Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows

Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, and Desislava Dimitrova, ***ETH Zurich;***
Matthew Forshaw, ***Newcastle University;*** **Timothy Roscoe,** ***ETH Zurich***

https://www.usenix.org/conference/osdi18/presentation/kalavri

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows

Vasiliki Kalavri[†], *John Liagouris*[†], *Moritz Hoffmann*[†],
Desislava Dimitrova[†], *Matthew Forshaw*[‡,*], *Timothy Roscoe*[†]

[†]*Systems Group, Department of Computer Science, ETH Zürich, firstname.lastname@inf.ethz.ch*

[‡]*Newcastle University, firstname.lastname@newcastle.ac.uk*

Abstract

Streaming computations are by nature long-running, and their workloads can change in unpredictable ways. This in turn means that maintaining performance may require dynamic scaling of allocated computational resources.

Some modern large-scale stream processors allow dynamic scaling but typically leave the difficult task of deciding *how much* to scale to the user. The process is cumbersome, slow and often inefficient. Where automatic scaling is supported, policies rely on coarse-grained metrics like observed throughput, backpressure, and CPU utilization. As a result, they tend to show incorrect provisioning, oscillations, and long convergence times.

We present DS2, an automatic scaling controller for such systems which combines a general performance model of streaming dataflows with lightweight instrumentation to estimate the *true* processing and output rates of individual dataflow operators.

We apply DS2 on Apache Flink and Timely Dataflow and demonstrate its accuracy and fast convergence. When compared to Dhalion, the state-of-the-art technique in Heron, DS2 converges to the optimal, backpressure-free configuration in a single step instead of six.

1 Introduction

We present DS2, a low-latency, robust controller for *dynamic scaling* of streaming analytics applications, which can vary the resources available to a computation so as to handle variable workloads quickly and efficiently.

Static provisioning is a poor fit for continuous, long-running streaming applications: it forces users to choose a single point on the spectrum between allocating resources for worst-case, peak load (which is inefficient) and suffering degraded performance during load spikes. Fixing resources *a priori* almost inevitably leads to a system which is over- or under-provisioned for much of its execution.

[*]Work done while visiting the Systems Group at ETH Zürich

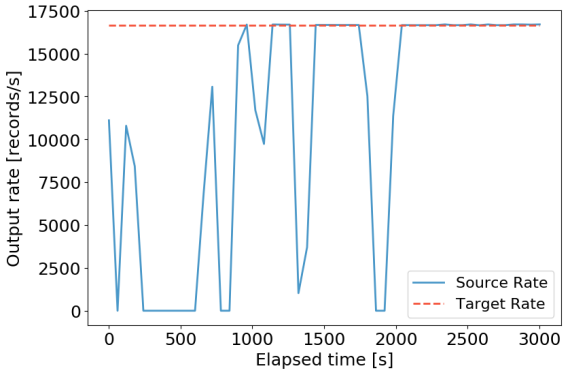


Figure 1: Effect of Dhalion's scaling decisions on the source rate when trying to match the target throughput of an under-provisioned word count dataflow.

The solution is to dynamically scale the system in response to load, an idea used extensively in cloud environments [30, 31]. This requires both a *mechanism* for scaling the computation, and a *scaling controller* which decides when and how to scale. This paper focuses on the latter; DS2 is designed to be mechanism-agnostic.

A scaling controller makes two kinds of decisions. First, it detects symptoms of over- or under-provisioning (e.g. backpressure) and decides *whether* to make a change. Detection is often straightforward and addressed by conventional monitoring tools. Second, the controller must identify the *causes* of symptoms (e.g. a bottlenecked or idle operator) and *propose* a scaling action.

The second decision is challenging, involving performance analysis and prediction. Streaming systems supporting a form of automatic dynamic scaling (e.g. Google Cloud Dataflow [26, 5], Heron [27, 13], Pravega [11], Spark Streaming [45], and IBM System S [15]) and research prototypes (e.g. Seep [12] and StreamCloud [17]) focus on the first decision and either ignore or provide speculative, often ad-hoc solutions for the second.

A good scaling controller should provide the SASO properties [19] familiar from Control Theory: *Stability* (not oscillating between different configurations), *Accuracy* (finding the optimal configuration for the given workload), *Short settling times* to reach the optimal configuration, and *not Overshooting*.

Speculative scaling decisions which do not provide these properties can be bad for streaming systems. First, they lead to temporary over- or under-provisioning, and the resulting sub-optimal resource utilization incurs unnecessary costs. Second, oscillations can in turn degrade performance due to frequent scaling actions. Finally, speculative scaling can be slow to converge, resulting in Service Level Objective (SLO) violations or load shedding.

Figure 1 illustrates these problems in the state-of-the-art Dhalion controller [13] of Heron, using the same word count dataflow as in the original paper. The dashed line shows the target throughput (source output rate), while the solid line tracks the achieved throughput, which varies due to backpressure as Dhalion changes the computation scale. Dhalion performs six scaling decisions, taking more than 30 minutes to converge.

We make the following contributions in this paper. First, we review how existing dynamic scaling techniques can lead to inaccurate, unstable, or slow provisioning decisions. We identify the causes of these effects (§ 2), which we attribute to the lack of a comprehensive performance model, dependence on heuristics, and use of coarse-grained, externally-observed execution metrics.

Second, we propose DS2, a general model and controller for automatic scaling of distributed streaming dataflows (§ 3). DS2 can accurately estimate parallelism for *all* dataflow operators within a *single* scaling decision, and operates reactively online. As a result, DS2 eliminates oscillation and overprovisioning when making scaling decisions. DS2 bases scaling decisions on real-time performance traces, and is general: it relies neither on specific signals like backpressure, as in [13], nor simplistic assumptions like 1-1 operator selectivity, as in [41].

Third, DS2 gives leverage on existing state-of-the-art approaches: when used in Heron, it identifies the optimal backpressure-free configuration in a few seconds and one step, while Dhalion performs six steps to reach an overprovisioned configuration in the same scenario (§ 5.2).

Fourth, we apply DS2 on Apache Flink (§ 5.3) and demonstrate fully-automatic scaling of streaming dataflows under dynamic workload.

Finally, we show that DS2 is accurate and converges quickly for both Apache Flink and Timely Dataflow (§ 5.4 and § 5.5). In all experiments DS2 takes at most three steps to reach the optimal configuration.

2 Background and Motivation

Designing a scaling controller with SASO properties is non-trivial, and existing dynamic scaling techniques for stream processing do not achieve them. Here, we summarize existing approaches, and then examine why they frequently lead to inaccurate, unstable, and slow scaling decisions, before proposing our solution.

Many stream processors [45, 8, 40, 27, 4, 43] have elastic runtimes and allow job reconfiguration by migrating or by externalizing state, but the majority relies entirely on manual intervention for both symptom detection and scaling actions.

Table 1 summarizes those systems that do provide some form of automatic scaling (for details also see [10]). We categorize them by (i) *metrics* used for symptom detection, (ii) *policy* logic for deciding when to scale, (iii) type of *scaling action* which defines which operators to scale and by how much, and (iv) optimization *objective* (i.e. latency or throughput SLO).

We identify two areas in which current systems fall short of the controller properties we would like: first, the metrics used do not provide enough information to make fast and accurate decisions as to how to rescale the system, and second, the policies used for scaling (and the models they are based on) are often simplistic and rule-based.

Limited metrics: Most systems rely on coarse-grained *externally observed* metrics to detect suboptimal scaling: CPU utilization, throughput, queue sizes, etc.

CPU and memory utilization can be inadequate metrics for streaming applications, particularly in cloud environments due to multi-tenancy and performance interference [38]. StreamCloud [17] and Seep [12] try to mitigate the problem by separating user time and system time, but preemption can make these metrics misleading: high CPU usage by a task running on the same physical machine as a dataflow operator can trigger incorrect scale-ups (false positives) or prevent correct scale-downs (false negatives), for example. Google Cloud Dataflow [26] uses CPU utilization only for scale-down decisions but could still suffer from false negatives. CPU usage is also unsuitable for systems like Timely [32, 33], where operators spin waiting for input.

These metrics also imply continuous threshold tuning, a cumbersome and error-prone process. Incorrect scaling decisions can often arise from slightly misconfigured thresholds, even on fine-grained metrics [13].

Dhalion [13] and IBM Streams [15] also use backpressure and congestion to identify bottlenecks. These signals are only helpful where a bottleneck exists. If the dataflow is using resources unnecessarily, such metrics will not trigger reconfiguration. Moreover, in under-provisioned dataflows, backpressure will only detect a single bottle-

System	Metrics	Policy	Scaling Action	Objective
Borealis [3]	CPU, network slack, queue sizes	Rule-based	Load shedding	Latency, throughput
StreamCloud [17]	Average CPU, observed rates	Threshold-based	Speculative, multi-operator	Throughput
Seep [12]	User/system CPU time	Threshold-based	Speculative, single-operator	Latency, throughput
IBM Streams [15]	Congestion, observed rates	Threshold-based, blacklisting	Speculative, single-operator	Throughput
FUGU+ [18]	CPU, processing time	Threshold-based	Speculative, single-operator	Latency
Nephele [29]	Mean task latency, service time, interarrival time, channel latency	Queuing theory model	Predictive, multi-operator	Latency
DRS [14]	Service time, interarrival time	Queuing theory model	Predictive, multi-operator	Latency
Stela [44]	Observed rates	Threshold-based	Speculative, single-operator	Throughput
Spark Streaming [1, 2]	Pending tasks	Threshold-based	Speculative, multi-operator	Throughput
Google Dataflow [6]	CPU, backlog, observed rates	Heuristics	Speculative, multi-operator	Latency, throughput
Dhalion [13]	Backpressure, queue sizes, observed rates	Rule-based, blacklisting	Speculative, single-operator	Throughput
Pravega [11]	Observed rates	Rule-based	Speculative, single-operator	Throughput
DS2	True processing and output rates	Dataflow model	Predictive, multi-operator	Throughput

Table 1: Overview of automatic scaling policies in distributed dataflow systems.

neck; for this reason and to minimize the effects of incorrect decisions [39, 13], each scaling action only configures one operator, increasing convergence time.

Simplistic performance models: scaling policy is generally expressed in simple rules, using predefined thresholds and conditions, e.g. *CPU utilization* $>$ *50 and backpressure* $\implies$ *scale up*. This results in a simple performance model with poor predictive accuracy, which is unable to consider the structure of the dataflow graph or computational dependencies among operators. We note the exceptions of Nephele [29] and DRS [14], which use queuing theory models. Both systems show poor prediction quality in some cases, while Nephele also seems to suffer from temporary over-provisioning and slow convergence.

Since the controller cannot accurately estimate how much to scale an operator, scaling actions are mostly *speculative*. The system applies pessimistic strategies which introduce only small changes to the number of provisioned resources [12, 15] and most policies configure a single operator at a time. This delays convergence to a steady state significantly, as all steps of the scaling process are repeated many times: SLO monitoring, decision making, state migration, and redeployment. [13] shows that, from the point that backpressure is observed, Heron needs almost an hour to reach a steady state that can handle the input rate.

More aggressive strategies apply configurations, blacklisting them if they degrade performance. [39] allows arbitrary scaling steps but requires a user-defined function to calculate the new number of instances whereas [2] supports exponential increase in resources. StreamCloud [17] tries to estimate the optimal number of VMs in a single step, but using very coarse-grained scaling (a subgraph of the dataflow topology). Google Cloud Dataflow is the only system we know with fully automatic scaling per operator, although the details of the model used have not been disclosed.

A better approach: stepping back, it seems a more promising approach for making scaling decisions would take into account both (i) each operator's *true* processing and output capabilities, regardless of backpressure or other effects, and (ii) the dataflow topology and how scaling each operator will affect downstream operators.

Figure 2 gives an intuition of how this works showing the execution timelines of operator instances in a simple dataflow. Solid lines show *useful* work performed by an instance (e.g. record processing) while dotted lines show it waiting for input or output. Edges across timelines represent data transfer.

In this example, o_1 is a bottleneck slowing down both the source and o_2 by pausing their execution. Backpressure means that an external observer sees o_1 processing 10 rec/s and o_2 processing 100 rec/s. Based on this, a policy might provision three additional instances for o_1 to reach a target of 40 rec/s, but it could not accurately estimate how much to scale o_2 and would need to make a speculative decision or apply an extra reconfiguration step.

A better approach would measure the useful time of an operator's timeline and would determine the true rate of o_1 as 10 rec/s and that of o_2 as 200 rec/s, inferring that when increasing the parallelism of o_1 to 4, it also needs to double the parallelism of o_2 to keep up with the output rate. Note this can be calculated globally, i.e. for all operators in the dataflow, *in a single step*.

DS2 does precisely this, obtaining rate measurements of each operator by lightweight instrumentation (already present in many streaming systems). In the rest of the paper we define this notion, extend it to more complex

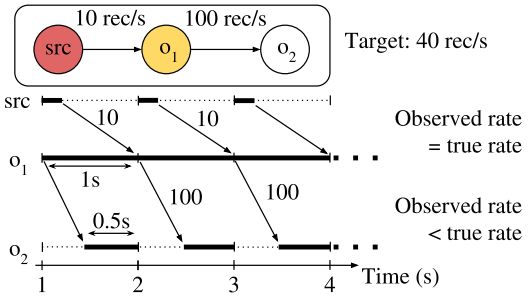


Figure 2: An under-provisioned dataflow and the execution timelines of its operators. Target throughput is 40 rec/s, but o_1 is a bottleneck creating backpressure and limiting the observed source rate to 10 rec/s.

dataflow graphs with multiple sources, and show how DS2 implements it to provide fast, accurate, and stable reconfiguration of streaming dataflows.

3 The DS2 model

DS2 identifies the optimal level of parallelism for each operator in the dataflow on the fly, while the computation executes, based on real-time performance traces. It maintains a changing *provisioning plan*, i.e. the number of resources allocated to each operator. It therefore works *online* and in a *reactive* setting.

Note that we do not target offline computation of an initial resource provisioning plan (as in [7]). Such initial configurations quickly become sub-optimal in a live system where workloads and/or internal operator states change continuously. However, for static workloads known *a priori*, DS2 could use historical performance metrics and offline micro-benchmarks (as in [20, 21, 16]) to estimate the optimal levels of parallelism before deployment.

In this section we define the scaling problem (§ 3.1), describe the DS2 model (§ 3.2), and discuss the model assumptions (§ 3.3) and properties (§ 3.4).

3.1 Problem definition

We target distributed streaming dataflow systems like Flink [9] and Heron [27] that execute data-parallel computations on shared-nothing clusters. Such a computation can be represented as a *logical* directed acyclic graph $G = (V, E)$, where vertices in V denote operators and edges in E are data dependencies between them. A vertex with no incoming edges (no *upstream* operators) is a *source* and a vertex with no outgoing edges (no *downstream* operators) is a *sink*.

A dataflow computation runs as a *physical* execution plan which maps dataflow operators to provisioned compute resources (or workers). Let the graph $G' = (V', E')$ represent the execution plan. Vertices in V' are *operator* (or *task*) instances of a corresponding vertex in V and edges are data *channels*. The assignment of tasks to workers is system-specific. We show in § 5 that DS2's scaling policy is independent of this assignment.

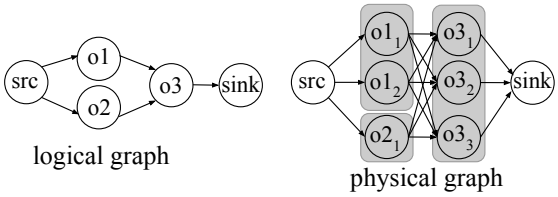


Figure 3: Logical and physical dataflow graphs.

Figure 3 illustrates a logical graph and its corresponding physical graph for a dataflow with a source, a sink, and three operators. Operators $o1$, $o2$ and $o3$ execute with two, one and three instances, respectively.

The Scaling Problem. Given a logical dataflow with sources $s_1, s_2, ..., s_n$ and rates $\lambda_1, \lambda_1, ..., \lambda_n$, identify the minimum parallelism π_i per operator such that the physical dataflow can sustain all source rates.

Source operators generate records at a *rate* λ_s, defined by application data sources (sensors, stock market feeds, etc.). To maximize system throughput, the execution plan must sustain the full source rate. This means that each operator must be able to process data without stalling its upstream operators from producing output.

Like any controller, DS2 targets workload changes on a timescale greater than its convergence time, and reacting to spikes or other changes on a shorter timescale than the convergence time would cause inefficient fluctuations. In these latter cases, the use of backpressure, buffering, or load shedding leads to more stable results than dynamic scaling at the cost of increased latency or lost data.

3.2 Performance model

We consider operator instances as repeatedly performing three activities in sequence: *deserialization*, *processing*, and *serialization*. This fits all types of operators in most modern streaming dataflow systems, including Heron, Flink, and Timely. When an operator instance is scheduled for execution, it pulls records from its input, deserializes them, applies its processing logic, and serializes the results (if any), which are pushed to the output. Serialization and deserialization are optional and happen only when data is moved between operator instances executed within different OS processes, otherwise data is usually exchanged via shared memory (e.g. queues).

The model is based on the concept of *useful time*, which we define for an operator instance as follows:

Useful Time. The time spent by an operator instance in deserialization, processing, and serialization activities.

Useful time excludes time spent waiting on input or output. Such waiting does occur in practice, for different reasons depending on the design of the reference system. In Flink, an operator instance may block on input when the input buffers are empty, or on output when there is no free space in the (bounded) output buffers. In Timely, operator instances may continuously "spin" checking their input queues until new records appear. In Heron, instances may be forced to wait due to a backpressure signal from a slow downstream operator.

In all cases, the useful time amounts to the time an operator instance runs for if executed in an *ideal* setting where it never has to wait to obtain input or push output. In general, useful time differs from the total observed time the instance needs to process and output records, and plays a key role in solving the problem of § 3.1.

Based on this distinction, we define the *true* processing and output rate of an operator instance as follows:

True Rates. The true processing (resp. output) rate corresponds to how many records an operator instance can process (resp. output) per unit of useful time.

Intuitively, the true rates denote the capacity of the operator instance, i.e. the *maximum* processing and output rate the instance could sustain for the current workload. In contrast, the observed rates are those measured by simply counting the number of records processed and output by the instance over a unit of elapsed time, which might include waiting. More precisely:

Observed Rates. The observed processing (resp. output) rate corresponds to how many records an operator instance processes (resp. outputs) per unit of observed time.

Although the observed rates are more sensitive to changing workloads, due to the potential change in waiting time, true rates typically have lower variance, especially within short time periods (e.g. a few seconds of execution) as they represent the average "cost" to process and output a single record. This cost naturally can depend on factors like the size of the record, its content, and the state maintained by the operator instance, but the average cost can be estimated using appropriate instrumentation of the operator without needing to saturate it.

We define all rates in our model relative to windows of size W seconds of observed time. We denote the useful time for an operator instance W_u, where $0 \leq W_u \leq W$. More precisely:

$$\lambda_p = \frac{R_{\text{prc}}}{W_u} \quad (1) \qquad \lambda_o = \frac{R_{\text{psd}}}{W_u} \quad (2)$$

$$\widehat{\lambda}_p = \frac{R_{\text{prc}}}{W} \quad (3) \qquad \widehat{\lambda}_o = \frac{R_{\text{psd}}}{W} \quad (4)$$

Symbol	Description
G	logical dataflow graph
m	number of operators in G ($m > 1$)
n	number of source operators in G ($0 < n < m$)
W	size of a window in time units (observed time)
W_u	useful time for an operator instance in W
R_{prc}	number of records pulled from the input in W
R_{psd}	number of records pushed to the output in W
$\widehat{\lambda}_p$	observed processing rate of an operator instance
$\widehat{\lambda}_o$	observed output rate of an operator instance
λ_p	true processing rate of an operator instance
λ_o	true output rate of an operator instance
o_i	i-th operator in G (in topological order)
p_i	number of instances of the i-th operator
$o_i[\lambda_p]$	aggregated true processing rate of the i-th operator
$o_i[\lambda_o]$	aggregated true output rate of the i-th operator
π_i	optimal number of instances for the i-th operator

Table 2: Notation used in this paper.

where λ_p and λ_o are the true processing and output rate respectively (undefined when $W_u = 0$), $\widehat{\lambda}_p$ and $\widehat{\lambda}_o$ are the observed processing and output rates (undefined when $W = 0$), and R_{prc} (resp. R_{psd}) is the total number of records the instance processed (resp. pushed) in W.

For a specific operator instance and a window W, the following inequalities hold: $0 \leq \widehat{\lambda}_p \leq \lambda_p$ and $0 \leq \widehat{\lambda}_o \leq \lambda_o$, since $0 \leq W_u \leq W$. In general, the less an operator instance waits on its input and output the smaller the difference between the observed and true rates. Table 2 summarizes the notation.

We instantiate the model with (i) the logical dataflow graph G, (ii) the output rate of each data source, and (iii) the true processing and output rates (λ_p and λ_o) of each operator instance. G is static (known at compile time) and does not change during execution, since the logical dataflow is unaffected by the scaling decisions. The output rates of the data sources are continuously monitored outside the reference system, and the true rates of the operator instances are computed based on system-generated traces, as we explain in § 4.1. The output of DS2 is the optimal parallelism, i.e. number of instances, for each logical operator in the graph G, subject to the constraints of the problem in § 3.1.

The calculation proceeds as follows: let A be the adjacency matrix of G. $A_{ij} = 1$ iff the i-th operator outputs to the j-th operator, otherwise $A_{ij} = 0$. We consider operators numbered in topological order from $i = 0$ to $i = m-1$, where m is the total number of operators in G. This means that if o_i outputs to o_j and, hence, $A_{ij} = 1$, then $0 \leq i < j < m$. Since G is acyclic (cf. § 3.1), there is a topological ordering of its nodes and it can be computed in linear time.

For a time window W and operator o_i with p_i

instances, $p_i \geq 1$, we define the *aggregated* true processing and output rates $o_i[\lambda_p]$ and $o_i[\lambda_o]$ as:

$$o_i[\lambda_p] = \sum_{k=1}^{k=p_i} \lambda_p^k \quad (5) \qquad o_i[\lambda_o] = \sum_{k=1}^{k=p_i} \lambda_o^k \quad (6)$$

where λ_p^k and λ_o^k are the true processing and output rates of the k-th instance of o_i, as given by Eq. 1 and Eq. 2.

The optimal level of parallelism π_i for an operator o_i is now computed using the ratio of the aggregated true output rate of its upstream operators (when they keep up with their inputs) to the average true processing rate per instance of o_i. More formally:

$$\pi_i = \left\lceil \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^* \cdot \left(\frac{o_i[\lambda_p]}{p_i} \right)^{-1} \right\rceil, n \leq i < m \quad (7)$$

where m is the total number of operators in G, and n is the number of source operators in G, $0 < n < m$.

$o_j[\lambda_o]^*$ denotes the aggregated true output rate of an operator o_j, when o_j itself and all operators before it (in topological order) are deployed with their optimal parallelism to keep up with their inputs. It is recursively computed as follows:

$$o_j[\lambda_o]^* = \begin{cases} o_j[\lambda_o] = \lambda_{\text{src}}^j, & 0 \leq j < n \\ \frac{o_j[\lambda_o]}{o_j[\lambda_p]} \cdot \sum_{\forall u: u<j} A_{uj} \cdot o_u[\lambda_o]^*, & n \leq j < m \end{cases} \quad (8)$$

where λ_{src}^j is the output rate of the j-th source operator, $0 \leq j < n$.

Note that $o_j[\lambda_o]^*$ depends on (i) the ratio $\frac{o_j[\lambda_o]}{o_j[\lambda_p]}$, which denotes the selectivity of o_j, and (ii) the estimated true output rate of the upstream operators ($\forall u : u < j$ in the summation). The latter implies that $o_j[\lambda_o]^*$ and, hence, π_i can be efficiently computed for *all* operators in the dataflow with a *single traversal* of G, starting from the sources. This property is important in practice, as it allows us to estimate the required number of instances for all operators in the dataflow in the same scaling decision.

3.3 Assumptions

DS2 makes the following assumptions about the dataflow system it is controlling:

Data-parallel operators. An operator's output can be produced by partitioning its input on a key and applying the operator logic separately to each partition. Other than this, the operator's internal logic can be any user-defined function. Data-parallelism is essential for effective scaling decisions: executing multiple operator instances entails partitioning its state into chunks of data processed in parallel. In contrast, non-data-parallel operators do not benefit from scaling. System users could tag such operators for DS2 to ignore, or their lack of parallelism could be identified online by comparing input and output rates before and after scaling. As with existing systems, we leave the integration of such operators for future work.

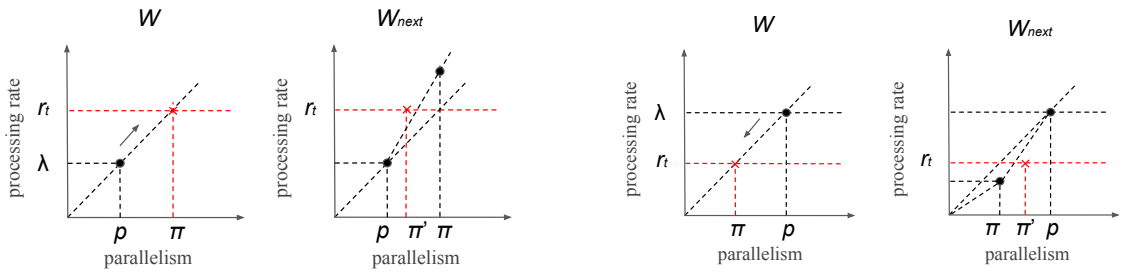
No data or computation imbalance. Our scaling model addresses neither data skew across operator instances nor computational stragglers. Both these types of imbalance can trigger backpressure which cannot be tackled by changing the degree of parallelism of one or more operators. Several robust solutions to the skew and straggler problems exist and have been incorporated into real systems. Techniques such as partial key grouping [35] introduced in Storm [34] and further evaluated in [25], and work-stealing for straggler mitigation in MapReduce [28] and Google Dataflow [26] are complementary to DS2. In § 4.2 we describe how DS2 could be integrated in a general controller for streaming applications which would not only handle dynamic scaling but also include skew and straggler handling components.

Stable workloads during scaling. Like existing scaling mechanisms, DS2 operates with the understanding that workload characteristics remain stable between a scaling decision being made and the new parallelism configuration being deployed. This window is the time taken for DS2 to make a decision (which we evaluate in § 5) plus the time to deploy the new configuration, which depends on the dataflow system in use. In practice, we find this timescale is dominated by the latter in current systems.

3.4 Properties

DS2 estimates the optimal parallelism for each operator assuming perfect scaling, that is, the true processing and output rates change linearly with the number of instances. In general, however, true rates are described by non-linear, most commonly sub-linear functions. Super-linear speedups are possible [16] (e.g. when state fits in cache after a scale-up) but are rare in practice. When this "perfect scaling" assumption holds, DS2 estimations (Eq. 7) correspond to bounds and the model enjoys the following two properties:

Property 1. No overshoot: a scale-up decision will not result in over-provisioning. The estimated optimal number of instances π_i for an under-provisioned operator is always less than or equal to the minimum required to keep up with the target rate $r_t = \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Eq. 7.



(a) No overshoot when scaling up

(b) No undershoot when scaling down

Figure 4: Given a target rate r_t and aggregated true processing rate λ which does not scale super-linearly, our model guarantees no over-provisining when scaling up and no under-provisioning when scaling down.

Property 2. No undershoot: a scale-down decision will not result in under-provisioning (and, hence, backpressure). The estimated optimal number of instances π_i for an over-provisioned operator is always greater than or equal to the minimum needed to keep up with the target rate $r_t = \sum_{\forall j: j<i} A_{ji} \cdot o_j[\lambda_o]^*$ in Eq. 7.

Figure 4 shows hypothetical scale up and scale down scenarios, each during two consecutive time windows, W and W_{next}. Consider an operator initially configured with parallelism p and aggregated processing rate $\lambda < r_t$, where r_t is the target rate, as shown in Figure 4a (left). Assuming linear scaling, our model assigns π instances to reach the target rate r_t. Property 1 states that there exists no $\pi' < \pi$ such that π' matches r_t. Indeed such a π' can only exist in W_{next} if the aggregated processing rate scales super-linearly, as shown in Figure 4a (right).

Similarly, if an operator is initially configured with parallelism p and aggregated processing rate $\lambda > r_t$, as in Figure 4b (left), our model assigns $\pi < p$ instances to scale down to r_t. Property 2 states that there exists no $\pi' > \pi$ such that π' matches the target r_t. As shown in Figure 4b (right), such a π' would violate the assumption of non-superlinear aggregated true processing rate.

Together, these properties imply that repetitive applications of DS2 do not oscillate: they will monotonically converge to the target rate from below or above, ensuring stability without the need to blacklist previous decisions, and simplifying the scaling mechanism significantly.

When true rates are linear and the target rate r_t is accurately estimated for each operator, DS2 converges in at most one step. When one of these two conditions does not hold, for example, true rates do not scale well due to other overheads (e.g. worker coordination) or dataflow operators have data-dependent output rates, DS2 needs more steps to converge to a stable configuration. In each of these steps, DS2 tries to minimize the error of its previous decision to get closer to the target, as any typical controller does. We omit the details of this process here and we only show empirically (in § 5.4) that DS2 needs at most three steps to converge in all our experiments. Further reducing the number of steps requires good approximation of non-linear rates, which could be gradually learned by DS2 using machine learning techniques, opening an interesting direction for future work.

4 Implementation and deployment

The DS2 controller consists of about 1500 lines of Rust running as a standalone process. Here we describe the instrumentation requirements it imposes and discuss the issues encountered integrating it with three different stream processing engines: Flink, Timely dataflow, and Heron.

4.1 Instrumentation requirements

DS2 requires a subset of the instrumentation required by bottleneck detection tools for stream processors like SnailTrail [23]. The stream processor must periodically collect and report records processed, records produced, and useful time (serialization, deserialization, processing) or waiting time per operator instance.

Flink gathers some of the metrics required by DS2 (e.g. records read and produced) by default but we extended its runtime so that each operator instance maintains local counters for (de)serialization and processing duration as well as for buffer wait time, reporting them to DS2 in configurable intervals. For record-at-a-time systems like Flink, tracking and emitting metrics for every record might incur significant overhead. Instead, we aggregate measurements per input buffer for all operators, except for sources where we aggregate per output buffer. Specifically, we have implemented a `MetricsManager` module which is responsible for gathering, aggregating, and reporting policy metrics. We assign one `MetricsManager` instance per parallel thread executing operator logic. Each

thread maintains local counters for records read, records produced, (de)serialization duration, processing duration, and waiting for input and output buffers. Source operator instances send their current local counters to the `MetricsManager` every time an output buffer gets full and regular operator instances send their local counters every time they receive a new input buffer for processing. The `MetricsManager` maintains a data structure with the current aggregate metrics of its operator instance and reports them to the outside world in configurable intervals.

Timely [32] outputs raw tracing information, which we aggregate in configurable intervals to produce metrics for DS2. We use a similar `MetricsManager`, as in Flink, which receives streams of logged events coming from Timely workers and aggregates them on the fly. Each Timely worker logs individual events of different types, such as scheduling an operator or sending a message over a data channel, along with their timestamp in nanoseconds. Recall that operator instances in Timely are not blocked on their input or output queues; instead, they are continuously spinning, i.e. they are scheduled for execution (in a round-robin fashion) even if there are no data records to process. Spinning results in a huge amount of scheduling event logs, which quickly saturate the `MetricsManager`, although most of these logs are not needed for computing the true rates. To tackle this problem, we modified Timely's logger to trace and send to the `MetricsManager` only the "useful" scheduling events, i.e. those that correspond to an operator instance doing some "useful work" for the actual computation.

Heron also by default outputs detailed, aggregated metrics [22], which are periodically collected and fed into DS2. The aggregation window depends on how frequently Heron samples its metrics and can be configured.

4.2 Integration with stream processors

DS2 is mechanism-agnostic and can be integrated with any stream processor capable of dynamically varying resources and migrating state. Figure 5 shows the high-level architecture of such an integration. Instrumented streaming jobs periodically report metrics to a repository. DS2 consists of a *Scaling Policy* component implementing the model of § 3.2, and a *Scaling Manager* monitoring the repository, invoking the policy when new metrics are available, and sending scaling commands to the stream processor.

While DS2 currently only offers scaling functionality, it could be easily extended with skew and straggler mitigation techniques as shown in Figure 5. In this case, the system would consist of multi-purpose Manager and Policy components, where the first detects the problem type (e.g., presence of skew) and the latter invokes the appropriate policy. Note that DS2 collects metrics from each operator instance separately, thus skew detection can be effortlessly implemented by the Manager.

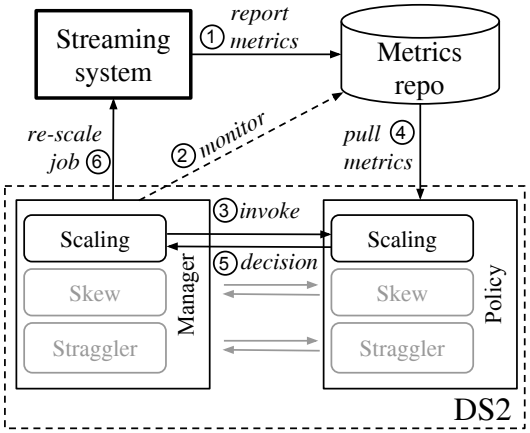


Figure 5: DS2 integration with streaming systems

We have integrated DS2 with Apache Flink, which employs a simple scaling mechanism: when instructed, Flink takes a *savepoint*, a consistent snapshot of the job state, halts the computation, and redeploys it with the updated parallelism [24]. We demonstrate this integration in action and evaluate it under a dynamic source rate in § 5.3.

4.2.1 Scaling Manager

Operational issues in real deployments that are not captured by the model must be handled by the implementation instead. To deal with factors that might affect scaling decisions in practice, the Scaling Manager provides the following configuration parameters:

Policy interval defines the frequency with which metrics are gathered and the policy invoked. Tuning the policy interval allows the scaling manager to aggregate metrics meaningfully, e.g. to ensure enough data is available to compute averages for processing and output rates. Long intervals give stable metrics but also increase reaction time. The interval must also be tuned based on the reconfiguration mechanism of the reference system. In our experiments, we found 5–30s intervals reasonable for Flink and Timely. For Heron, we found the default 60s suitable.

Warm-up time is the number of consecutive policy intervals ignored after a scaling action, since rate measurements can be unstable at the start of a computation or before backpressure builds up.

Activation time specifies when DS2 applies a scaling decision, as the number of consecutive policy decisions considered by the scaling manager before issuing a scaling command. Activation time plus an appropriate

policy interval mitigates the effects of irregularities in some streaming computations, such as non-incremental tumbling windows or data-dependent operators. For instance, consider naively-implemented window operators that buffer records and only apply the computation logic after the window fires. As long as input is simply assigned to a window, the operator's processing rate will appear high but once the window fires and the actual computation is performed the processing rate will suddenly drop. DS2 can consider several consecutive policy decisions and, for example, compute the maximum or median parallelism across intervals before applying a scaling action.

Target rate ratio defines a maximum allowed difference between the observed source rate achieved by the policy and the target rate, addressing the practical issue that processing and output rates might be affected by overheads not captured by instrumentation. For instance, adding workers to a distributed computation might incur higher coordination, channel selection cost, or resource contention, and so a computation might need more resources to achieve the target rate than the policy indicates. DS2 estimates the additional resources required by computing the ratio between the currently achieved rate and the target rate.

4.2.2 Practical considerations

DS2 also ignores minor changes (e.g. changing an operator's parallelism by one or two), which can be triggered by noisy metrics. External disruptions, such as garbage-collection in Java-based systems or disk I/O, can also influence rates measurements. For example, when integrating DS2 with Flink, we took care to properly configure task managers, heap memory, and network buffers. We are also aware that system performance might degrade after a scaling action (though we have not observed this in practice). If this were to happen, DS2 rolls back to the previous configuration. Similarly, consecutive decisions resulting in very small improvements indicate a performance issue (e.g. data skew, stragglers) that cannot be improved by scaling. DS2 can limit the number of decisions to prevent further reconfiguration.

4.2.3 DS2 in the presence of skew

Even though the scaling model assumes no data imbalance and the current implementation of DS2 does not offer skew mitigation functionality, it is worth discussing how the system behaves if skew actually appears in a streaming application it is controlling. In such a case, the system makes a scaling decision assuming data balance (§ 3.3) by averaging true processing and output rates. Thus, DS2 proposes a configuration which might not meet the target throughput but at the same time will not over-provision the system. Further, due to DS2's ability to limit the number of decisions (§ 4.2.2), the policy is guaranteed to converge. We have verified the above behavior experimentally on Flink varying the skew parameter in the Dhalion benchmark from 20% to 50% and 70%. In all cases, DS2 converged after two steps to the configuration which would be optimal if there was no skew, but which in this experiment did not meet the target throughput.

4.3 Execution model independence

DS2's policy can be applied on streaming systems regardless of their execution model. In Flink and Heron each dataflow operator is assigned a number of worker threads that define its level of parallelism, i.e. the number of parallel instances executing the operator's logic. In this case, Eq. 7 can be directly used to configure operator parallelism independently. In Timely, on the other hand, parallelism is configured globally for the whole dataflow. Each worker runs every operator in the dataflow graph according to a round-robin scheduling strategy.

For Timely, DS2 estimates the optimal number of total workers by summing up the optimal level of parallelism, as given by Eq. 7, for all operators in the dataflow. The intuition here is simple: an operator that needs π_i instances to keep up with its input actually needs $\pi_i \cdot 100\%$ computing power per unit of time. In an execution model like Timely's where operators share computing resources (worker threads), the total computing power needed so that the system can keep up with its input is $\sum_{\forall i} \pi_i \cdot 100\%$. We experimentally validate the accuracy of DS2 decisions on Timely in § 5.5.

5 Experimental evaluation

Our evaluation covers DS2 in use with three different streaming systems: Heron, Flink, and Timely Dataflow. We start our evaluation by comparing DS2 with the state-of-the-art Dhalion scaling controller used in Heron, with the benchmark in the original Dhalion publication [13]. We then demonstrate DS2 in action through end-to-end, dynamic scaling experiments with Flink, followed by measurements of DS2 convergence and accuracy in using both Flink and Timely. Finally, we evaluate the overhead of the instrumentation used by DS2.

5.1 Setup

We run all Flink and Timely experiments on up to four machines, each with 16 Intel Xeon E5-2650 @2.00GHz cores and 64GB of RAM, running Debian GNU/Linux 9.4. We use Apache Flink `1.4.1` configured with 12 TaskManagers, each with 3 slots (maximum parallelism

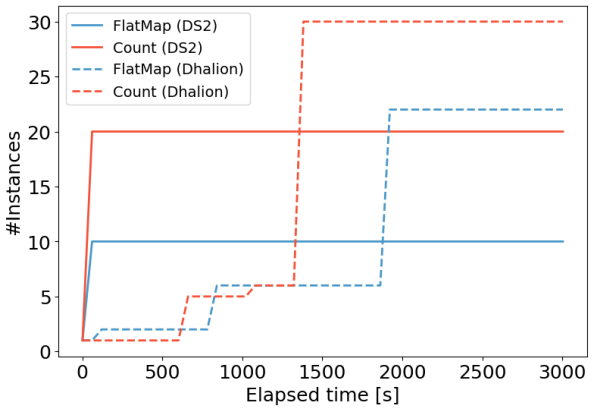


Figure 6: Comparison of DS2 vs Dhalion on Heron using the word count dataflow of [13].

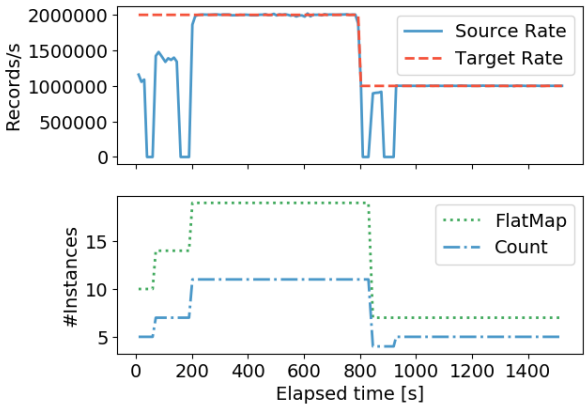


Figure 7: Dynamic scaling experiment with Flink using DS2 on the word count dataflow of [13].

per operator = 36), and Timely Dataflow `0.5.0` compiled with Rust `1.24.0`. For the comparison experiment, we run Heron `0.17.8` on a four socket-machine equipped with AMD Opteron 6276, with 64 threads in total and 256GiB of memory.

To demonstrate generality across diverse computations and streaming operators, we selected six queries from the Nexmark benchmarking suite of Apache Beam [42, 36, 37]. Specifically, we test the policy with Queries 1–3, 5, 8, and 11, which contain various representative streaming operators: stateless streaming transformations, i.e. map and filter in **Q1** and **Q2** respectively, a stateful record-at-a-time two-input operator (incremental join) in **Q3**, and various window operators: sliding window in **Q5**, tumbling window join in **Q8**, and session window in **Q11**. These queries specify computations both in processing and event time domains [5]. For the comparison with Dhalion (§ 5.2) and the end-to-end experiment on Flink (§ 5.3), we use the wordcount dataflow as specified in Dhalion's paper [13].

5.2 DS2 vs Dhalion on Heron

We compare the accuracy and convergence steps of DS2 with Dhalion, recreating the benchmark in [13].

We run Heron with Dhalion and its dynamic resource allocation policy enabled. The source operator of the three-stage wordcount topology (Source, FlatMap, Count) produces sentences at a fixed rate of 1M per minute. The FlatMap and Count operators are rate-limited to simulate bottlenecks: each FlatMap instance splits at most 100K sentences per minute, and each Count instance counts up to 1M words per minute (the same ratios as in the Dhalion paper). We start under-provisioned with one instance per operator and let Heron stabilize without backpressure.

We have already seen how the source rate evolves to match the target throughput in this experiment in Figure 1. Figure 6 shows the parallelism of FlatMap and Count over time, from the start until convergence. Dhalion makes six scale-up decisions (each involving a single operator) and reaches a stable configuration with 22 FlatMap instances and 30 Count instances after 2000 seconds.

We then apply DS2 on the same initial under-provisioned configuration using a 60*s* decision interval, no warm-up, one interval activation time, and 1.0 target ratio (cf. § 4). DS2 indicates a required parallelism of 10 for FlatMap and 20 for Count, which indeed is the minimum configuration that handles 1M sentences per minute. Note that DS2 correctly estimates the optimal parallelism in a single step, after only one minute of collecting the default Heron performance metrics.

Dhalion requires several re-configuration steps, each affecting a single operator, and reaches a final configuration that is significantly over-provisioned, even in this simple wordcount dataflow. In contrast, DS2 correctly identifies the optimal configuration in a single step and two orders of magnitude less time than Dhalion.

Besides those discussed in § 2, another reason Dhalion takes so long to reach a backpressure-free configuration is that its reaction time depends on the size of the operator queues. By default, Heron has a 100MiB buffer per operator queue, which may take some time to fill (depending on the workload) before backpressure kicks in and Dhalion can react. In contrast, DS2 only depends on the decision interval where metrics are aggregated, arbitrarily specified by the user and typically much smaller.

5.3 DS2 on Flink

We now show DS2 driving Apache Flink, in order to demonstrate the benefits of DS2 when combined with

	Bids		Auctions		Persons	
	Flink	Timely	Flink	Timely	Flink	Timely
Q1	4M	5M	—		—	
Q2	4M	5M	—		—	
Q3	—		500K	3M	100K	800K
Q5	500K	2M	—		—	
Q8	—		420K	4M	120K	4M
Q11	1M	9M	—		—	

Table 3: Target source rate (records/s) configuration for the Nexmark queries on Apache Flink and Timely.

a fast re-configuration mechanism such as that in Flink.

Here, DS2 uses a 10*s* decision interval, 30*s* warm-up time, one interval activation time, and 1.0 target ratio. DS2 hence ignores the first three decisions after re-configuration, applying a decision immediately after.

We use the same wordcount dataflow as before, this time with two phases corresponding to scale-up and scale-down scenarios respectively. In phase 1, the source rate is 2M sentences per second and Flink starts under-provisioned with 10 FlatMap instances and 5 Count instances. In this state, FlatMap can not keep up with the source rate, neither can Count handle FlatMap's output rate. Once Flink has reached a backpressure-free configuration, we keep the source rate stable for 10 minutes. During the second phase, we decrease the source rate to 1M sentences per second and keep it stable for another 10 minutes.

Figure 7 shows observed source rate and operator parallelism over time. DS2 applies two scale-up actions. First, at 40*s* it re-deploys the dataflow with 14 FlatMap instances and 7 Count instances. This happens right after the warm-up and activation time, and Flink takes around 30*s* to snapshot state and restart from the savepoint [24].

At 150*s* DS2 acts again to increase FlatMap to 19 and Count to 11 instances. This time Flink takes about 50*s* to redeploy the backpressure-free configuration at 200*s*.

At 803*s* (3*s* into the second phase) DS2 reacts to the reduced source rate by reducing the configuration to 7 FlatMap and 4 Count instances at 845*s*. At 900*s* it makes a final decision to increase Count parallelism by one, and Flink successfully applies the change at 930*s*, reaching the new optimal configuration.

This shows that DS2 plus an efficient re-configuration mechanism can offer robust dynamic scaling for streaming dataflows, allowing the reference system to react to changes in its workload in just a few seconds – significantly faster than any other systems we are aware of.

5.4 Convergence

We now show DS2 convergence from both over- and under-provisioned states on more complex dataflows. We use the same Flink configuration as before, and execute each query with fixed source rates (cf. Table 3) and initial configurations of varying parallelism. We run each query-configuration combination for 5 minutes and evaluate DS2 with 30*s* decision interval, 30*s* warm-up time, 1.0 target ratio, and five intervals activation (i.e. we consider the policy to have converged if the decision is unchanged over 5 consecutive intervals).

Table 4 shows the indicated parallelism per decision step for the main operator of each query on Flink. Note that queries **Q3**, **Q5**, **Q8**, and **Q11** include many operators, but we show results for the main operator of each for simplicity. DS2 converges in one step for simple queries and initial configurations close to optimal (e.g. **Q1** with parallelism 12), and in at most three steps for complex queries and initial configurations far from optimal (e.g. **Q5** with initial parallelism 8).

In all cases, DS2 takes at most three steps to converge. It needed three steps in 3 experiments (with **Q2**, **Q5**, and **Q11**), two steps in 14 experiments, and a single step in 19 out of 36 total experiments. We also ran the same queries using Timely Dataflow and the results were similar.

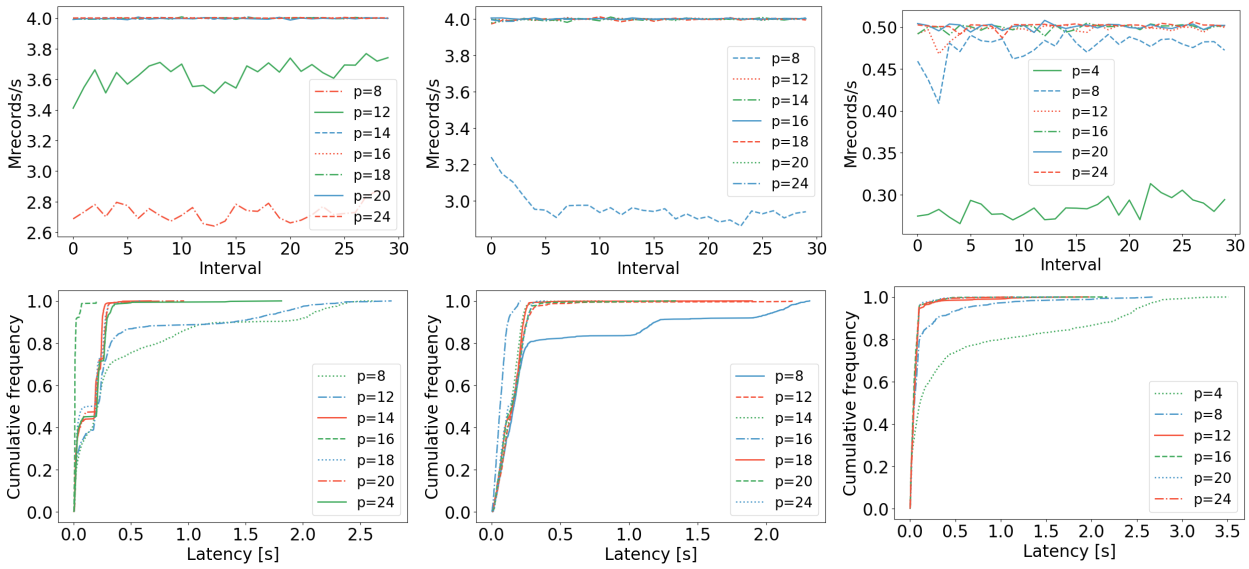
This shows that DS2 provides two important SASO properties: *stability* and *short settling time*.

Intuitively, one DS2 step moves close to optimal by estimating ideal linear scaling (§ 3.4). For far-from-optimal initial configurations, the second step "refines" this decision with a more accurate measurement, and the third step compensates for uncaptured overheads.

5.5 Accuracy

We next show accuracy: DS2 converges to configurations that exhibit no backpressure (and thus keep up with the source rates) while minimizing resource usage. In particular, we show that for a given dataflow, fixed input rate, and initial configuration, DS2 identifies the optimal parallelism regardless of whether the job is initially under- or over-provisioned. We further show that there exists no other backpressure-free configuration with lower parallelism than the one DS2 computes. Finally, we show that this configuration gives low latency by minimizing waiting time per operator instance.

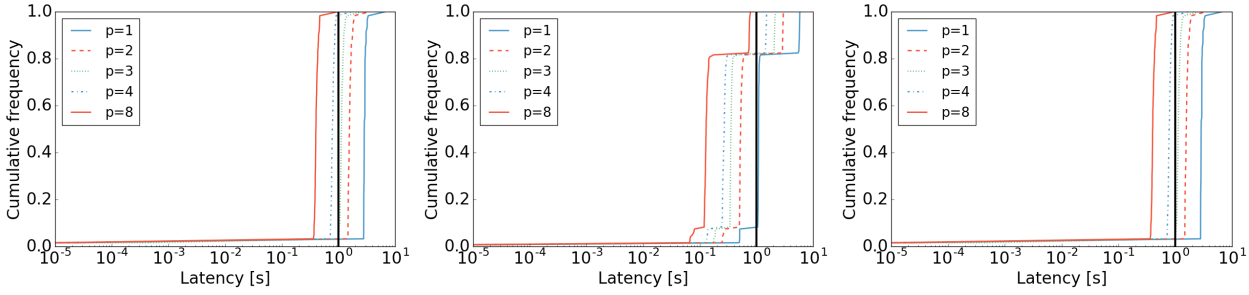
We set source rates as in Table 3 and parallelism given by the convergence experiment. Figure 8 plots observed source rates (top) and per-record latency (bottom) for the main operator of each Nexmark query on Flink with different configurations. For queries with two sources (**Q3** and **Q8**), we show results for the higher-rate source (results for the low-rate sources are similar). In all cases, DS2 successfully identifies the lowest parallelism that can keep up with the source rate. Further increasing the parallelism does not significantly improve latency and would waste resources, while lower parallelism would



(a) [Q1] Indicated parallelism: 16 (b) [Q2] Indicated parallelism: 14 (c) [Q3] Indicated parallelism: 20

(d) [Q5] Indicated parallelism: 16 (e) [Q8] Indicated parallelism: 10 (f) [Q11] Indicated parallelism: 28

Figure 8: Observed source output rates and per-record latency CDFs for different configurations of the Nexmark operators on Apache Flink.

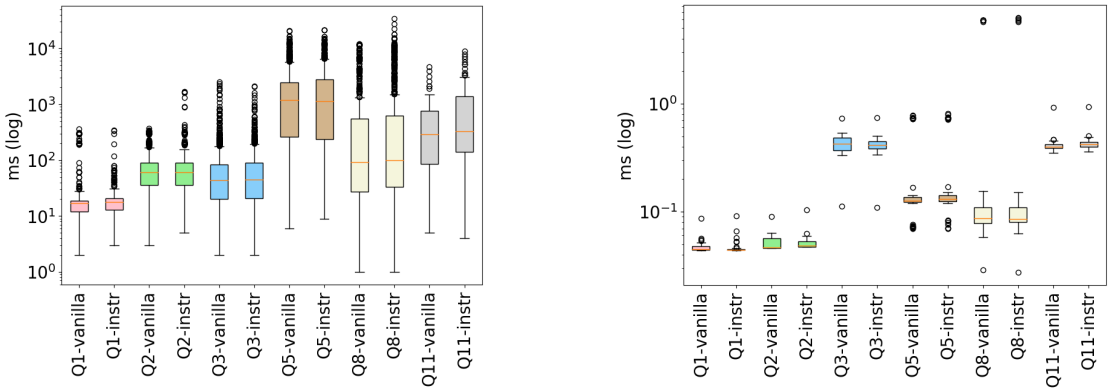


(a) [Q3] Indicated parallelism: 4 (b) [Q5] Indicated parallelism: 4 (c) [Q11] Indicated parallelism: 4

Figure 9: CDFs of per-epoch latencies for different configurations of the Nexmark operators on Timely.

Initial configuration	Q1	Q2	Q3	Q5	Q8	Q11
8	12→**16**	11→13→**14**	16→**20**	14→15→**16**	**10**	12→22→**28**
12	**16**	**14**	18→**20**	**16**	**10**	22→**28**
16	**16**	12→**14**	**20**	**16**	8→**10**	26→**28**
20	**16**	13→**14**	**20**	14→**16**	8→**10**	**28**
24	**16**	**14**	**20**	14→**16**	8→**10**	**28**
28	**16**	**14**	**20**	13→**16**	8→**10**	**28**

Table 4: DS2 convergence steps for Nexmark queries on Flink. Values are the level of parallelism of the main operator of each query. Leftmost column shows initial parallelism (from 8 to 28 instances); subsequent columns show optimal level of parallelism as estimated by DS2 in each step. Final decisions converged to by DS2 are highlighted.



(a) Flink instrumentation overhead

(b) Timely instrumentation overhead

Figure 10: Policy instrumentation overhead for the Nexmark queries of Table 3 with instrumentation disabled (`vanilla`) and enabled (`instr`) for both Flink (10a) and Timely (10b).

cause backpressure.

Timely does not have a backpressure mechanism so data sources are never delayed and the observed source rates are always equal to the initial fixed rate (instead, queues grow when the system cannot keep up). We therefore simply show CDFs of per-epoch latencies with different configurations for Timely. Figure 9 shows these for **Q3**, **Q5**, and **Q11**; results are similar for other queries. Each epoch in the CDFs corresponds to 1s of data, which must be processed in less than 1s. The optimal parallelism indicated by DS2 is $p = 4$ in all queries, regardless of the starting configuration. For **Q3** (left) and **Q11** (right), $p = 4$ is clearly the configuration that can keep up with the 1s target (vertical line in the plots) using minimum required resources. For **Q5**, 18% of the epochs are above the target by up to 0.5s. Here, the larger percentage of epochs that cannot keep up is because of the window operator, which stashes data and then forwards it at certain time points. This manifests as load spikes, which require additional resources for the system to keep up. Longer decision intervals smooth out the spikes but tend to affect policy decisions towards higher optimal configurations, which is why DS2 indicated $p = 4$ (cf. § 4.2).

In summary, DS2 identified optimal configurations in all experiments and never overshot (provisioned more resources than needed), thereby exhibiting the remaining two SASO properties: *accuracy* and *no overshoot*.

5.6 Instrumentation overhead

Finally, we evaluate instrumentation overhead. We run the Nexmark queries for 5 minutes with source rates from Table 3 and a 10s decision interval — the smallest we use in this paper, which results in the most frequently aggregated logs and has the highest potential overhead on the system performance.

We measure per-record latency in Flink using its built-in metric and per-epoch latency in Timely using 1s event-time epochs. Figure 10 shows boxplots for both systems. Individual columns show latency with logging completely off (vanilla) and instrumentation activated (instr). Overheads are small: at most 13% on Flink (40ms absolute difference) and at most 20% on Timely (5ms absolute difference) across all queries. Performance penalties are an acceptable trade-off for a good scaling policy, and could be further reduced with a larger decision interval and pre-

aggregation of metrics. Note that Heron incurs no overhead since it gathers the required metrics by default.

6 Conclusion

In this paper we have described and evaluated DS2, a novel automatic scaling controller for distributed streaming dataflows. Unlike existing scaling approaches, which rely on coarse-grained metrics and simplistic models, DS2 leverages knowledge of the dataflow graph, the computational dependencies among operators, and estimates the operators' true processing and output rates.

DS2 uses a general performance model that is mechanism-agnostic and broadly applicable to a range of streaming systems. We have implemented DS2 atop different stream processing engines: Apache Flink, Timely Dataflow, and Apache Heron, and showed that it is capable of accurate scaling decisions with fast convergence, while incurring negligible instrumentation overheads.

An interesting question for future work is what kind of scaling and adaptation mechanisms are a good match for a controller like DS2. The efficiency of DS2's model means that responsiveness is often limited by the latency of the scaling mechanism of the stream processor (when it is not determined by the granularity of measurement). All the stream processors we test against implement scaling actions by checkpointing the dataflow, redeploying, and restoring from the checkpoint. A faster, more dynamic reconfiguration mechanism might allow DS2 to operate on shorter timescales than the tens of seconds it allows in current systems.

We will release DS2 as open source, together with all code and data used to produce the results in this paper.

Acknowledgements

We thank Nicolas Hafner for his help with the Nexmark queries implementation on Timely. We would also like to thank the anonymous OSDI reviewers for their insightful comments, and our shepherd Matei Zaharia for his guidance in improving the paper. This work was partially supported by the Swiss National Science Foundation, a Google Research award, and a gift from VMware Research. Vasiliki Kalavri is supported by an ETH Postdoctoral fellowship.

References

[1] Dynamic allocation in spark. `https://www.slideshare.net/databricks/dynamic-allocation-in-spark`, 2015.

[2] Dynamic resource allocation in spark. `https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation`, 2018.

[3] 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 *Cidr*, volume 5, pages 277–289, 2005.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. Mcveety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[6] E. Anderson and M. Dvorsky. Comparing Cloud Dataflow autoscaling to Spark and Hadoop. `https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop`, 2016.

[7] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, pages 189–200, 2017.

[8] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.*, 10(12):1718–1729, Aug. 2017.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, 38(4), 2015.

[10] M. D. de Assunção, A. D. S. Veith, and R. Buyya. Resource elasticity for distributed data stream processing: A survey and future directions. *CoRR*, abs/1709.01363, 2017.

[11] S. Desimone. Storage reimagined for a streaming world. `http://blog.pravega.io/2017/04/09/storage-reimagined-for-a-streaming-world/`, 2017.

[12] R. C. 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*, pages 725–736, 2013.

[13] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 2017.

[14] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

[15] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[16] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres. Dynamic configuration of partitioning in spark applications. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1891–1904, 2017.

[17] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[18] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[19] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[20] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 261–272, 2011.

[22] Apache Heron. `https://github.com/apache/incubator-heron` (accessed: April 2018).

[23] M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. Dimitrova, S. Wicki, Z. Chothia, and T. Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, 2018. USENIX Association.

[24] F. Hueske. Savepoints: Turning Back Time. `https://data-artisans.com/blog/turning-back-time-savepoints`, 2016.

[25] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *PVLDB*, 10(11):1286–1297, 2017.

[26] E. Kirpichov and M. Denielou. No shard left behind: dynamic work rebalancing in Google Cloud Dataflow (accessed: March 2018). `https://cloud.google.com/blog/big-data/2016/05/no-shard-left-behind-dynamic-work-rebalancing-in-google-cloud-dataflow`.

[27] 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*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[28] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[29] B. Lohrmann, P. Janacik, and O. Kao. Elastic Stream Processing with Latency Guarantees. In *Proceedings - International Conference on Distributed Computing Systems*, 2015.

[30] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, Dec. 2014.

[31] S. S. Manvi and G. K. Shyam. Resource management for infrastructure as a service (iaas) in cloud computing: A survey. *Journal of Network and Computer Applications*, 41:424 – 440, 2014.

[32] F. McSherry. A modular implementation of timely dataflow in Rust (accessed: April 2018). `https://github.com/frankmcsherry/timely-dataflow`.

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

[34] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 137–148, 2015.

[35] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 589–600, 2016.

[36] Apache Beam Nexmark benchmark suite. `https://beam.apache.org/documentation/sdks/java/nexmark`.

[37] NEXMark benchmark. `http://datalab.cs.pdx.edu/niagaraST/NEXMark`.

[38] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 233–244. IEEE, 2016.

[39] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.

[40] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, and M. Fu. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, SIGMOD '14, pages 147–156, New York, New York, USA, 2014. ACM Press.

[41] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 787–798. VLDB Endowment, 2006.

[42] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Benchmark for Queries over Data Streams DRAFT. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[43] Y. Wu and K.-L. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*, pages 723–734. IEEE, apr 2015.

[44] L. Xu, B. Peng, and I. Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 22–31, 2016.

[45] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.