. Then Stela uses this to update the output rate for o, and the input rates for o's children (o's children's processing rates do not need updates as their resources remain unchanged. The same applies to o's grand-descendants.). Stela updates emit rate of target operator in the same manner to ensure estimated operator submission rate can be applied.

Once this is done, Stela re-calculates the ETP of all operators by again using Algorithm 1 – we call these new ETPs as *projected* ETPs, or PETPs, because they are based on estimates. The PETPs are used as ETPs for the next iteration. These iterations are repeated until all available instance slots at the new machines are filled. Once this procedure is completed, the schedule is committed by starting the appropriate executors on new instances.

In Algorithm 2, procedure FindETP involves searching for all reachable sinks for every congested operator – as a result each iteration of Stela has a running time complexity of $O(n^2)$ where n is the number of operators in the workflow. The entire algorithm has a running time complexity of $O(m \cdot n^2)$, where m is the number of new instance slots at the new workers.

F. Stela: Scale-In

For scale-in, we assume the user only specifies the number of machines to be removed and Stela picks the "best" machines from the cluster to remove (if the user specifies the exact machines to remove, the problem is no longer challenging).

Algorithm 2 Stela: Scale-out

```
1: function SCALE-OUT
2:     slot ← 0;
3:     while slot < N_instances do
4:         CongestedMap ← CONGESTIONDETECTION;
5:         if CongestedMap.empty = true then
               return source; // none of the operators are congested
6:         end if
7:         for each operator o ∈ workflow do
8:             ETPMap ← FINDETP(Operator o);
9:         end for
10:        target ← ETPMap.max;
11:        ProcessingRateMap.update(target);
12:        EmitRateMap.update(target);
13:        //update the target execution rate
14:        slot++;
15:    end while
16: end function
```

We describe how techniques used for scale-out can also be for scale-in, particularly, the ETP metric. For scale-in we will not be merely calculating the ETP per operator but instead *per machine* in the cluster. That is, we first, calculate the *ETPSum* for each machine:

$$ETPSum(machine_k) = \sum_{i=1}^{n} FindETP(FindComp(\tau_i))$$

ETPSum for a machine is the sum of all ETP of instances of all operators that currently reside on the machine. Thus, for every instance, τ_i, we first find the operator that instance τ_i is an instance of (e.g., $operator_o$) and then find the ETP of that $operator_o$. Then, we sum all of these ETPs. ETPSum of a machine is thus an indication of how much the instances executing on that machine contribute to the overall throughput. The intuition is that a machine with lower ETPSum is a better candidate to be removed in a scale-in operation than a machine with higher ETPSum since the former influences less the application in both throughput and downtime.

The SCALE-IN procedure of Algorithm 3 is called iteratively, as many times as the number of machines requested to be removed. The procedure calculates the ETPSum for every machine in the cluster and puts the machine and its corresponding ETPSum into the ETPMachineMap. The ETPMachineMap is sorted in increasing order of ETPSum values. The machine with the lowest ETPSum will be the target machine to be removed in this round of scale-in. Operators from the machine that is chosen to be removed are re-assigned to the remaining machines in the cluster, in a round robin fashion in increasing order of their ETPSum.

Performing operator migration to machines with lower ETPSum will have less of an effect on the overall performance since machines with lower ETPSum contribute less to the overall performance. This also helps shorten the amount of downtime the application experiences due to the rescheduling. This is because while adding new instances to a machine, existing computation may need to be paused for a certain du-

Algorithm 3 Stela: Scale-in

```
1: function SCALE-IN
2:     for each Machine n ∈ cluster do
3:         ETPMachineMap ← ETPMACHINESUM(n)
4:     end for
5:     ETPMachineMap.sort()
6:     //sort ETPSums by increasing order
7:     REMOVEMACHINE(ETPMachineMap.first())
8: end function
9: function REMOVEMACHINE(Machine n, ETPMachineMap map)
10:    for each instance τ_i on n do
11:        if i > map.size then
12:            i ← 0
13:        end if
14:        Machine x ← map.get(i)
15:        ASSIGN(τ_i, x)
16:        //assigns instances to a round robin fashion
17:        i++
18:    end for
19: end function
```

ration. Instances on a machine with lower ETPSum contribute less to the overall performance and thus this approach causes lower overall downtime.

After this schedule is created, Stela commits it by migrating operators from the selected machines, and then releases these machines. Algorithm 3 involves sorting *ETPSum*, which results in a running time complexity of $O(nlog(n))$.

III. Implementation

We have implemented Stela as a custom scheduler inside Apache Storm [5].

A. Overview of Apache Storm

Storm Application: Apache Storm is a distributed real time processing framework that can process incoming live data [5]. In Storm, a programmer codes an application as a Storm *topology*, which is a graph, typically a DAG (While Storm topologies allow cycles, they are rare and we do not consider them in this paper.), of operators, sources (called spouts), and sinks. The operators in Storm are called bolts. The streams of data that flow between two adjacent bolts are composed of tuples. A Storm task is an instantiation of a spout or bolt.

Users can specify a *parallelism hint* to say how many executors each bolt or spout should be parallelized into. Users can also specify the number of tasks for the bolt or spout – if they don't, Storm assumes one task per executor. Once fixed, Storm does not allow the number of tasks for a bolt to be changed, though the number of executors is allowed to change, to allow multiplexing – in fact Stela leverages this multiplexing by varying the number of executors for congested bolts.

Storm uses *worker processes*. In our experiments, a machine may contain up to 4 worker processes. Worker processes in turn contain *executors* (equivalent to an "instance" in our model in Section II-A). An executor is a thread that is spawned in a worker process. An executor may execute one or more tasks. If an executor has more than one task, the tasks are executed in a sequential manner inside. The number of tasks cannot be changed in Storm, but the number of executors executing the tasks can increase and decrease.

The user specifies in a *topology* (equivalent to a DAG) the bolts, the connections, and how many worker processes to use. The basic Storm operator, namely the bolt, consumes input streams from its parent spouts and bolts, performs processing on received data, and emits new streams to be received and processed downstream. Bolts may filter tuples, perform aggregations, carry out joins, query databases, and in general any user defined functions. Multiple bolts can work together to compute complex stream transformations that may require multiple steps, like computing a stream of trending topics in tweets from Twitter [5]. Bolts in Storm are stateless. Vertices 2-6 in Figure 1 are examples of Bolts.

Storm Infrastructure: A typical Storm Cluster has two types of machines: the master node, and multiple workers nodes. The master node is responsible for scheduling tasks among worker nodes. The master node runs a daemon called *Nimbus*. Nimbus communicates and coordinates with Zookeeper[6] to maintain a consistent list of active worker nodes and to detect failure in the membership.

Each server runs a worker node, which in turn runs a daemon called the *supervisor*. The supervisor continually listens for the master node to assign it tasks to execute. Each worker machine contains many worker processes which are the actual containers for tasks to be executed. Nimbus can assign any task to any worker process on a worker node. Each source (*spout*) and operator (bolt) in a Storm topology can be parallelized to potentially improve throughput. The user specifies the parallelization hint for each bolt and spout.

Storm Scheduler: Storm's default Storm scheduler (inside Nimbus) places tasks of all bolts and spouts on worker processes. Storm uses a round robin allocation in order to balance out load. However, this may result in tasks of one spout or bolt being placed at different workers. Currently, the only method for vanilla Storm to do any sort of scale-in or -out operation, is for the user to execute a *re-balance* operation. This re-balance operation simply deletes the current scheduling and re-schedules all tasks from scratch in a round robin fashion to the modified cluster. This is inefficient, as our experiments later show.

B. Core Architecture

Stela runs as a custom scheduler in a Java class that implements a predefined IScheduler interface in Storm. A user can specify which scheduler to use in a YAML formated configuration file call *storm.yaml*. Our scheduler runs as part of the Storm Nimbus daemon. The architecture of Stela's implementation in Storm is visually presented in Figure 3. It consists of three modules:

1) StatisticServer - This module is responsible for collecting statistics in the Storm cluster, e.g., throughput at each task, bolt, and for the topology. This data is used as input to congestion detection in Sections.

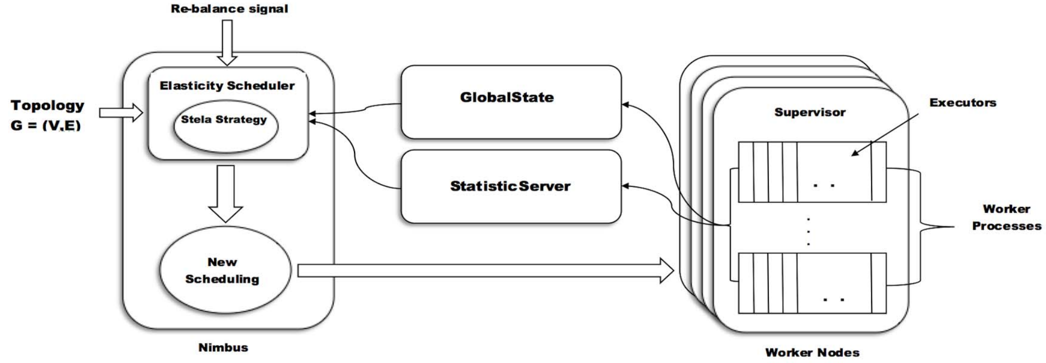


Fig. 3: Stela Architecture.

2) GlobalState - This module stores important state information regarding the scheduling and performance of a Storm Cluster. It holds information about where each task is placed in the cluster. It also stores statistics like sampled throughputs of incoming and outgoing traffic for each bolt for a specific duration, and this is used to determine congested operators as mentioned in Section II-C.
3) Strategy - This module provides an interface for scale-out strategies to implement so that different strategies (e.g., Algorithm 2) can be easily swapped in and out for evaluation purposes. This module calculates a new schedule based on the scale-in or scale-out strategy in use and uses information from the Statistics and GlobalState modules. The core Stela policy (Section II-A) and alternative strategies (Section III-C) are implemented here.
4) ElasticityScheduler - This module is the custom scheduler that implements IScheduler interface. This class starts the StatisticServer and GlobalState modules, and invokes the Strategy module when needed.

When a scale-in or -out signal is sent by the user to the ElasticityScheduler, a procedure is invoked that detects newly joined machines based on previous membership. The ElasticityScheduler invokes the Strategy module, which calculates the entire new scheduling, e.g., for scale-out, it decides all newly created executors that need to be assigned to newly joined machines. The new scheduling is then returned to the ElasticityScheduler which atomically (at the commit point) changes the current scheduling in the cluster. Computation is thereafter resumed.

Fault-tolerance: When no scaling is occurring, failures are handled the same way as in Storm, i.e., Stela inherits Storm's fault-tolerance. If a failure occurs during a scaling operation, Stela's scaling will need to be aborted and restarted. If the scaling is already committed, failures are handled as in Storm.

C. Alternative Strategies

Initially, before we settled on the ETP design in Section II, we attempted to design several alternative topology-aware strategies for scaling out. We describe these below, and we will compare the ETP-based approach against the best of these in our experiments in Section IV. One of these strategies also captures existing work [11].

Topology-aware strategies migrate existing executors instead of creating more of them (as Stela does). These strategies aim to find "important" components/operators in a Storm Topology. Operators, in scale-out, deemed "important" are given priority for migration to the new worker nodes. These strategies are:

- Distance from spout(s) - In this strategy, we prioritize operators that are closer to the source of information as defined in Section II-A. The rationale for this method is that if an operator that is more upstream is a bottleneck, it will affect the performance of all bolts that are further downstream.
- Distance to output bolt(s) - Here, higher priority for use of new resources are given to bolts that are closer to the sink or output bolt. The logic here is that bolts connected near the sink affect the throughput the most.
- Number of descendants - Here, importance is given to bolts with many descendants (children, children's children, and so on) because such bolts with more descendants potentially have a larger effect on the overall application throughput.
- Centrality - Here, higher priority is given to bolts that have a higher number of in- and out-edges adjacent to them (summed up). The intuition is that "well-connected" bolts have a bigger impact on the performance of the system then less well-connected bolts.
- Link Load - In this strategy, higher priority is given to bolts based on the load of incoming and outgoing traffic. This strategy examines the load of links between bolts and migrates bolts based on the status of the load on those links. Similar strategies have been used in [11] to improve Storm's performance. We implemented two strategies for evaluation: Least Link Load and Most Link Load. Least Link Load strategy sorts executors by the load of the links that it is connected to and starts migrating tasks to new workers by attempting to maximize the number of adjacent bolts that are on the same server. The Most Link Load strategy does the reverse.

During our experiments, we found that all the above strategies performed comparably for most topologies, however the Link Load based strategy was the only one that improved

performance for the Linear topology. Thus, the Least Link Load based strategy is representative of strategies that attempt to minimize the network flow volume in the topology schedule. Hence in the next section, we will compare the performance of the Least Link Load based strategy with Stela. This is thus a comparison of Stela against [11].

IV. Evaluation

Our evaluation is two-pronged, and consists of both microbenchmark topologies and real topologies (including two from Yahoo!). We adopt this approach due to the absence of standard benchmark suites (like TPC-H or YCSB) for stream processing systems. Our microbenchmarks include small topologies such as star, linear and diamond, because we believe that most realistic topologies will be a combination of these. We also use two topologies from Yahoo! Inc., which we call PageLoad topology and Processing topology, as well as a Network Monitoring topology [15] . We also present a comparison among Stela, the Link Load Strategy (Section III-C and [11]), and Storm's default scheduler (which is state of the art).

A. Experimental Setup

For our evaluation, we used two types of machines from Emulab [7] testbed to perform our experiments. Our typical Emulab setup consists of a number of machines running Ubuntu 12.04 LTS images, connected via a 100Mpbs VLAN. A type 1 machine has one 3 GHz processor, 2 GB of memory, and 10,000 RPM 146 GB SCSI disks. A type 2 machine has one 2.4 GHz quad core processor, 12 GB of memory, 750 GB SATA disks. The settings for all topologies tested are listed in Table I. For each topology, the same scaling operations were applied to all strategies.

B. Micro-benchmark Experiments

Storm topologies can be arbitrary. To capture this, we created three micro-topologies that commonly appear as building blocks for larger topologies. They are:

- Linear - This topology has 1 source and 1 sink with a sequence of intermediate bolts.
- Diamond - This topology has 1 source and 1 sink, connected parallel via several intermediate bolts (Thus tuples will be processed via the path of "source - bolt - sink".
- Star - This topology has multiple sources connected to multiple sinks via a single unique intermediate bolt.

Figure 4a, 4b, 4c present the throughput results for these topologies. For the Star, Linear, and Diamond topologies we observe that Stela's post scale-out throughput is around 65%, 45%, 120% better than that of Storm's default scheduler, respectively. This indicates that Stela correctly identifies the congested bolts and paths and prioritizes the right set of bolts to scale out. The lowest improvement is for the Linear topology (45%) – this lower improvement is due to the limited diversity of paths, where even a single congested bolt can bottleneck the sink.

In fact, for Linear and Diamond topologies, Storm's default scheduler does not improve throughput after scale-out. This is because Storm's default scheduler does not increase the number of executors, but attempts to migrate executors to a new machine. When an executor is not resource-constrained and it is executing at maximum performance, migration doesn't resolve the bottleneck.

C. Yahoo Storm Topologies and Network Monitoring Topology

We obtained the layouts of two topologies in use at Yahoo! Inc. We refer to these two topologies as the Page Load topology and Processing topology (these are not the original names of these topologies). The layout of the Page Load Topology is displayed in Figure 5a, the layout of the Processing topology is displayed in Figure 5b and the layout of the Network Monitoring topology is displayed in Figure 5c.

We examine the performance of three scale-out strategies: default, Link based (Section III-C and [11]), and Stela. The throughput results are shown in Figure 6. Recall that link load based strategies reduce the network latency of the workflow by co-locating communicating tasks to the same machine.

From Figure 6, we observe that Stela improves the throughput by 80% after a scale-out for both Yahoo topologies. In comparison, Least Link Load strategy barely improves the throughput after a scale-out because migrating tasks that are not resource-constrained will not significantly improve performance. The default scheduler actually decreases the throughput after the scale-out, since it simply unassigns all executors and reassigns all the executors in a round robin fashion to all machines including the new ones. This may cause machines with "heavier" bolts to be overloaded thus creating newer bottlenecks that are damaging to performance especially for topologies with a linear structure. In comparison, Stela's post-scaling throughput is about 125% better than Storm's post-scaling throughput for both Page Load and Processing topologies – this indicates that Stela is able to find the most congested bolts and paths and give them more resources.

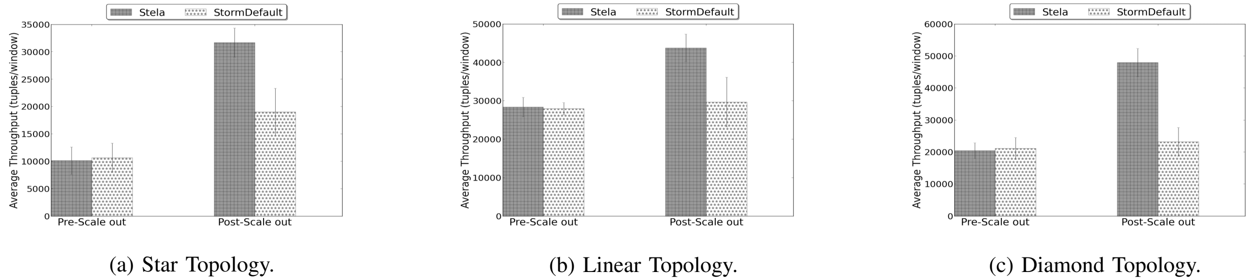
In addition to the above two topologies, we also looked at a published application from IBM [15], and we wrote from scratch a similar Storm topology (shown in Figure 5c). By increasing cluster size from 8 to 9, our experiment (Figure 6c) shows that Stela improves the throughput by 21% by choosing to parallelize the congested operator closest to the sink. In the meantime Storm default scheduler does not improve post scale throughput and Least Link Load strategy decreases system throughput.

D. Convergence Time

We measure interruption to ongoing computation by measuring the *convergence time*. The convergence time is the duration of time between when the scale-out operation starts and when the overall throughput of the Storm Topology stabilizes. Concretely, the convergence time duration stopping criteria are: 1) the throughput oscillates twice above and twice below the average of post scale-out throughput, and 2) the oscillation is within a small standard deviation of 5%. Thus a lower convergence time means that the system is less intrusive

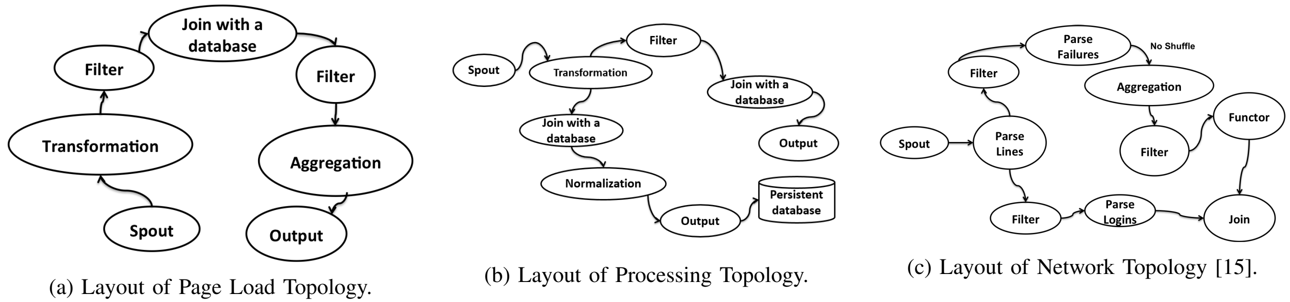
Toplogy Type	# of tasks per Component	Initial # of Executors per Component	# of Worker Processes	Initial Cluster Size	Cluster Size after Scaling	Machine Type
Star	4	2	12	4	5	1
Linear	12	6	24	6	7	1
Diamond	8	4	24	6	7	1
Page Load	8	4	28	7	8	1
Processing	8	4	32	8	9	1
Network	8	4	32	8	9	2
Page Load Scale-in	15	15	32	8	4	1

TABLE I: Experiment Settings and Configurations.



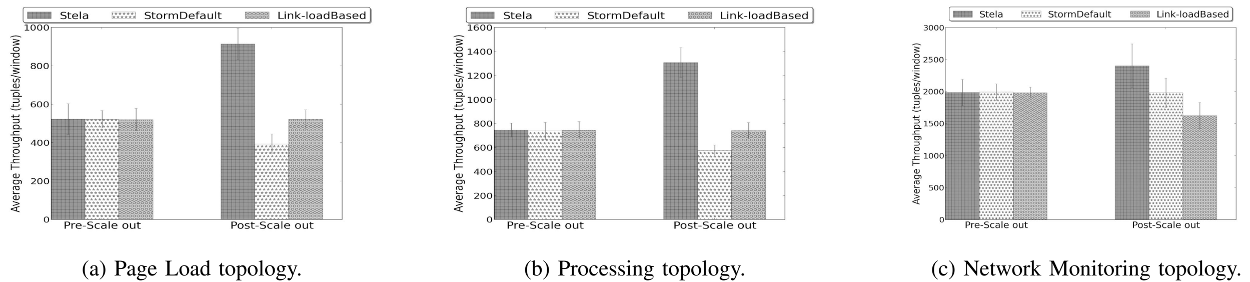
(a) Star Topology. (b) Linear Topology. (c) Diamond Topology.

Fig. 4: Scale-out: Throughput Behavior for Micro-benchmark Topologies. (Window size 10 seconds)



(a) Layout of Page Load Topology. (b) Layout of Processing Topology. (c) Layout of Network Topology [15].

Fig. 5: Two Yahoo! Topologies and a Network Monitoring Topology derived from [15]



(a) Page Load topology. (b) Processing topology. (c) Network Monitoring topology.

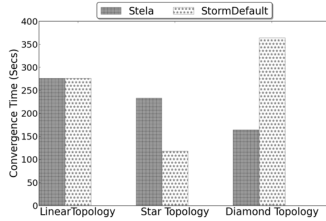
Fig. 6: Scale-out: Throughput Behavior for Yahoo! Topologies and Network Monitoring Topology. (Window size 10 seconds)

during the scale out operation, and it can resume meaningful work earlier.

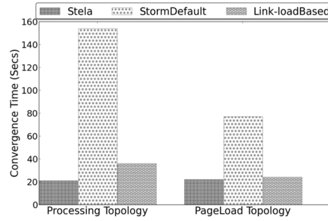
Figure 7a and Figure 7b show the convergence time for both Micro-benchmark Topologies and Yahoo Topologies. We observe that Stela is far less intrusive than Storm when scaling out in the Diamond topology (92% lower) and about as intrusive as Storm in the Linear topology. Stela takes longer to converge than Storm in some cases like the Star topology, primarily because a large number of bolts are affected all at once by the Stela's scaling. Nevertheless the post-throughput scaling is worth the longer wait (Figure 4a). Further, for the Yahoo Topologies, Stela's convergence time is 88% and 75% lower than that of Storm's default scheduler.

The main reason why Stela has a better convergence time than both Storm's default scheduler and Least Link Load strategy [11] is that Stela does not change the current scheduling at existing machines (unlike Storm's default strategy and Least Link Load strategy), instead choosing to schedule operators at the new machines only.

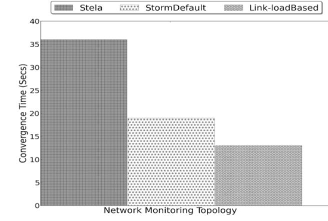
In Network Monitoring topology, Stela experiences longer convergence time than Storm's default scheduler and Least Link Load strategy due to re-parallelization during the scale-out operation (Figure 7c). However, the benefit, as shown in Figure 6c, is the higher post-scale throughput provided by



(a) Throughput Convergence Time for Micro-benchmark Topologies.



(b) Throughput Convergence Time for Yahoo! Topologies.



(c) Throughput Timeline for Network Monitoring Topologies.

Fig. 7: Scale-out: Convergence Time Comparison (in seconds).

Stela.

E. Scale-In Experiments

We examine the performance of Stela scale-in by running Yahoo's PageLoad topology. The initial cluster size is 8 and Figure 8a shows the throughput change after shrinking cluster size to 4 machines. (We initialize the operator allocation so that each machine can be occupied by tasks from fewer than 2 operators (bolts and spouts)). We compare against the performance of a round robin scheduler (same as Storm's default scheduler), using two alternative groups of randomly selected machines.

We observe Stela preserves throughput after scale-in while the Storm groups experience 80% and 40% throughput decrease respectively. Thus, Stela's post scale-in throughput is 2X - 5X higher than randomly choosing machines to remove. Stela also achieves 87.5% and 75% less down time (time duration when throughput is zero) than group 1 and group 2, respectively – see Figure 8b. This is primarily because in Stela migrating operators with low ETP will intrude less on the application, which will allow downstream congested components to digest tuples in their queues and continue producing output. In the PageLoad Topology, the two machines with lowest ETPs are chosen to be redistributed by Stela, which generates less intrusion for the application thus significantly better performance than Storm's default scheduler.

Thus, Stela is intelligent at picking the best machines to remove (via ETPSum). In comparison, Storm has to be lucky. In the above scenario, 2 out of the 8 machines were the "best". The probability that Storm would have been lucky to pick both (when it picks 4 at random) = $\binom{6}{2}/\binom{8}{4}$ = 0.21, which is low.

V. Related Work

Based on its predecessor Aurora [8], Borealis [24] is a stream processing engine that enables queries to be modified on the fly. Borealis focuses on load balancing on individual machines and distributes load shedding in a static environment. Borealis also uses ROD (resilient operator distribution) to determine the best operator distribution plan that is closest to an "ideal" feasible set: a maximum set of machines that are underloaded. Borealis does not explicitly support elasticity.

Stormy [21] uses a logical ring and consistent hashing to place new nodes upon a scale out. It does not take congestion into account, which Stela does. StreamCloud [16] builds elasticity into the Borealis Stream Processing Engine [9]. StreamCloud modifies the parallelism level by splitting queries into sub queries and uses rebalancing to adjust resource usage. Stela does not change running topologies because we consider it intrusive to the applications.

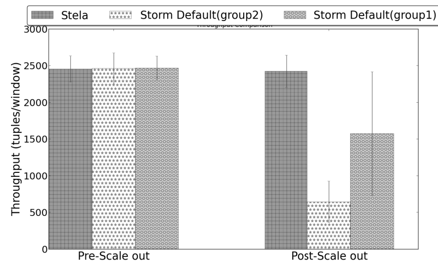
SEEP [13] uses an approach to elasticity that mainly focuses on operator's state management. It proposes mechanisms to backup, restore and partition operators' states in order to achieve short recovery time. There have been several other papers focusing on elasticity for stateful stream processing systems. [15][23] from IBM both enable elasticity for IBM System S [10][19][26] and SPADE [14], by increasing the parallelism of processing operators. These papers apply networking concepts such as congestion control to expand and contract the parallelism of a processing operator by constantly monitoring the throughput of its links. These works do not assume fixed number of machines provided (or taken away) by the users. Our system aims at intelligently prioritizing target operators to further parallelize to (or migrate from) user-determined number of machines joining in (or taken away from) the cluster, with a mechanism to optimize throughput.

Twitter's Heron [20] improves Storm's congestion handling mechanism by using back pressure – however elasticity is not explicitly addressed. Recent work [22] proposes an elasticity model that provides latency guarantee by tuning task-wise parallelism level in a fixed size cluster. Meanwhile, another recent work [17] implemented stream processing system elasticity. However, [17] focused on latency (not throughput) and on policy (not mechanism). Nevertheless, Stela's mechanisms can be used as a black box inside of [17].

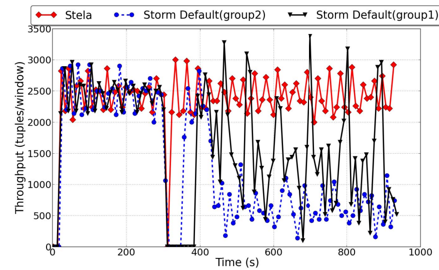
Some of these works have looked at *policies* for adaptivity [17], or [21][15][23][16] focus on the *mechanisms* for elasticity. These are important building blocks for adaptivity. To the best of our knowledge, [11] is the only existing mechanism for elasticity in stream processing systems – Section IV compared Stela against it.

VI. Conclusion

In this paper, we presented novel scale-out and scale-in techniques for stream processing systems. We have created a novel metric, ETP (Effective Throughput Percentage), that accurately captures the importance of operators based on congestion and contribution to overall throughput. For scale-out, Stela first selects congested processing operators to re-parallelize based on ETP. Afterwards, Stela assigns extra resources to



(a) Throughput Convergence Time for Yahoo! Topologies.



(b) Post Scale-in Throughput Timeline

Fig. 8: Scale-in Experiments (Window size 10 seconds).

the selected operators to reduce the effect of the bottleneck. For scale-in, we also use a ETP-based approach that decides which machine to remove and where to migrate affected operators. Our experiments on both micro-benchmarks Topologies and Yahoo Topologies showed significantly higher post-scale out throughput than default Storm and Link-based approach, while also achieving faster convergence. Compared to Apache Storm's default scheduler, Stela's scale-out operation reduces interruption time to a fraction as low as 12.5% and achieves throughput that is 45-120% higher than Storm's. Stela's scale-in operation chooses the right set of servers to remove and performs 2X-5X better than Storm's default strategy.

ACKNOWLEDGEMENT

We thank our collaborators at Yahoo! Inc. for providing us the Storm topologies: Matt Ahrens, Bobby Evans, Derek Dagit, and the whole Storm development team at Yahoo. This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, and AFOSR/AFRL FA8750-11-2-0084, and a generous gift from Microsoft.

REFERENCES

[1] "Apache Hadoop," http://hadoop.apache.org/, 2016, ONLINE.
[2] "Apache Hive," https://hive.apache.org/, 2016, ONLINE.
[3] "Apache Pig," http://pig.apache.org/, 2016, ONLINE.
[4] "Apache Spark," https://spark.apache.org/, 2016, ONLINE.
[5] "Apache Storm," http://storm.incubator.apache.org/, 2016, ONLINE.
[6] "Apache Zookeeper," http://zookeeper.apache.org/, 2016, ONLINE.
[7] "Emulab," http://emulab.net/, 2016, ONLINE.
[8] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin *et al.*, "Aurora: A data stream management system," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 2003, pp. 666–666.
[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the Borealis stream processing engine." in *The Conference on Innovative Data Systems Research (CIDR)*, vol. 5, 2005, pp. 277–289.
[10] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A distributed, scalable platform for data mining," in *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*. ACM, 2006, pp. 27–37.
[11] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in Storm," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM, 2013, pp. 207–218.
[12] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[13] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 725–736.
[14] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, pp. 1123–1134.
[15] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 25, no. 6, pp. 1447–1463, 2014.
[16] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems.*, vol. 23, no. 12, pp. 2351–2365, 2012.
[17] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 276–287.
[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.
[19] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, 2006, pp. 431–442.
[20] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.
[21] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: An elastic and highly available streaming service in the cloud," in *Proceedings of the 2012 Joint EDBT/ICDT Workshops*. ACM, 2012, pp. 55–60.
[22] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, June 2015, pp. 399–410.
[23] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009*. IEEE, 2009, pp. 1–12.
[24] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, "Load management and high availability in the Borealis distributed stream processing engine," in *GeoSensor Networks*. Springer, 2008, pp. 66–85.
[25] M. Wall, "Big Data: Are you ready for blast-off?" http://www.bbc.com/news/business-26383058, 2016, ONLINE.
[26] K.-L. Wu, K. W. Hildrum, W. Fan, P. S. Yu, C. C. Aggarwal, D. A. George, B. Gedik, E. Bouillet, X. Gu, G. 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 Databases*. VLDB Endowment, 2007, pp. 1185–1196.
[27] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 2012, pp. 10–10.