DABS-Storm: A Data-Aware Approach for Elastic Stream Processing

Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, Nicoló Rivetti, Yann Busnel

DABS-Storm: A data-aware approach for elastic stream processing *

Roland Kotto Kombi[1], Nicolas Lumineau[2], Philippe Lamarre[1], Nicolo Rivetti[3], Yann Busnel[4]

[1]Univ Lyon, INSA de Lyon, LIRIS UMR5205, [2]Univ Lyon, University Claude Bernard Lyon 1, LIRIS UMR5205, [3]Technion, Israel Institute of Technology, [4]IMT Atlantique

Abstract. In the last decade, stream processing has become a very active research domain motivated by the growing number of stream-based applications. These applications make use of continuous queries, which are processed by a stream processing engine (SPE) to generate timely results given the ephemeral input data. Variations of input data streams, in terms of both volume and distribution of values, have a large impact on computational resource requirements. Dynamic and Automatic Balanced Scaling for Storm (DABS-Storm) is an original solution for handling dynamic adaptation of continuous queries processing according to evolution of input stream properties, while controlling the system stability. Both fluctuations in data volume and distribution of values within data streams are handled by DABS-Storm to adjust the resources usage that best meets processing needs. To achieve this goal, the DABS-Storm holistic approach combines a proactive auto-parallelization algorithm with a latency-aware load balancing strategy.

1 Introduction

With the proliferation of connected devices (smartphones, sensors, etc.), more and more data stream sources emit real-time data with fluctuations in input rate and value distribution over time [19]. Processing these Big Data streams (volume and velocity) in soft-real time (*i.e.,* low latency), satisfying end-user performance requirements, still raises several research problems.

To process a stream set, a user can submit a query to the execution infrastructure. This query, called a *continuous query* [7, 19], computes new results as new *stream elements* are generated by sources over time. Users define continuous queries through declarative languages [2,6,7,12] or, more imperatively, through a high-level language [28,41] (Java, Python, C, etc.). In any case, these continuous queries are usually turned into direct acyclic graphs (DAG) of operators, called *workflows* or *topologies*, corresponding to execution plans [1, 2, 28].

*This work has been partially supported by the project SocioPLug (ANR-13-INFR-0003) funded by the French National Research Agency, the Association Nationale Recherche Technologie (ANRt) `http://socioplug.univ-nantes.fr/index.php/SocioPlug_Project`

To generate timely results, a workflow requires some resources (CPU, RAM, bandwidth).The problem is that any evolution of the input streams (in input rate or value distribution) impacts the amount of ressources needed to process it. Furthermore, end-users usually require a good end-to-end latency and no data loss, regardless of any other consideration. Since, in general, evolutions of input streams cannot be captured through a-priori knowledge, a dynamic method is required that dynamically adapts the assigned resource according to evolution of needs. Such a method has to be as precise as possible. Indeed, whatever its imprecision, its consequences are negative. On the one hand, an under-provisioning may lead to *congestion*, implying reduced throughput and increased latency, or worse, data loss [1]. On the other hand, an over-provisioning induces resource and financial wastes, while potentially generating massive network overheads [39] and resource shortage.

Industrial [9, 20], open-source [4, 5] and academic [1, 2, 6, 8, 12, 13, 36] *stream processing engines* (SPE) have been developed to simplify stream management. Nevertheless, due to a lack of holistic and automatic strategies embracing all aspects related to elasticity, most of these solutions rely on user expertise and reactivity to face critical fluctuations in input rate. In particular, this is the case for the STORM family solutions.

To adapt provisioning, three linked problems have to be considered for each operator: parallelism degree, scheduling, and load balancing. Operator parallelism defines how many threads work together to process the incoming load of one operator. Note that, until an asymptote is reached, increasing the number of threads improves system performance. The scheduling strategy assigns threads to available processing units. Finally, the load balancing strategy distributes the incoming data among the available threads.

In this work, we aim at identifying and solving the issues raised by the dynamic adaptation of an SPE resource allocation while facing critical fluctuations in input rate and value distribution. Most existing SPEs integrate efficient automatic scheduling strategies designed to implement different objectives. For example, the STORM family includes RSTORM [28], TSTORM [39], and Stela [40] which respectively aim at finding the scheduling plan that reduces the number of active processing units, thus minimizing network traffic between processing units and avoiding processing bottlenecks due to input overload. To attain this goal, each strategy affects the scheduling plan so that data are processed with short latency. For example, in TSTORM [39], authors highlight overheads generated by network communications. This observation is reused in [28] and extended to resource usage to define an optimal scheduling plan, *i.e.,* a scheduling plan involving minimal computation overheads. In this paper, we focus on parallelism degree and load balancing management so as to propose a solution that is compatible with each of the scheduling strategies. Our goal is to obtain a preventive solution which adapts the system to data stream evolutions before problems occur. Furthermore, we expect as general a solution as possible, which does not depend on users whether for obtaining information from their experience or for triggering system adaptations. To reach this goal, we propose to build over two

already published solutions, AUTOSCALE [24] and OSG [29–31]. AUTOSCALE proposes a method to fix the parallelism degree of each operator of a workflow with an original data-driven approach, which considers the complete graph of operators and data streams in the workflow to avoid inconsistent local decisions that lead to rapid revisions and therefore significant system instability. An example of this is an operator starting a scale-in while it is apparent that its activity will augment soon due to an increase of the output stream(s) of upstream(s) operator(s). Unfortunately, AUTOSCALE presents some instability problems which had to be studied and fixed. OSG (ONLINE SHUFFLE GROUPING) deals with load balancing. Even if tuple processing times are not similar from one value to the other, OSG aims at reducing tuple completion times by carefully scheduling each incoming tuple.

The original contributions presented in this paper are three-fold:

1. An auto-parallelization strategy improving the approach presented in [24]. AUTOSCALE+, thanks to a better modeling of STORM effective resource usage, enables quicker deployment of adequate resources, thus improving system throughput and stability.
2. The integration of AUTOSCALE+ and OSG into DYNAMIC AND AUTOMATIC BALANCED SCALING FOR STORM (DABS-STORM), a holistic and automatized approach to parallelism and load balancing in stream processing systems, has been enabled due to their compatibility.
3. A thorough experimental evaluation of DABS-STORM highlighting its ability to process streams with critical fluctuations in input rate and value distribution for complex continuous queries. In addition, we compare DABS-STORM with well-known approaches from the literature.

In the remainder of this paper, section 2 presents the execution context from logical and physical points of view. We describe how continuous queries are processed over distributed infrastructures and the processing model. Section 3 presents the related work, reviewing the background on dynamic and elastic stream processing and the main elasticity mechanisms at infrastructure and query levels for handling variance in input load. Approaches for parallelism management and load balancing are described, respectively, in section 4 and section 5. Our original approach, DABS-STORM, is detailed in Section 6 while section 7 is devoted to its experimental evaluation.

2 System Model

2.1 Execution environment

To make things more concrete while introducing some notations, let us consider three continuous queries Q_1, Q_2 and Q_3 represented by workflows $\mathcal{W}1$, $\mathcal{W}2$ and $\mathcal{W}3$ with respective associated output streams S'_1, S'_2 and S'_3 (see Figure 1). A workflow $\mathcal{W} = (\mathcal{O}, \mathcal{V})$ is a direct acyclic graph where $\mathcal{O}$ is the set of operators and $\mathcal{V}$ the set of streams. Presented workflows are quite simple: $\mathcal{W}1$ is linear,

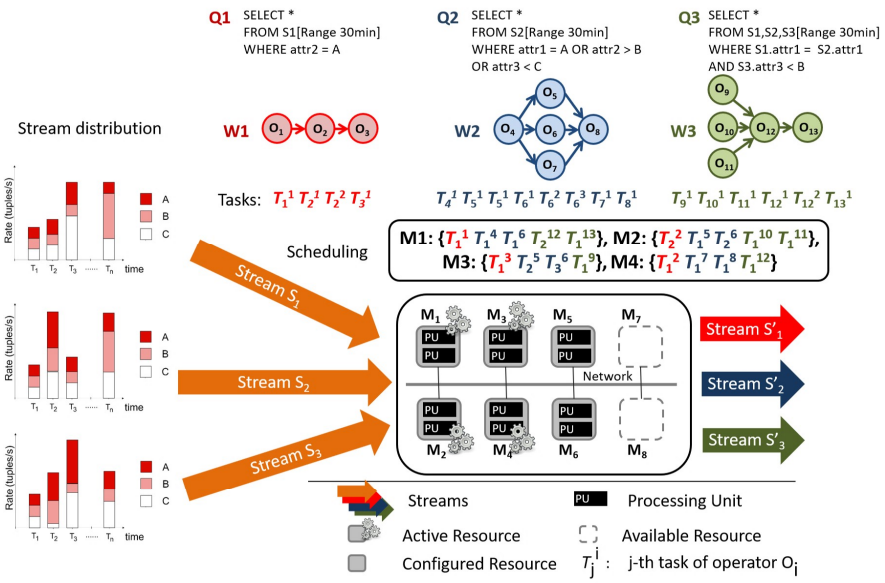


Fig. 1: Distributed stream processing.

$\mathcal{W}2$ is a diamond, while $\mathcal{W}3$ is a star. Despite their simplicity, these workflows are interesting to study. Indeed, they are general patterns used to build much more complex workflows [28]. Each workflow processes a set of input stream(s) which, in our example, is included in $\{S_1, S_2, S_3\}$.

A stream is a potentially infinite sequence of *tuples*, *i.e.,* key/value pairs, arriving over time. An input stream may have fluctuations in input rate and value distribution as shown on the left of Figure 1. It is worth noting the impact these fluctuations can have, not only on the processing time, but also on the *selectivity* of operators, *i.e.,* the ratio between the number of output and input tuples. This second point can be critical for operators such as joins [15, 37] as well as having direct impact on downstream operators.

Each operator $O_i \in \mathcal{O}$ is processed in parallel. The parallelism degree $d(O_i)$ of operator O_i corresponds to the number of tasks assigned to the operator. For instance, on Figure 1, operator O_2, executed by tasks T_2^1 and T_2^2, has a parallelism degree of $d(O_2) = 2$.

A scheduling strategy assigns tasks to the processing units, in this case eight available machines (M_1 to M_8). For instance, on Figure 1, the four tasks of the workflow $\mathcal{W}1$ are distributed on machines M_1 to M_4.

For a machine, three states are possible: *active*, *configured* and *available*. On Figure 1, machines M_1 to M_4 are active, and run some assigned tasks. Machines M_5 and M_6 are configured but inactive (no assigned tasks). Finally, M_7 and M_8

are available but not configured, which means the scheduler cannot assign tasks to them.

While DABS-Storm could be extended to handle heterogeneous machines, in this paper, for the simplicity's sake, we restrict the execution context to homogeneous machines (**H1**), *i.e.,* all machines have the same amount of CPU, RAM and bandwidth. Furthermore, we also assume that there is never resource starvation. In other words, there are always enough computational resources available to process input streams, no matter their input rates (**H2**). Thus, a loss of quality or performance cannot be ascribed to a lack of resources. To enforce **H2**, if the scenario does not ensure it, load shedding techniques [22,30] can be relied on, which drop some of the inputs in order to prevent buffer overflows or trashing.

Stream elements are assumed to be heterogeneous with respect to processing latency, depending on their values (**H3**). Consequently, DABS-Storm can handle homogeneous data streams as well as heterogeneous ones.

Finally, as we aim at proposing a generic solution supporting user-defined functions as well as common operators like filters and joins, we intend to deal with stateless and stateful operators. However, in this paper, we focus on stateless operators (**H4**). Indeed, it has been shown that solutions can also be useful for other kinds of operators [31], and most SPEs supporting stateful operators, like joins, provide a state management method while replicating these operators, such that we can rely on them for this part.

2.2 Processing model

Each operator O_i has a logical input stream $\sigma_i = \langle e_1, \ldots, e_q, \ldots, e_m \rangle$. Since operator O_i may be executed in parallel by $k = d(O_i)$ tasks $T_i^1, \ldots, T_i^k$, then each task receives a physical input sub-stream $\sigma_i^1, \ldots, \sigma_i^k$. Notice that $\sigma_i = \bigcup_{x \in [k]} \sigma_i^x$. Tuples of σ_i are assigned to a sub-stream, and thus to a task, according to a predefined load balancing strategy. We denote by $f(e)$ the unknown frequency[1] of tuple e, *i.e.,* the number of occurrences of e in the stream of size m. Before being processed, a tuple e_q is buffered in a FIFO input queue consumed by a task. The processing latency $w_i^x(q)$ of tuple e_q on the task T_i^x depends on the time complexity of O_i, on the computational power available to task T_i^x, and potentially, on the values of e_q attributes. Without loss of generality, we assume that tuples in a stream σ are identified by a single integer drawn from a large universe $[n] = \{1, \ldots, n\}$. In other words, tuples can be modeled as single values. The processing latency is modeled as an unknown function[2] of the value of e_q. The probability distribution of e_q values may vary over time. In a stable system the average processing latency of operator O_i can be defined as

$$\overline{W}_i = \frac{1}{|\sigma_i|} \sum_{x \in [k]} \sum_{e_q \in \sigma_i^x} w_i^x(q) \tag{1}$$

[1] This definition of *frequency* is compliant with the data streaming literature [7,35].

[2] The experimental evaluation relaxes the model by taking into account processing latency variance.

Let $\ell(q)$ be the completion time or end-to-end latency of e_q, *i.e.,* how much time it took for tuple e_q from the instant it was inserted into the assigned task's buffer to when it was processed by the tasks. Then we can define the average completion time for operator O_i as

$$\overline{L}_i = \frac{1}{|\sigma_i|} \sum_{e_q \in \sigma_i} \ell(q) \tag{2}$$

Table 1 summarizes the notation.

Workflow/Topology	$\mathcal{W}$
Workflow input and output streams	S, S'
Operator	$O_i \in \mathcal{O}$
Parallelism degree	$d(O_i)$
Task of operator O_i	$T_i^x, x \in [k]$
Task T_i^x and operator O_i input streams	$\sigma_i^x \subseteq \sigma_i$
q^{th} Tuple in the stream σ_i	$e_q \in \sigma_i$
Processing latency of e_q on tasks T_i^x	$w_i^x(q)$
Average processing latency of operator O_i	$\overline{W}_i$
Completion time of e_q	$\ell(q)$
Average completion time of operator O_i	$\overline{L}_i$
Tuple e frequency	$f(e)$
Tuple e empirical probability of occurrence	$p(e)$
Size of the stream	m
Number of distinct tuples in the stream	n
2-universal hash function	h

Table 1: Notations.

3 Related Works

This section presents and discusses the most relevant strategies in the literature. Adaptation mechanisms aiming at maintaining processing within some performance goals are said to be *elastic* [32], *i.e.,* they adapt to input stream variance. Considering the huge difference between elastic mechanisms working at physical level (i.e. adapting resource consumption at infrastructure level) and those working at logical level (elastic mechanisms adapting workflows to fit processing load requirements), in this paper we only focus on the latter.

3.1 Elastic mechanisms at logical level

Workflows can be adapted to handle variations in input load. Logical level approaches can be classified as parallelism management, scheduling, and load-balancing.

Parallelism management — To process stream elements timely, operator output throughput should be greater than input throughput (taking into account also the selectivity factor). Nevertheless, with a fixed number of threads, two critical cases can occur:

- If input throughput is greater than output throughput for a sizable time period, then the number of elements in the buffering queue increases. This scenario causes an unacceptable increase in end-to-end latency and may lead to *congestion* [22, 33].
- If input throughput is smaller than output throughput for a sizable time period, then buffering queues are mostly empty and tasks are often idle. While in this scenario the system has low latencies, it also implies that resource usage is not maximized.

To handle these critical scenarios, SPEs should integrate a more refined parallelism management strategy. When facing an overload, SPEs should increase the parallelism degree (*scale-out*) of operators, thus decreasing the queuing time of incoming elements. Conversely, when input throughput is low, SPEs should decrease the parallelism degree (*scale-in*) to minimize resource waste.

Scheduling strategy — Given the operator parallelism degree, SPEs must schedule the tasks to the available processing units (Figure 1). We identify three classes of scheduling strategies:

- Strategies based on CPU load balancing between all processing units [1, 27, 41] assign threads on as many units as possible to divide processing load evenly. Using all available resources is an appropriate solution to limit processing bottlenecks due to CPU shortage. The problem is that it may imply massive network overheads [39] and underused units.
- Strategies based on network traffic reduction [3, 39] tend to concentrate as many threads as possible on the same processing units to minimize network traffic. These approaches improve throughput of SPEs [39] and reduce the number of active machines compared to the previous class. However, when input rate increases significantly, active machines tend to be overloaded more quickly and imply major reconfiguration compared to strategies spreading load evenly among all available units.
- Resource-aware strategies [3, 28] aim at avoiding processing unit overload and minimizing resource consumption. Through resource monitoring and processing requirements, this class of scheduling strategies allows threads to be grouped on processing units, thus minimizing resource waste. It offers efficient scheduling while having resource requirements for each thread to be assigned. The problem is that it requires accurate specifications about resource requirements and thus relies on user expertise. If user specifications are oversized or undersized, this leads to a waste or lack of resources, respectively.

Intra-operator load balancing — Operators can be classified as being either *stateful* (*e.g.,* standard deviation computation) or *stateless* (*e.g.,* filtering).

When the target operator is stateful, its state must be kept continuously synchronized among its instances, with potentially severe performance degradation at runtime; a well-known workaround to this problem consists in partitioning the operator state and letting each instance work on the subset of input stream containing the tuples affecting its state partition [22]. In this case, *key grouping* is the preferred choice as stream partitioning can be performed to correctly assign all the tuples containing specific data values and only those to the same operator instance, thus greatly simplifying the development of parallelizable stateful operators at the expense of performance.

In recent years there has been new interest in improving load balancing with key grouping [17, 26, 31]. It is worth noting that all works cited assume that all tuples of a stream have the same execution time.

Considering stateless operators, *i.e.,* data operators whose output is only a function of the current tuple in input, parallelization is straightforward. The grouping function is free to assign the next tuple in input stream to any available instance of the receiving operator (contrary to stateful operators, where tuple assignment is constrained). Such stream partitioning functions are often called *shuffle grouping* and represent a fundamental element of a large number of stream processing applications [22]. Notice that solutions for shuffle grouping techniques can be applied to stateful operators as well, provided that the operator implementation includes some mechanism to warranty state consistency (*e.g.,* a subsequent reduce phase). Given its generality, in this work we consider only shuffle grouping stream partitioning.

Typical implementations of shuffle groupings are based on round-robin scheduling [4, 5]. However, the processing latency of many operators are intrinsically sensitive to values. For example, an operator applying a transformation on each character of a text has a processing latency depending on the length of the text. Thus, high fluctuations in such values most likely increase load imbalance considerably, which lead to performance problems.

3.2 Triggering elastic stream processing

Solving operator congestion in a stream processing context is a complex problem. Out of the three major factors (parallelism management, scheduling, load balancing), to our knowledge, most works [3, 18, 33, 39] address only one at any time.

However, a clear distinction can be made between *reactive* approaches [21, 33, 40], which detect and remove potential problems from the current state of the system, and *proactive* approaches which predict potential problems and anticipate solutions [15, 34]. Among reactive solutions, we distinguish between *on user-demand* [40] and *automatic* [21, 33] solutions.

In [40], authors suggest a solution triggering *scale-in* and *scale-out* on user demand. This solution relies on the user adding enough resources when through-

put declines. Consequently, this solution is mainly limited by the need for user expertise and presence.

Dynamic and automatic approaches [18, 33] also aim at adapting parallelism degrees to avoid congestion of operators. They are based on global and local consumption thresholds (CPU, memory), which aim at separating a normal consumption from a critical one. In addition, in [21], authors suggest an algorithm integrating a knowledge base, built through a learning phase and updated at runtime. This knowledge base associates parallelism degrees with expected throughput for each operator. These solutions share the fact of using current resource consumption to detect potential congestion, thus making anticipation almost impossible. Furthermore, they pay no attention to data distribution within the input data streams.

Finally, some model-based solutions [15, 34] anticipate congestion, thanks to a complete model of the execution support and operator features (processing latencies, pending queues, *etc.*). Here, the parallelism degrees are adapted to minimize overall latency. Unfortunately, these solutions require detailed characteristics of the system and do not support any evolution of the execution support. In [38], authors suggest the Chronostream system, which is able to scale operators transparently and to manage internal states for both stateless and stateful operators. Even if this approach has demonstrated its efficiency in terms of scalability, Chronostream relies on stream partitioning to balance the load between operator instances. Thus, if there is a significant difference between the average processing latency for distinct keys, Chronostream is unable to compensate that imbalance accurately.

Summarizing SPE elasticity at logical level, the elasticity of a SPE mainly depends on choices related to parallelism management, scheduling, and load-balancing. Other aspects like workflow optimization [2, 6] and implementation selection [22] are user-provided and cannot be modified at runtime. In this context, we aim at suggesting a stream-based solution scaling treatments according to stream evolution in terms of input rate and value distribution.

4 Parallelism management with AUTOSCALE+

In [24] we defined a proactive approach, named AUTOSCALE, to manage dynamically and automatically the parallelism degree of operators using indicators monitored on streams and operators. Our proposed algorithm decides which operators have to be reconfigured (scale-out or scale-in) and what their new parallelism degrees are. These decisions are based on estimations of data stream evolution and resource consumption, which are computed from monitored indicators. The main originality of AUTOSCALE is that it considers the workflow as a whole, and more precisely the dependencies between operators, when validating a reconfiguration decision. It is worth noting that the algorithm we proposed offers satisfying results in deciding when a reconfiguration is required, but that the new parallelism degree computed was not always relevant, generating too frequent reconfiguration, thus leading to system instability in some specific cases.

We have investigated the reasons for such behaviors and identified two main causes. The first corresponds to variations in computation times from one item to another depending on their values. Clearly, variations in distribution of these values within the input streams also has an impact. The problem is that such variations of computation times jeopardize the default STORM load balancing method. To solve this problem, the only solution is to replace the load-balancing method with a new method that has to pay attention to such variations (see section 5). The second cause is related to the AUTOSCALE method itself. It transpired that the resource consumption analysis was not precise enough. In AUTOSCALE+, it has to be improved to better fit the specificities of the STORM's architecture. In this section, we first recall the general principles of AUTOSCALE before describing new methods embedded in AUTOSCALE+. This new proposal improves AUTOSCALE, taking CPU usage and user constraints into account.

4.1 AUTOSCALE+ Metrics

After presenting the general principles, we focus on the monitoring problem. Finally, we detail the new metrics embedded in AUTOSCALE+.

Principle
Just like its predecessor AUTOSCALE [24], AUTOSCALE+ anticipates potential congestion of operators through stream and operator monitoring[3]. For each operator, input volumes in the near future are estimated according to time series analysis [10]. The two algorithms also share the analysis of many dynamic properties like processing latency, pending queues, and the selectivity of each operator. Based on these, processing rates, or *capacity* can be estimated. The combination of these estimations make it possible to recommend scale-in, scale-out or nothing for each operator. Depending on available and configured resources dedicated to the SPE, a reconfiguration of threads running on the cluster could be triggered.

In contrast to AUTOSCALE, AUTOSCALE+ considers precise resource usage in terms of CPU, RAM and bandwidth. This allows improvement of decisions, for example avoiding reconfigurations when parallelism degree is not the root cause of problems, and more quickly reaching the adequate parallelism degree.

Monitoring management.
Monitoring management is based on sliding windows observing simultaneously all threads assigned on the execution support.

Let $\mathcal{F}$ be a set of monitoring sliding windows $\mathcal{F}_i = \{(F_j^i)\}_{j \in \mathbb{N}^+}$. Each window $\mathcal{F}_i$ is associated with an operator $\mathcal{O}_i$, and is composed of iterations F_j^i. Each iteration F_j^i is defined by a duration Δ and gathers measurements collected during this interval. These measurements are collected according to a predefined

[3]At each scale-in or scale-out, system monitoring is disabled while the system stabilizes. Indeed, the data acquired during this transition period do not provide any information about the nominal behavior of the new configuration of the system.

set of timestamps $\mathcal{M}_i = \{m^{i,1}, m^{i,2}, \ldots, m^{i,n}\}_{n \in \mathbb{N}^+}$. For each operator $\mathcal{O}_i$, our approach collects some measurements taking into account the stream elements received and processed on the interval $[m^{i,p-1}, m^{i,p}[$ with $p \in [n]$ where $[n] = \{1, \ldots, n\}$. More information about monitoring management is available in [24].

Let $\mathcal{R}^i$ be the set, potentially infinite, of stream elements received by operator $\mathcal{O}_i$. We consider $\mathcal{R}^{i,j}$ as the subset of elements received by $\mathcal{O}_i$ during the iteration F^i_j, and $\mathcal{R}^{i,p}$ the subset of elements received between $[m^{i,p-1}, m^{i,p}[$.

Let us now consider a parent operator $\mathcal{O}_{par}$ and a child operator $\mathcal{O}_{ch}$ consuming stream elements produced by $\mathcal{O}_{par}$. For both operators, we observe inputs $\mathcal{R}^{i,j}$. These inputs are inserted in pending queues where elements are consumed by associated functions. We define as $Input^{i,j}$ the sum of inputs processed currently by the function and stream elements pending in the queue during the iteration F^i_j. At the same time, we monitor the processing latency of the function and its selectivity factor for filter-based operators as presented in [23].

Metrics on operator input and output.
From these monitored values, we compute some metrics to analyze the activity of each operator. The aim is to identify operators which could have critical input volumes according to their processing capacities in the near future.

To do this, incoming volumes during the next iteration of the monitoring window are estimated, see Figure 2. This estimation, called $Estim\mathcal{R}^{i,j}$, is computed using a regression function f^i_{j-1} computed based on the previous iteration as follows:

$$Estim\mathcal{R}^{i,j} = \sum_{m^i_p \in \mathcal{M}_i} \lceil f^i_{j-1}(m^i_p) \rceil \tag{3}$$

where each m^i_k belongs to the next iteration of the window.

To estimate precisely f^i_{j-1}, AUTOSCALE+ selects the best candidate, i.e. the one best fitting to the previous iteration, among three competitors: linear, logarithmic, and exponential regression models. Compared to AUTOSCALE, the computation overhead is very small, while stream fluctuations are improved.

Operator history is not the only information that can be considered. Indeed, the near future of an operator is clearly influenced by antecedent operators. Furthermore, for an (antecedent) operator, a simple combination of the already computed $EstimInput^{i,j}$ with the average selectivity factor $\overline{\alpha}(F^i_{j-1})$ provides an estimation of its output $EstimOutput^{i,j}$:

$$EstimOutput^{i,j} = EstimInput_{F^i_j} \times \overline{\alpha}(F^i_{j-1}) \tag{4}$$

So, considering an operator who has ancestors, to estimate its inputs in a near future, we have two pieces of information. On the one hand, we have $EstimInput^{i,j}$ computed from its history, while, on the other hand, we have $EstimOutput^{i,j}$ the estimated outputs of its antecedent operators. A *combine* strategy is used to mix these elements. Last but not least, to approximate the total volume of stream elements each operator will have to process during the next iteration, attention must be paid to the pending queue. Stream elements

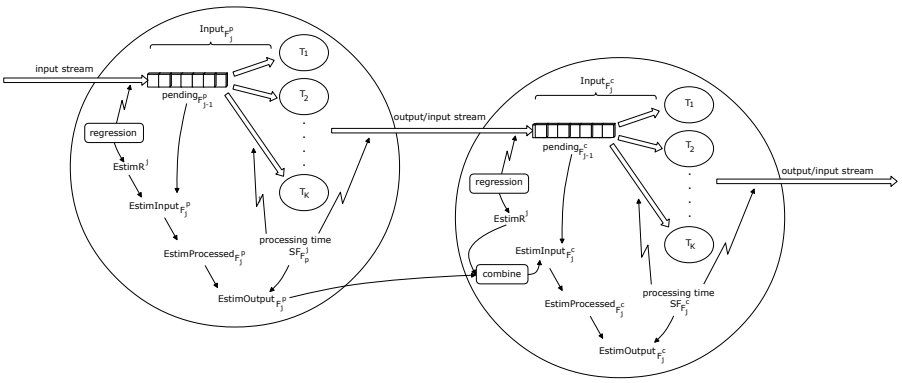


Fig. 2: Illustrating metrics considering two operators.

pending in the operator queue, noted $pending^{i,j-1}$, just have to be added to the estimation. Finally, $EstimInput^{i,j}$ is defined as:

$$EstimInput^{i,j} = combine\left(Estim\mathcal{R}^{i,j}, \sum_{p \in par(\mathcal{O}_i)} EstimOutput^{p,j}\right) + pending^{i,j-1} \quad (5)$$

where $par(\mathcal{O}_i)$ returns all parent operators of $\mathcal{O}_i$.

Many different *combine* functions can be proposed or obtained by learning techniques. By default, AUTOSCALE+ simply returns the *max* of the two values. This corresponds to a cautious strategy with respect to scale-in operation. Indeed, scale-in is analyzed with respect to the highest estimation. On the contrary, the *combine* strategy can return the *min* estimation to avoid over-consumption of resources due to an ephemeral increase in input rates. The strategy used will depend on the user's priorities.

Operator capacity estimation.
Intuitively, the capacity of an operator to treat items during a period Δ can be estimated considering the processing time of elements.

$$IdealCapacity^{i,j} = \frac{\Delta}{Lat^{i,j}} \quad (6)$$

where $Lat^{i,j}$ is the processing latency.

This approximation would be quite good if computational resources used by an operator were constant. Unfortunately, it is not so simple. For example, a task can take advantage of free CPU to make use of more CPU than reserved. To illustrate this point, let us consider an example of three operators $\mathcal{O}_A$, $\mathcal{O}_B$

and $\mathcal{O}_C$, executed, respectively, by threads T_A, T_B and T_C, running on a single CPU, C. As depicted on figure 3, some reservations have been made for each of them [28], let's say $ResaCPU_A$, $ResaCPU_B$ and $ResaCPU_C$. While the interest of this constraint is to avoid assignments leading to resource starvation, it should be kept in mind that a resource used by a task is not fully defined by the reservations made for it.

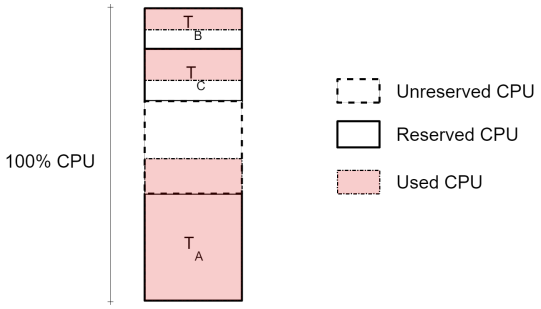


Fig. 3: Usable CPU for threads on one core

Considering figure 3, threads T_B and T_C use less CPU than they reserved. Clearly, there is no congestion since some CPU time is not used. Thread T_A takes advantage of the situation.

Here, the problem is to estimate usable CPU in a near future, and this has to be done for each operator/thread. We estimate the usable CPU by a thread T_X according to formula 7, where a *weighting factor* $\lambda \in [0;1[$ has been introduced to underestimate slightly available CPU and thus avoid fast overload.

$$UtilCPU(T_X, C) = \lambda \times max(UsedCPU(T_X, C), ResaCPU_X) + \frac{1}{n}(100 - \sum_{\forall T_Y \neq T_X} UsedCPU(T_Y, j)) \quad (7)$$

Considering an operator $\mathcal{O}_i$, to estimate the CPU it can use ($UtilCPU_i$), all its associated threads have to be considered (T_i^1, T_i^2,..., T_i^m). The global CPU time $UtilCPU_i$ for $\mathcal{O}_i$ is estimated as follows:

$$UtilCPU_i = \min_{x=1...x=m}(UtilCPU(T_i^x, CPU(T_i^x)) \quad (8)$$

Here, we assume that the input rate increase is spread evenly across all threads.

Capacity is then estimated:

$$Capacity^{i,j} = \frac{\Delta}{Lat^{i,j}} \times UtilCPU_i \quad (9)$$

4.2 Parallelism degree management in AUTOSCALE+

We now have enough information to detect imbalances between processing requirements and resource usage. Analysis of divergences will lead to one of the following three conclusions: need for scale-out, possibility of scale-in, or do nothing. If scale-out is often a need, scale-in is only a possibility. Indeed, these two operations are costly, and system stability is important to avoid wasting computing resources and time.

An "ideal" parellism degree.
Considering the estimations of the incoming workload and of the capacity, the ideal parallelism degree, leading the operator to efficient stream processing and denoted $idealK$, can be estimated according to:

$$idealK = \frac{EstimInput^{i,j}}{Capacity^{i,j}} \tag{10}$$

A working interval for stability issues.
Stability is a major issue for an automatic process and it is important to find a good balance. As scale-outs are needed for the system to work correctly, the focus is on scale-in operations. Intuitively, even if it is possible, the scale-in operation will be retained until a large benefit is attained. To encode this intuition, we introduce a "working interval": this defines a zone where, even if the estimated $idealK$ is smaller than the actual parallelism degree, no reconfiguration will be carried out due to a lack of benefits. Furthermore, we suggest a controllable interval: its size should vary depending on the parallelism degree, and it should be controllable with respect to different considerations (user preferences, evolution -e.g. reducing- over time and so on). Trivially, to define an interval, two bounds have to be defined. The upper bound will be the current parallelism degree, while the lower bound $\min_k$ will be a function of the current parallelism degree k:

$$min_k = \beta \times k \tag{11}$$

where $\beta \in]0;1]$ is a controllable parameter. If β is close to 0, it means that AUTOSCALE+ performs scale-in only when input volumes are very small compared to operator capacities. If β is close to 1, AUTOSCALE+ performs scale-in as soon as possible.

It is worth noting that the greater k, the greater the associated working interval. We choose this property because the more tasks there are to merge, the more time it takes to merge pending queues distributed over the cluster and to re-route stream elements.

Decision of parallelism degree modification
A scale-out should be performed for an operator $\mathcal{O}_i$ when $idealK$ exceeds the working interval as represented on Figure 4.

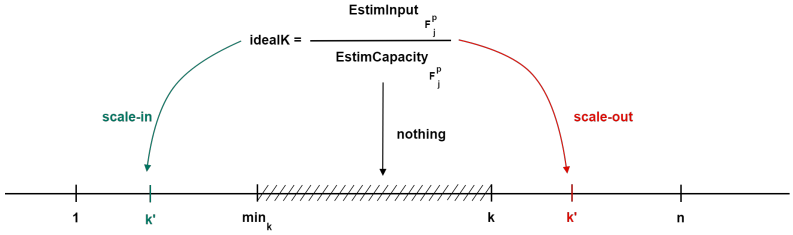


Fig. 4: Modification of parallelism degree

On the other hand, if $idealK$ is smaller than min_k, a scale-in will be performed.

Otherwise, if $idealK$ remains within the interval, even if a scale-in is possible, the operator is not modified and retains the current parallelism degree.

Computation of the appropriate parallelism degree

Thus, each time a change is decided, whether it is a scale-in or a scale-out, AUTOSCALE+ computes a new appropriate parallelism degree k' according to estimated needs and computing resources possibilities:

$$argmin_{k'}(\frac{EstimInput^{i,j}}{ResCapacity^{i,j}} \leq k) \tag{12}$$

where particular attention must be paid to denominator. Indeed, $ResCapacity^{i,j}$ deals with resources, but compared to $idealK$ there is a major difference: reservations $ResCPU_i$ are used instead of resource estimations. Indeed, such a change may lead to reallocations of threads and the only guarantee we have is the amount of resource required by the reservation. This leads to the formula:

$$ResCapacity^{i,j} = \frac{\Delta}{Lat^{i,j}} \times ResCPU_i \times \lambda \tag{13}$$

where $\alpha \in]0;1]$ is a parameter allowing AUTOSCALE+ to consider a relative margin between effective CPU usage and CPU reservation. This means that AUTOSCALE+ takes into account the fact that some threads may need more than their reservation at runtime. Just like β, the parameter λ can be defined using several methods like empirical study, reinforcement learning or user expertise.

5 Load balancing with OSG

While an adequate parallelism degree is important, alone it does not fully solve the problem. Indeed, changing the parallelism degree is not enough to address any variations in value distribution when facing a significant heterogeneity in

terms of computational resource needs. Failure to pay attention to load balancing, as round-robin scheduling would do, may lead to imbalance problems (see Figure 5)

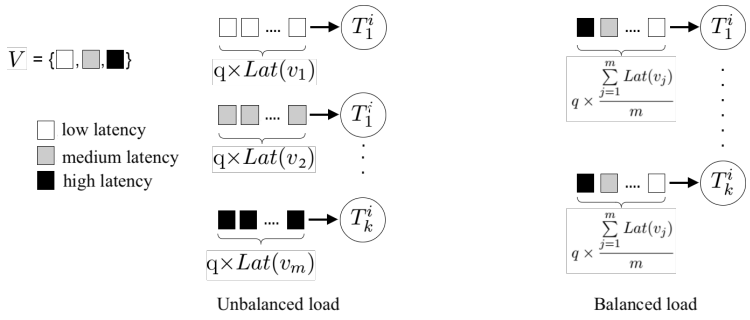


Fig. 5: Worst and optimal cases of load balancing

Furthermore, imbalance problems could jeopardize any method trying to manage the parallelism degree, as AUTOSCALE+. A careful load balancing strategy, compatible with our philosophy and proposed solutions, is definitely needed. First, it has to be proactive. Prevention is better than cure since it avoids misleading the parallelism degree strategy. More generally, its behavior should not interfere with AUTOSCALE+ and both have to work together as smoothly as possible. Second, significant efforts have been made in AUTOSCALE+ to limit any dependence on the user. So as not to render these useless, the load balancing strategy should not require any user intervention.

This section is devoted to OSG, a load balancing strategy which has been specifically designed to deal with significant variations in computational resource needs depending on stream item values. We present its principles and main solutions. For more information, proofs and specific experimental evaluations, readers should refer to [29–31]. Indeed, this paper is focussed on the evaluation of DABS-STORM to study how AUTOSCALE+ and OSG interact and react to data stream fluctuations.

OSG is a shuffle grouping implementation based on a simple, yet effective idea: if we assume to know of the execution time $w_i^x(t)$ of each tuple t the parallel tasks of a given operator O_i, we can schedule the execution of incoming elements on such tasks with the aim of minimizing the average per tuple completion time of the tasks. However, the value of $w_i^x(t)$ is generally unknown. A common solution to this problem is to build a cost model for the execution time and then use it to pro-actively schedule the incoming load. However, building an accurate cost model usually requires a large amount of *a priori* knowledge on the system. Furthermore, once a model has been built, it can be hard to handle changes in the system or input stream characteristics at runtime.

To overcome all these issues, OSG takes decisions based on the estimation $\widehat{\mathcal{C}}_i^x$ of the execution time assigned to task T_i^x of operator O_i, that is $\mathcal{C}_i^x = \sum_{t \in \sigma_i^x} w_i^x(t)$. In order to do so, OSG computes an estimation $\widehat{w}_i^x(t)$ of the execution time $w_i^x(t)$ of each tuple t on task T_i^x of operator O_i. Then, OSG can also compute the sum of the estimated execution times of the tuples assigned to task T_i^x, *i.e.,* $\widehat{\mathcal{C}}_i^x = \sum_{t \in \sigma_i^x} \widehat{w}_i^x(t)$, which in turn is the estimation of $\mathcal{C}_i^x$. A greedy scheduling algorithm (Section 5.1) is then fed with estimations for all the available operator tasks.

To implement this approach, each operator task builds a sketch (*i.e.,* a memory efficient data structure) that will track the execution time of the tuples it processes. When a change in the stream or task(s) characteristics affects the tuple execution times on some tasks, the concerned task(s) will forward an updated sketch to the scheduler that will then be able to (again) correctly estimate the tuples execution times. This solution does not require any *a priori* knowledge of the stream composition or the system, and is designed to continuously adapt to changes in the input distribution or the tasks load characteristics. Moreover, this solution is *proactive*, namely its goal is to avoid imbalance through scheduling, rather than detecting the unbalance and then attempting to correct it. A *reactive* solution can hardly be applied to this problem, as it would schedule input tuple on the basis of a previous, possibly stale, load state of the operator tasks. Furthermore, reactive scheduling typically imposes a periodic overhead even if the load distribution imposed by input tuples does not change over time.

For clarity's sake, we consider a topology with two operators: a non parallelized operator O_{sched} (*i.e.,* a *scheduler*), which schedules the tuples of a stream σ_{op}, and an operator O_{op}, whose k instances consume the stream σ_{op} (see Figure 6). To encompass topologies where the operator generating the stream σ_{op} is itself parallelized, we can easily extend the model by taking into account parallel tasks of the operator O_{sched}. More precisely, there are s tasks/schedulers $T_{\text{sched}}^1, \ldots, T_{\text{sched}}^s$, where task/scheduler T_{sched}^x schedules only a subset of σ_{op}, *i.e.,* its own output. In [31] we also show that in this setting OSG performances are better than the Round-Robin scheduling policy. In other words, OSG can be deployed when the operator O_{sched} is parallelized. Notice that our approach is hop-by-hop, *i.e.,* we consider a single shuffle grouped edge in the topology at a time. However, OSG can be applied to any shuffle grouped stage of the topology.

5.1 `Count Min` sketch algorithm

In [14], Cormode and Muthukrishnan introduced the `Count Min` sketch that provides, for each item e in the input stream, an (ε, δ)-additive-approximation $\hat{f}(e)$ of the frequency $f(e)$.

An algorithm is said to be an (ε, δ)-additive-approximation of the function ϕ on a stream σ if, for any prefix of size m of items of the input stream σ, the algorithm output $\hat{\phi}$ is such that $\mathbb{P}\{\mid \hat{\phi} - \phi \mid > \varepsilon C\} < \delta$, where ε, $\delta > 0$ are given as precision parameters and C is an arbitrary constant. The parameter ε represents the precision of the approximation estimation. For instance $\varepsilon = 0.1$

means that the additive error is less than 10% and $\delta = 0.01$ means that this approximation will not be satisfied with a probability less than 1%.

The `Count Min` sketch consists of a two dimensional matrix Φ of size $r \times c$, where $r = \lceil \log(1/\delta) \rceil$ and $c = \lceil 2.7/\varepsilon \rceil$. Each row is associated with a different 2-universal hash function $h_i : [n] \rightarrow [c]$.

A collection $\mathcal{H}$ of hash functions $h : [n] \rightarrow [c]$ is said to be 2-universal if for every two different items $x, y \in [n]$, for all $h \in \mathcal{H}$, $\mathbb{P}\{h(x) = h(y)\} \leq 1/c$, which is the probability of collision obtained if the hash function assigned truly random values in $[c]$. Carter and Wegman [11] provide an efficient method for building large families of hash functions approximating the 2-universality property.

When the `Count Min` algorithm reads item e from the input stream, it updates each row: $\forall i \in [r], \Phi[i, h_i(e)] \leftarrow \Phi[i, h_i(e)] + 1$. Thus, the cell value is the sum of the frequencies of all the items mapped to that cell. Upon request of f_e estimation, the algorithm returns the smallest cell value among the cells associated with t: $\hat{f}_e = \min_{i \in [r]}\{\Phi[i, h_i(e)]\}$.

Fed with a stream of m items, the space complexity of this algorithm is $\mathcal{O}(\log[(\log m + \log n)/\delta]/\varepsilon)$ bits, while update and query time complexities are $\mathcal{O}(\log(1/\delta))$. The `Count Min` algorithm guarantees that the following bound holds on the estimation accuracy for each item read from the input stream: $\mathbb{P}\{\mid \hat{f}(e) - f(e) \mid \geq \varepsilon(m - f_e)\} \leq \delta$, while $f(e) \leq \hat{f}(e)$ is always true.

This algorithm can be easily generalized to provide (ε, δ)-additive-approximation of point queries $\mathcal{Q}_e$ on a stream of udpates, *i.e.,* a stream where each item e carries a positive integer update value v_e. When the `Count Min` algorithm reads the pair $\langle e, v_e \rangle$ from the input stream, the update routine changes as follows: $\forall i \in [r], \Omega_[i, h_i(e)] \leftarrow \Omega_[i, h_i(e)] + v_e$.

Greedy Online Scheduler A classical problem in the load balancing literature is to schedule independent tasks on identical machines minimizing the makespan, *i.e.,* the *Multiprocessor Scheduling* problem. In this paper, we adapt this problem to our setting, *i.e.,* to schedule *online* independent tuples on non-uniform operator instances in order to minimize the average per tuple completion time $\overline{L}$. Online scheduling means that the scheduler does not know in advance the sequence of tasks it has to schedule. The Greedy Online Scheduler algorithm assigns the currently submitted tuples to the less loaded available operator instance. In [31] we show that this algorithm is a $(2 - \frac{1}{k})$-approximation of an optimal omniscient scheduling algorithm, namely an algorithm that knows in advance all the tuples it will receive. Notice that this is a variant of the join-shortest-queue (JSQ) policy [25], where we measure queue length as the time needed to execute all the buffered tuples, instead of the number of buffered tuples.

5.2 Online Shuffle Grouping design

Each operator O_{op} task instance T^x_{op} maintains two `Count Min` sketch matrices (Figure 6.A): the first, denoted by Φ^x_{op}, tracks the tuple frequencies $f_{t,\text{op}}$; the

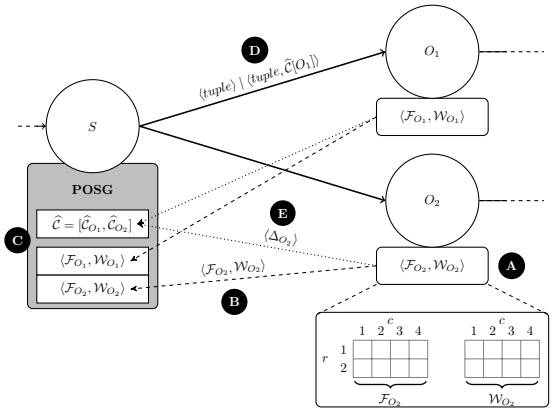


Fig. 6: OSG design where $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$) and $k = 2$.

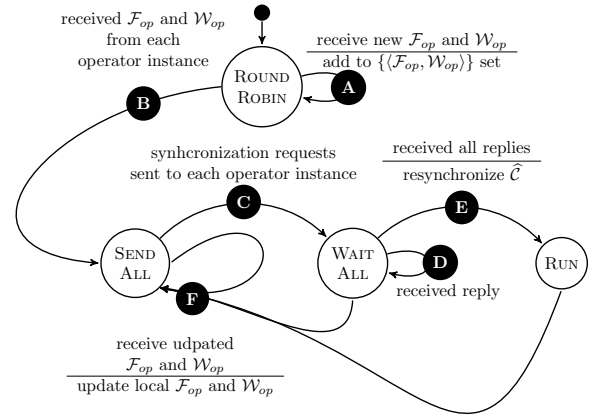
second, denoted by Ω^x_{op}, tracks tuples cumulated execution times $\Omega^x_{\text{op}} = w^x_{\text{op}}(t) \times f^x_{\text{op}}(t)$. Both `Count Min` matrices have the same sizes and hash functions. The latter is the generalized version of the `Count Min` presented in Section 5.1, where the update value is the tuple execution time when processed by the instance (*i.e.,* $v_t = w^x_{\text{op}}(t)$). The operator instance will update both matrices after each tuple execution.

The operator tasks are modeled as a finite state machine (Figure 7b) with two states: START and STABILIZING. The START state lasts until the task has executed N tuples, where N is a user-defined window size parameter. The transition to the STABILIZING state (Figure 7b.Ⓐ) triggers the creation of a new snapshot Ψ^x_{op}. A snapshot is a matrix of size $r \times c$ where $\forall i \in [r], j \in [c]$: $\Psi^x_{\text{op}}[i,j] = \Omega^x_{\text{op}}[i,j] / \Phi^x_{\text{op}}[i,j]$. We say that the Φ^x_{op} and Ω^x_{op} matrices are stable when the relative error η^x_{op} between the previous snapshot and the current one is smaller than μ, that is if

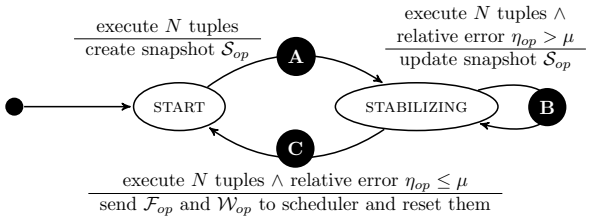
$$\eta^x_{\text{op}} = \frac{\sum_{i=1}^{r} \sum_{j=1}^{c} \left| \Psi^x_{\text{op}}[i,j] - \frac{\Omega^x_{\text{op}}[i,j]}{\Phi^x_{\text{op}}[i,j]} \right|}{\sum_{i=1}^{r} \sum_{j=1}^{c} \Psi^x_{\text{op}}[i,j]} \leq \mu \tag{14}$$

is satisfied. Then, each time task T^x_{op} has executed N tuples, it checks whether Equation 14 is satisfied. **(i)** If not, then Ψ^x_{op} is updated (Figure 7b Ⓑ). **(ii)** Otherwise the task sends the Φ^x_{op} and Ω^x_{op} matrices to the scheduler (Figure 6 Ⓑ), resets them and moves back to the START state (Figure 7b Ⓒ).

There is a delay between any change in the stream or operator task characteristics and when the time the scheduler receives the updated Φ^x_{op} and Ω^x_{op} matrices from the affected operator tasks(s). This introduces a skew in the cumulated execution times estimated by the scheduler. To compensate for this skew, we introduce a synchronization mechanism that springs whenever the scheduler receives a new pair of matrices from any task. Notice also that there is an initial transient phase in which the scheduler has not yet received any information from



(a) Scheduler.



(b) Operator task.

Fig. 7: OSG finite state machines.

operator instances. This means that, in this first phase, it has no information on the tuple execution times and is forced to use the Round-Robin policy. This mechanism is thus also needed to initialize the estimated cumulated execution times when the Round-Robin phase ends.

The scheduler maintains the estimated cumulated execution time for each task, in a vector $\widehat{\mathcal{C}}$ of size k, and the set of pairs of matrices: $\{\langle \Phi^x_{\text{op}}, \Omega^x_{\text{op}} \rangle\}$, initially empty.

The scheduler is modeled as a finite state machine with four states: ROUND-ROBIN, SEND ALL, WAIT ALL, and RUN.

The ROUND-ROBIN state is the initial state in which scheduling is performed with the Round-Robin policy. In this state, the scheduler collects the Φ^x_{op} and Ω^x_{op} matrices sent by the operator tasks (Figure 7a Ⓐ). After receiving the two matrices from each instance (Figure 7a Ⓑ), the scheduler is able to estimate the execution time for each submitted tuple and moves to the SEND ALL state. When in the SEND ALL state, the scheduler sends the synchronization requests towards to the k tasks. To reduce overhead, requests are piggy backed (Figure 6 Ⓓ) with outgoing tuples applying the Round-Robin policy for the next k tuples: the i-th tuple is assigned to operator instance $i \mod k$. On the other hand, the estimated

cumulated execution time vector $\widehat{\mathcal{C}}$ is updated with the tuple estimated execution time.When all the requests have been sent (Figure 7a Ⓒ), the scheduler moves to the WAIT ALL state. This state collects the synchronization replies from the operator tasks (Figure 7a Ⓓ). Operator task T^x_{op} reply (Figure 6 Ⓔ) contains the difference Δ^x_{op} between the instance cumulated execution time $\mathcal{C}^x_{\text{op}}$ and the scheduler estimation $\widehat{\mathcal{C}}[op]$.

In the WAIT ALL state, scheduling is performed as in the RUN state. When all the replies for the current epoch have been collected, synchronization is performed and the scheduler moves to the RUN state (Figure 7a Ⓔ). In the RUN state, the scheduler assigns the input tuple applying the Greedy Online Scheduler algorithm, *i.e.,* assigns the tuple to the task with the least estimated cumulated execution time. Then it increments the target instance estimated cumulated execution time with the estimated tuple execution time. Finally, in any state except ROUND ROBIN, receiving an updated pair of matrices Φ^x_{op} and Ω^x_{op} moves the scheduler to the SEND ALL state (Figure 7a Ⓕ).

Readers can refer to [29] for the complete theoretical analysis of OSG, in terms of correctness, accuracy and complexities.

6 DABS-Storm

We now have two methods, AUTOSCALE+, which adapts the parallelism degree of each operator, and OSG, which carefully balance streams' items between tasks of an operator. Integration seems quite easy. Nevertheless, mixing methods can always raise compatibility issues.

An auto-parallelization approach like AUTOSCALE+ assumes that congestion is due to an input overload of all tasks associated with an operator. In this case, adding more tasks is indeed the recommended solution. The better the load balance, the greater the scale-out effects will be. Furthermore, considering the definition of $UtilCPU$ (see equation 8), the assumption of a good load balancing is present. Consequently, with a careful proactive load balancing preventing load imbalance, OSG is expected to improve both the decision process and the effects of AUTOSCALE+. OSG is not a random choice. Indeed, the proactive aspect is here of major importance. Although one might think that a reactive solution could work as well, sometimes (too often) a non-prevented load balancing problem could lead to unnecessary scale-outs of an unpredictable magnitude.

On the other hand, to guarantee good performance, OSG needs a non-congested environment. This means that scale-outs have to be performed before any congestion occurs. It also means that scale-in has to be performed cautiously, not too soon, to avoid any risk of congestion due to a reversal of the trend within data streams. AUTOSCALE is also a proactive method, and is designed to perform a scale-out before congestion occurs. For scale-in, things are less evident. First, as explained in Section 4.1, page 12, users can choose between two strategies (related to the choice of the *combine* function): a cautious one, or a resource-oriented one aimed at avoiding over-consumption of resources. Second, the size of the *working interval*, and more particularly the parameter β (see section4.2,

page 14), is of major importance here. Indeed, it controls the margin of security thickness before performing a scale-in. To expect AUTOSCALE and OSG to work well together, the *combine* function must be a cautious one (i.e. a *max* function, which is the default choice), and the parameter β should be chosen close to zero to perform scale-in only when input volumes are very small compared to operator capacities.

To summarize, to take advantage of the benefits of both methods, it is not enough to use both of them. We must also ensure they are compatible. This seems to be the case here, provided that two AUTOSCALE+ parameters are correctly selected. However, an experimental study is essential to confirm this hypothesis.

7 Experiments

The implementations of AUTOSCALE+ and OSG which we use in experimental evaluations, have been developed to be integrated with Apache STORM [5]. The principles presented in AUTOSCALE+ and OSG could be integrated with many other DSMS, like for example Apache Spark Streaming [41], Flink [4], etc. However, for evident reasons of time and resources, at the beginning of the project we had to make a choice. We settled on STORM mainly for three reasons. First, in the STORM paradigm, stream elements must not be represented as key/value pairs, necessary for MapReduce-based approaches [16]. Second, it offers great flexibility for operator definition. Third, STORM serves as a guarantee that every item will be tracked and processed until an operator discards it (*e.g.* a filter or a final operator). Finally, it allows manual reconfiguration of parallelism degrees at runtime.

Thus, this section starts with some general reminders about this DSMS. Then we detail the experimental setups. Finally, we present and comment on the obtained results.

7.1 Overview of Apache Storm

Apache STORM [5] is an open source DSMS allowing users to express their continuous queries through a declarative language or to directly build their topologies using a high-level language (Java, Python, Clojure, etc.).

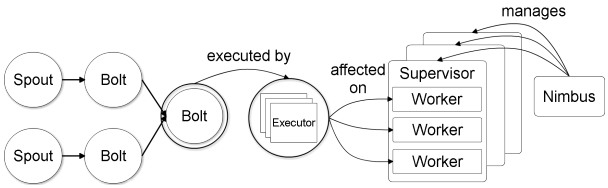


Fig. 8: Storm architecture

Whatever the language used, an operator, named a *component* in Storm's terminology, belongs to one of the two categories: *spouts* or *bolts*. A *spout* is a connector to a stream source, and thus can be used as an entry of a topology. It distributes stream elements to components to which it is connected and can process filtering operations if required. A *bolt* consumes items from any component and computes a result for each element received (*stateless* bolt) or for a set of elements (*stateful* bolt). Each *component* is executed in parallel by *executors*. Each *executor* is assigned to a processing unit by the scheduler (see Figure 8).

7.2 Experimental protocol

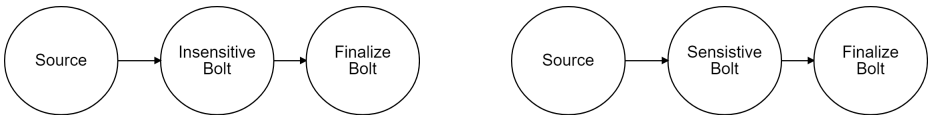
Experimental setup We experiment with the version 1.0.2 of Apache Storm. Our test cluster is composed of 10 VMs each with a dual-core CPU Intel(R) Xeon(R) E5-2620 running at 2.00GHz, 4GB of RAM and 40GB of hard disk space. A master VM, called *Nimbus*, is responsible for coordinating the 9 others dedicated to task execution. Each of these VMs, called a *Supervisor*, manage 4 processing units, called *workers*. Our module, managing the operator parallelism degree, implements the *IScheduler* interface of the Storm API. The module, managing distribution of stream elements between executors, implements the *CustomStreamGrouping* interface of the Storm API. We also deploy a MySQL database on Nimbus to store collected measurements. We summarize the main experimental parameters in Table 2:

Table 2: Main parameters

Component	Description	Symbol	Value
Storm	monitoring frequency		10s
	processing timeout		30s
Autoscale+	weighting factor	λ	0.3
	scale-in control	β	0.8
	combine strategy		max
OSG	precision	ϵ	0.05
	maximal probability of error	δ	0.05

Test topologies To validate our approach, we demonstrate its effectiveness on three topologies.

The simple insensitive topology (see figure 9a) composed of a spout (Source) emitting stream elements without filtering them. These stream elements are processed by a bolt (InsensitiveBolt) applying a function with a time complexity independent from the value read in input. Finally, a bolt (FinalizeBolt) ends the computation of each stream element by sending a termination signal to the Storm monitor.



(a) Simple insensitive topology. (b) Simple sensitive topology.

Fig. 9: Simple topologies.

The simple sensitive topology (see figure 9b) has the same structure as the simple insensitive topology. However, the function applied by the intermediate bolt (SensitiveBolt) has a time complexity that depends directly on the value read in the input.

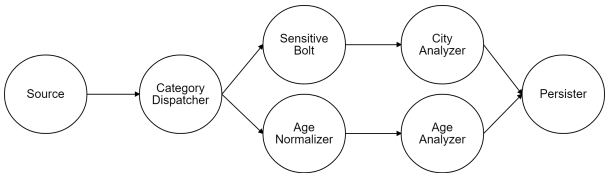


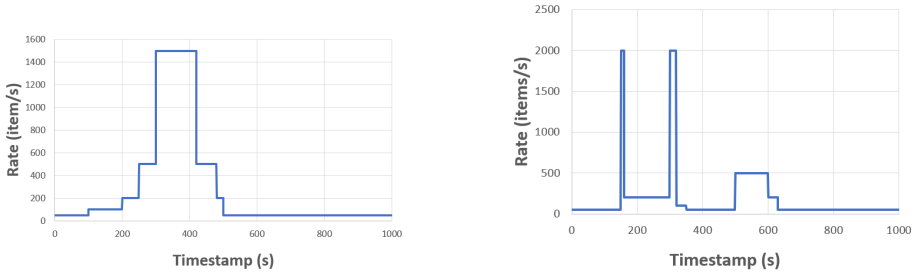
Fig. 10: Complex sensitive topology

The complex sensitive topology is inspired from real benchmark applications for Storm [4]. It is composed of several operators with various selectivity factors and average processing latencies. The spout (OpinionSource) emits stream elements concerning opinions submitted by users about a topic. Each opinion is described by information on the user, like her age and code representing her location, the topic and the user opinion. Stream elements are sent to a bolt (CategoryDispatcher) filtering unnecessary attributes and depending on the branch downstream. Moreover, it filters stream elements concerning a predefined list of irrelevant topics. A branch starts with a bolt (SensitiveBolt) retrieving information on user location from the code. This bolt has exactly the same properties as the sensitive bolt of the simple sensitive topology. Indeed, the time required to retrieve information on the city varies according to the code. It allows us to compare the impact of workflow structure and complexity on bolt behavior and dynamic adaptation of its parallelism degree. Then, a bolt (CityAnalyzer) extracts relevant subgroups according to opinion and location. The other branch, starting from the bolt CategoryDispatcher, performs similar treatments in order to define subgroups based on user opinion and age. Finally, the Persister takes as its input descriptions of subgroups and persists them in a storage file system.

It is important to bear in mind a major difference between this experimental setup and previous one and the consequences. The critical operator is not di-

[4] `https://github.com/yahoo/streaming-benchmarks`

rectly connected to the source. Stream elements are filtered and transformed by upstream operators. Consequently, its input rate may significantly differ from the source one, and strategies considering the workflow at a global scope will have a way to differentiate from those working only on local considerations. As a consequence, auto-parallelization strategies considering exclusively local observations cannot take advantage of fluctuations of input rate happening upstream. At the opposite, auto-parallelization strategies adapting parallelism degree of operators according to the global state of the workflow will have a natural advantage. So, this experimental setup highlights the fundamental difference between AUTOSCALE+ and other parallelization strategies considered in these experiments.



(a) Progressive stream.

(b) Erratic stream.

Fig. 11: Input streams.

Test data streams As illustrated in Figure 11, we build two synthetic streams with the following common features: 1) at least one critical increase in input rate leading the system to congestion with a minimal (one executor per operator) and static configuration 2) decrease in input rate to evaluate system elasticity. For each stream, we can set the distribution law, which may be be uniform over all possible values or biased according to a zipf law with a predefined skew. These streams allow us to determine which impact of DABS-STORM when facing critical fluctuations in both input rate and value distribution. The first corresponds to a large increase with a plateau before a decrease. The second is much more sudden, with far severer variations, highlighting system elasticity.

Baseline methods First, we have to deplore a lack of open source implementation of auto-parallelization strategies. We compare DABS-STORM to two methods.

The first is simply the native static behavior of Apache STORM an incremental strategy (noted *incremental* hereafter) considering only thresholds on CPU usage.

The second [18], is a reinforcement learning-based strategy mapping input rates to appropriate parallelism degrees at runtime (noted *Rlearning* hereafter).

For the experiments, the methods take advantage of a knowledge base base acquired through a training phase carried out using the test data streams. Then the knowledge base covers all the fluctuations encountered (which is not always the case in practice). More generally, this can be considered to be a favorable conditions.

7.3 Experiments and results

Not all the experiments conducted are presented here. More information can be found on our companion website [5]. In addition, this website includes a comparison between AUTOSCALE and AUTOSCALE+. It gives an overview of the gap between the performance of the auto-parallelization strategy presented in [24] and results presented below. In the remainder of this section, the results described correspond to average values over 5 iterations for each configuration, thus lessening the the impact of punctual anomalies during tests.

Variations in the input stream rate over an insensitive topology In this experiment, we confront the simple insensitive topology 9a with a stream with large, but not too erratic, variations in input rate, see Figure 11a.

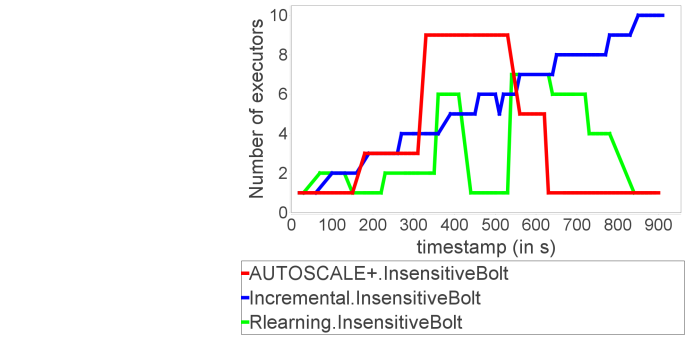
In this configuration, the volume of stream elements to process is the only impact factor and OSG is of little use. For the sake of equity, we choose to conduct an experiment where all compared solutions sharing the default grouping solution of STORM were denoted shuffle grouping. This means that here we only test the AUTOSCALE+ component of DABS-STORM. Note that other experimental evaluations show that, in this case, OSG behaves quite similarly to STORM's shuffle grouping

As expected, the *reinforcement learning* strategy increases the parallelism degree of the observed operator *InsensitiveBolt* (Figure 12a), before decreasing it just following the input rate. Nevertheless, these modifications have a major impact on average processing latency (Figure12b) and result quality (Figure 12c). Indeed, the system has been reconfigured (scale-in) with respect to the input rate and without considering the pending queues (which were far from empty).

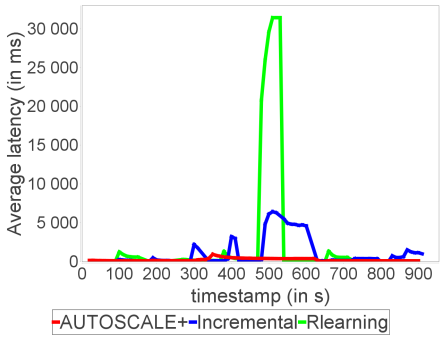
In comparison, the *incremental* strategy continuously increases the parallelism degree of the operator as long as the workload exceeds the processing rate (see figure 12). Even if the parallelism degree is increased, it cannot reach a suitable value as quickly as it needs to. This results in a large increase in average processing latency, causing a 29% loss of stream elements (dephased tuples) over the complete execution (Figure12c)). Moreover, in terms of resource usage, due to frequent system reconfigurations, the *incremental strategy* requires 64% more active processing units than the *reinforcement learning* strategy and 18% more than AUTOSCALE.

With AUTOSCALE+, Storm is able to anticipate suitable parallelism degrees over the complete execution. Even if AUTOSCALE+ tends to overestimate the

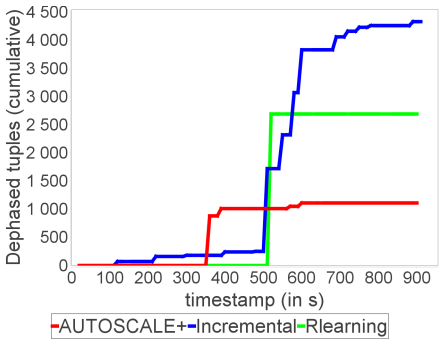
[5] `https://dabs.liris.cnrs.fr`



(a) Observed operator parallelism degree.



(b) Average global processing latency.



(c) Cumulated dephased tuples.

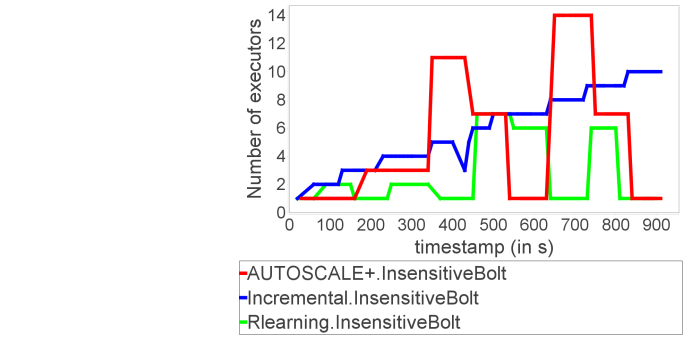
Fig. 12: Simple insensitive topology with progressive stream.

required parallelism degree due to regression, processing latency is much better (Figure 12b). It should also be noted that average processing latency remains remarkably stable, reducing losses to 7%.

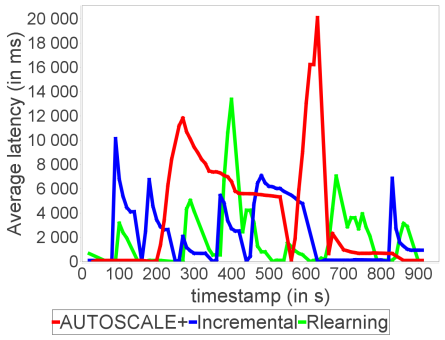
We can conclude that AUTOSCALE+, thus DABS-STORM, outperform the baseline methods when confronted with data streams with large input rate variations, even if workflow is unresponsive to data values.

***Erratic* variations in the input stream rate over an insensitive topology** Compared to the previous experiment, the variations in the input stream rate will be more erratic, confronting the same insensitive topology 9a with the second data stream, see Figure 11b.

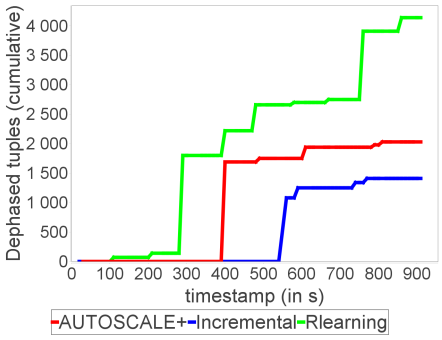
As illustrated in Figure 13a, the *reinforcement learning* strategy increases and decreases the parallelism degree of the observed operator *InsensitiveBolt* according to the two main peaks. However, the magnitude of the scale-out is not very high. Indeed, brief increases in input rate do not increase significantly the average input rate in recent history. Nevertheless, the, the sudden accumulation of a huge number of stream elements on pending queues increases the average processing latency. Luckily, the impact on result quality remains negligible with



(a) Observed operator parallelism degree.



(b) Average global processing latency.



(c) Cumulated dephased tuples.

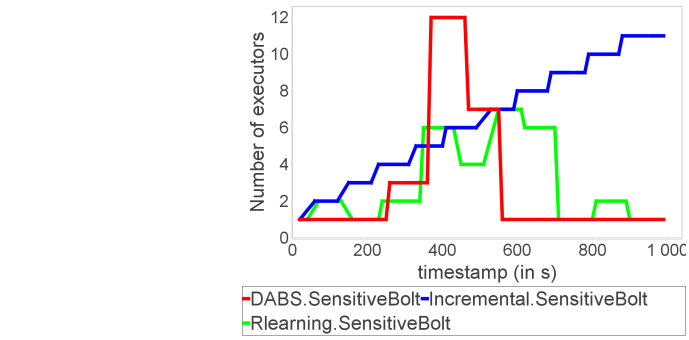
Fig. 13: Simple insensitive topology with erratic stream

only 13% of stream elements lost over the complete execution as shown on figure 13c.

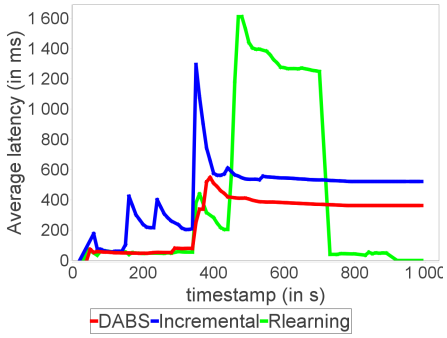
As the *incremental* strategy over-provisions the operator, available resources can hopefully handle brief increases in input rate. Consequently, the average processing latency (see Figure 13b) increases significantly only when the input rate remains high over a long period of time such as for the last increase in the erratic stream. While stream element losses (Figure 13c) are reduced to 19% over the complete execution, the usage of processing units remains higher than for the reinforcement learning strategy.

Considering Figure 13a, AUTOSCALE+ reacts faster to sudden input rates increasing the parallelism degree. However, the increase is too high with respect to the ephemeral nature of the phenomenon. In other words, AUTOSCALE+ overestimates processing requirements. This overestimation induces excessive reconfiguration overheads, affecting punctually the average processing latency (Figure 13b). Although results are delivered, 18% of the entire stream cannot be processed under the maximal threshold. As critical increase and decrease in input rate are sudden and brief, they cannot be anticipated and affect processing latency before AUTOSCALE+ reconfigures the system.

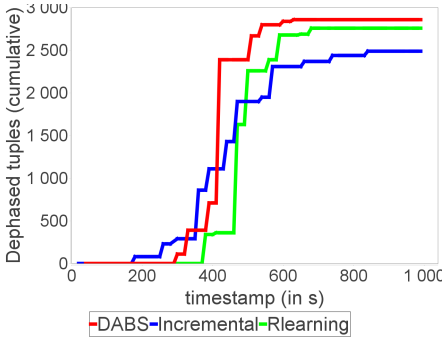
This experimental setup points out the limit of the predictive approach when input streams vary suddenly in volume. Indeed, while a progressive evolution of the input rate can be easily anticipated, sudden peaks in input rate induce inappropriate behavior of the system behavior. Several solutions can be considered such as reinforcement learning that can help reduce this effect.



(a) Observed operator parallelism degree.



(b) Average global processing latency.



(c) Cumulated dephased tuples.

Fig. 14: Simple sensitive topology with progressive stream

Variations in input stream rate and data distribution over a sensitive topology We now include variations in data distribution in the picture. The workflow and the data stream are of the same form as in experimentation 7.3 but with two major differences: first, in the input stream (Figure 11a) the value distribution is biased, following a zipf distribution with a skew of 1.5[6]; and second, the workflow (Figure 9b)) is sensitive to data values.

For the sake of equity, in this experiment, each parallelism degree strategy is combined with OSG to benefit from a better load balancing.

[6]This choice is motivated by previous results on OSG detailed in [31].

Here, AUTOSCALE+ anticipates processing requirements (Figure 14a) and is able to maintain a smaller processing latency (Figure 14b), while the stream is at its maximal rate.

The *reinforcement learning* strategy is able to reduce processing latency significantly (Figure 14b) when the input rate decreases.

With a parallelism degree evolution (Figure 14a) very closely approaching that observed in experiment7.3, the throughput of the *incremental* strategy is clearly less good. Having looked into this matter, this phenomenon is due to some kind of incompatibility between the incremental strategy and OSG. Indeed, step-by-step strategy trivially imposes frequent modifications of parallelism degree. At each step, OSG has to reevaluate its routing policy to keep a balance between executors.

In terms of tuple loss, all solutions deliver a similar performance even if the reinforcement learning strategy is able to keep losses under the incremental strategy and AUTOSCALE+. This is due to an overestimation of parallelism performed by AUTOSCALE+, affecting overall quality.

Concerning throughput, all solutions deliver a similar performance even if AUTOSCALE+ remains the auto-parallelization strategy keeping the smallest time shift between fluctuation in input rate and throughput.

Erratic variations in input stream rate and data distribution over a sensitive topology Considering DABS-STORM, the average processing latency (Figure 15b) remains low except for two punctual increases during the first two peaks in input rate and before the last increase in input rate which lasts longer. The loss of stream elements (Figure 15c) is limited to 4.8%.

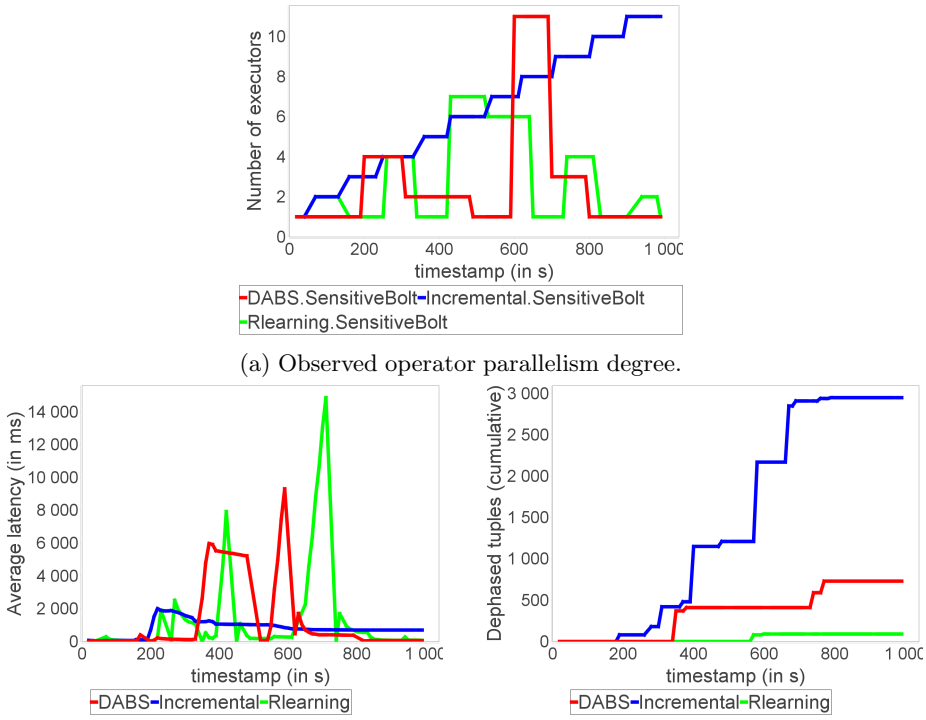
The *reinforcement learning* strategy provides only the suitable number of executors (Figure 15a) to avoid congestion.

While the *incremental* strategy maintains a low processing latency (Figure 15b) and delivers a throughput close to the input rate, it uses considerably more resources to complete the treatment of the entire stream (Figure 15a). Moreover, tuple loss (Figure 15c) has the worst score of all three approaches.

It is also interesting to notice that DABS-STORM is quite reactive, maintaining a smaller time shift between fluctuations in input rate and throughput than the reinforcement learning solution. So, even if stream elements arrive at high rates, the proactive reconfiguration performed by DABS-STORM does not delay their treatment.

We observe with this configuration that DABS-STORM offers the best compromise between performance with moderate increases, in average, processing latency and acceptable losses.

Real sensitive topology confronted with a progressive stream with data distribution fluctuations The topology used here, see Figure 10, is representative of real-world continuous queries. It includes common operators such as filters on values and attributes, joins with static databases and also user-defined functions from expert domains such as data mining.



(a) Observed operator parallelism degree.

(b) Average global processing latency.

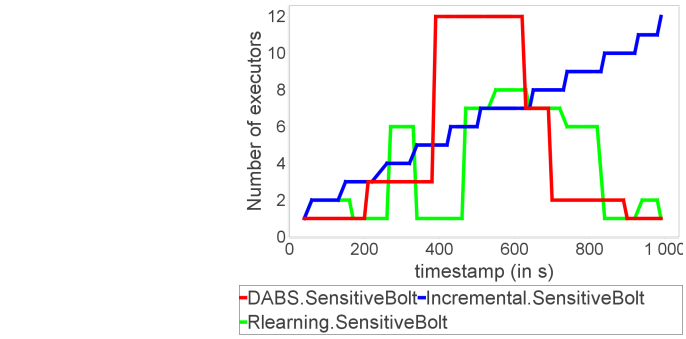
(c) Cumulated dephased tuples.

Fig. 15: Simple sensitive topology with erratic stream

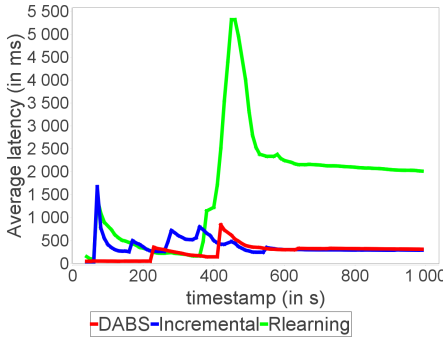
In such realistic conditions, DABS-Storm can take advantage of its global workflow approach. While a greater consumption of resources compared with other methods, see Figure 16a, can be observed, it all other solutions in terms of processing latency, see Figure 16b, and it also minimizes the loss of tuples, see Figure 16c, while the parallelism degree clearly adapts to the input rate, see Figure 16a.

Real sensitive topology confronted with an erratic stream The *reinforcement learning* strategy keeps reacting to local average input rate to adjust the parallelism degree. It results in an inconsistent scale-in at workflow scope, which is contradicted afterwards with a major impact on processing latency and result quality as illustrated on Figure 17b and Figure 17c.

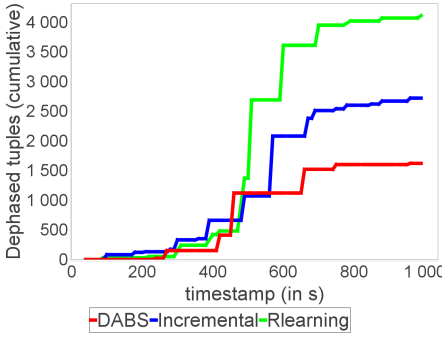
Even when confronted with erratic stream variations again, considering both processing latency (Figure 17b) and loss of tuples (Figure 17c), we can conclude that DABS-Storm copes better than other solutions. This confirms the interest of not limiting the analysis to local considerations for each operator, but rather of having a data-driven approach to analyze the behavior on the entire work-



(a) Observed operator parallelism degree.



(b) Average global processing latency.



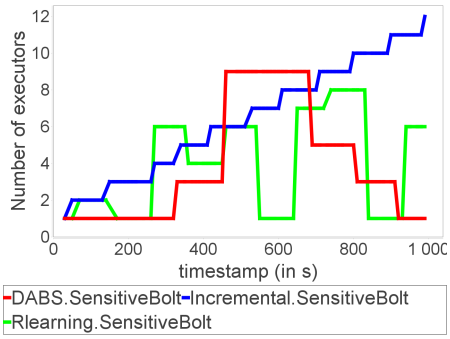
(c) Cumulated dephased tuples.

Fig. 16: Complex sensitive topology with progressive stream.

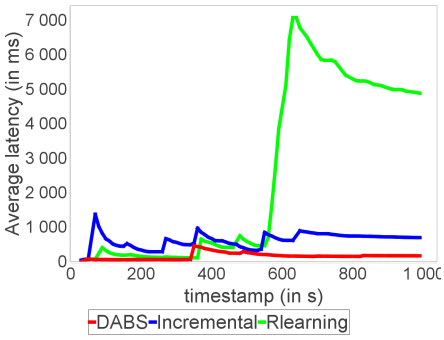
flow, as well as the complementarity and the compatibility of our two proposals, AUTOSCALE+ and OSG, making up DABS-STORM.

8 Conclusion

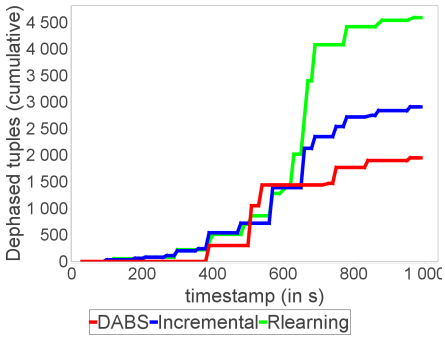
Proliferation and diversification of stream sources lead to new techniques in order to process large amounts of data with high velocity and quality. These techniques have to solve simultaneously three problems relating to management of the operators composing the workflow: parallelization, scheduling, and load balancing. In this paper, which focuses mainly on state-less operators, we have presented AUTOSCALE+, a proactive and coherent auto-parallelization strategy improving on AUTOSCALE [24]. It has been integrated into STORM with OSG, a cautious aware load balancing strategy, introducing a new member to the STORM family named DABS-STORM. Indeed, AUTOSCALE+ and OSG can be made perfectly compatible, complementing each other very well as two sides of the same coin. On the one side, OSG does not work well if the parallelism degree is underestimated, while AUTOSCALE anticipates to avoid such situation. On the other side, OSG



(a) Observed operator parallelism degree.



(b) Average global processing latency.



(c) Cumulated dephased tuples.

Fig. 17: Complex sensitive topology with erratic stream

improves load balancing, thus having a positive effect on the accuracy of estimations conducted by AUTOSCALE + and reducing unnecessary reconfigurations. Such an agreement between two strategies is not systematic. For example, the experiments conducted highlight some incompatibility problems between OSG and the progressive strategy, which changes the parallelism degree one step at a time. Furthermore, DABS-Storm can be used in combination with different existing scheduling strategies.

Experimental evaluations have shown that DABS-Storm improves the system stability and performances. For example, even when facing brief and unpredictable fluctuations in input data streams, Autoscale+ keeps loss of tuples under 18%. On complex real workflows, thanks to a global workflow analysis, DABS-Storm does even better, cutting losses to 10%. As long as the necessary resources are available, faced with large or very large fluctuations in input data streams, whether in terms of volume or data distribution (as can be observed, for example, in microblogging analysis), DABS-Storm is able to adapt. This automatic adaptation has many advantages. First, human supervision is no longer required to trigger and manage system reconfigurations. Note that a scarcity of resource remains a problem. Indeed, in the event of lack of resources blocking

scale-out, DABS-Storm does not change its strategies as one would expect. One of our future research goals is thus to study the problem of resources starvation and to search for a solution that maximizes system throughput. Second, thanks to careful load balancing, dynamic parallelism degree adaptation with a global workflow analysis, and a data-driven approach, DABS-Storm manages computing resources better, reaching an interesting equilibrium between system stability and limited resource consumption.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings. pp. 277–289. www.cidrdb.org (2005), http://cidrdb.org/cidr2005/papers/P23.pdf
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003). https://doi.org/10.1007/s00778-003-0095-z, https://doi.org/10.1007/s00778-003-0095-z
3. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: Chakravarthy, S., Urban, S.D., Pietzuch, P.R., Rundensteiner, E.A. (eds.) The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013. pp. 207–218. ACM (2013). https://doi.org/10.1145/2488222.2488267, http://doi.acm.org/10.1145/2488222.2488267
4. Apache Flink: https://flink.apache.org/
5. Apache Storm: https://storm.apache.org/
6. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: the stanford data stream management system. In: Garofalakis, M.N., Gehrke, J., Rastogi, R. (eds.) Data Stream Management - Processing High-Speed Data Streams, pp. 317–336. Data-Centric Systems and Applications, Springer (2016). https://doi.org/10.1007/978-3-540-28608-0_16, https://doi.org/10.1007/978-3-540-28608-0_16
7. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. **15**(2), 121–142 (2006). https://doi.org/10.1007/s00778-004-0147-z, https://doi.org/10.1007/s00778-004-0147-z
8. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Load management and high availability in the medusa distributed stream processing system. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 929–930. ACM (2004)
9. Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H.N., Moran, C.: IBM infosphere streams for scalable, real-time, intelligent transportation services. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010. pp. 1093–1104. ACM

(2010). https://doi.org/10.1145/1807167.1807291, http://doi.acm.org/10.1145/1807167.1807291
10. Box, G.: Box and Jenkins: Time Series Analysis, Forecasting and Control, pp. 161–215. Palgrave Macmillan UK, London (2013). https://doi.org/10.1057/9781137291264_6, http://dx.doi.org/10.1057/9781137291264_6
11. Carter, L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. **18**(2), 143–154 (1979). https://doi.org/10.1016/0022-0000(79)90044-8, https://doi.org/10.1016/0022-0000(79)90044-8
12. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., Shah, M.A.: Telegraphcq: Continuous dataflow processing. In: Halevy, A.Y., Ives, Z.G., Doan, A. (eds.) Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003. p. 668. ACM (2003). https://doi.org/10.1145/872757.872857, http://doi.acm.org/10.1145/872757.872857
13. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. In: CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings. www.cidrdb.org (2003), http://www-db.cs.wisc.edu/cidr/cidr2003/program/p23.pdf
14. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005). https://doi.org/10.1016/j.jalgor.2003.12.001, https://doi.org/10.1016/j.jalgor.2003.12.001
15. Das, R., Tesauro, G., Walsh, W.E.: Model-based and model-free approaches to autonomic resource allocation. Tech. Rep. RC23802, IBM Research Report (Nov 2005), http://domino.watson.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/f5e3b7f574b24bad852570c1005e35a9!OpenDocument&Highlight=0,tesauro
16. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Brewer, E.A., Chen, P. (eds.) 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004. pp. 137–150. USENIX Association (2004), http://www.usenix.org/events/osdi04/tech/dean.html
17. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. VLDB J. **23**(4), 517–539 (2014). https://doi.org/10.1007/s00778-013-0335-9, https://doi.org/10.1007/s00778-013-0335-9
18. Gedik, B., Schneider, S., Hirzel, M., Wu, K.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. **25**(6), 1447–1463 (2014). https://doi.org/10.1109/TPDS.2013.295, https://doi.org/10.1109/TPDS.2013.295
19. Golab, L., Garg, S., Özsu, M.T.: On indexing sliding windows over online data streams. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2992, pp. 712–729. Springer (2004). https://doi.org/10.1007/978-3-540-24741-8_41, https://doi.org/10.1007/978-3-540-24741-8_41
20. Google Cloud Dataflow: https://cloud.google.com/dataflow/

21. Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C.: Auto-scaling techniques for elastic data stream processing. In: Bellur, U., Kothari, R. (eds.) The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014. pp. 318–321. ACM (2014). https://doi.org/10.1145/2611286.2611314, http://doi.acm.org/10.1145/2611286.2611314
22. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4), 46:1–46:34 (2013). https://doi.org/10.1145/2528412, http://doi.acm.org/10.1145/2528412
23. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In: Dayal, U., Ramamritham, K., Vijayaraman, T.M. (eds.) Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India. pp. 341–352. IEEE Computer Society (2003). https://doi.org/10.1109/ICDE.2003.1260804, https://doi.org/10.1109/ICDE.2003.1260804
24. Kombi, R.K., Lumineau, N., Lamarre, P.: A preventive auto-parallelization approach for elastic stream processing. In: Lee, K., Liu, L. (eds.) 37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017. pp. 1532–1542. IEEE Computer Society (2017). https://doi.org/10.1109/ICDCS.2017.253, https://doi.org/10.1109/ICDCS.2017.253
25. Mukhopadhyay, A., Mazumdar, R.R.: Analysis of randomized join-the-shortest-queue (JSQ) schemes in large heterogeneous processor-sharing systems. IEEE Trans. Control of Network Systems **3**(2), 116–126 (2016). https://doi.org/10.1109/TCNS.2015.2428331, https://doi.org/10.1109/TCNS.2015.2428331
26. Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. In: Gehrke, J., Lehner, W., Shim, K., Cha, S.K., Lohman, G.M. (eds.) 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015. pp. 137–148. IEEE Computer Society (2015). https://doi.org/10.1109/ICDE.2015.7113279, https://doi.org/10.1109/ICDE.2015.7113279
27. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: distributed stream computing platform. In: Fan, W., Hsu, W., Webb, G.I., Liu, B., Zhang, C., Gunopulos, D., Wu, X. (eds.) ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010. pp. 170–177. IEEE Computer Society (2010). https://doi.org/10.1109/ICDMW.2010.172, https://doi.org/10.1109/ICDMW.2010.172
28. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.H.: R-storm: Resource-aware scheduling in storm. In: Lea, R., Gopalakrishnan, S., Tilevich, E., Murphy, A.L., Blackstock, M. (eds.) Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015. pp. 149–161. ACM (2015). https://doi.org/10.1145/2814576.2814808, http://doi.acm.org/10.1145/2814576.2814808
29. Rivetti, N., Anceaume, E., Busnel, Y., Querzoni, L., Sericola, B.: Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems. In: Proceedings of the 17th ACM/IFIP/USENIX International Middleware Conference. Middleware (2016)

30. Rivetti, N., Busnel, Y., Querzoni, L.: Load-aware shedding in stream processing systems. In: Gal, A., Weidlich, M., Kalogeraki, V., Venkasubramanian, N. (eds.) Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016. pp. 61–68. ACM (2016). https://doi.org/10.1145/2933267.2933311, http://doi.acm.org/10.1145/2933267.2933311
31. Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B.: Efficient key grouping for near-optimal load balancing in stream processing systems. In: Eliassen, F., Vitenberg, R. (eds.) Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015. pp. 80–91. ACM (2015). https://doi.org/10.1145/2675743.2771827, http://doi.acm.org/10.1145/2675743.2771827
32. Sattler, K., Beier, F.: Towards elastic stream processing: Patterns and infrastructure. In: Cormode, G., Yi, K., Deligiannakis, A., Garofalakis, M.N. (eds.) Proceedings of the First International Workshop on Big Dynamic Distributed Data, Riva del Garda, Italy, August 30, 2013. CEUR Workshop Proceedings, vol. 1018, pp. 49–54. CEUR-WS.org (2013), http://ceur-ws.org/Vol-1018/paper9.pdf
33. Schneider, S., Andrade, H., Gedik, B., Biem, A., Wu, K.: Elastic scaling of data parallel operators in stream processing. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. pp. 1–12. IEEE (2009). https://doi.org/10.1109/IPDPS.2009.5161036, https://doi.org/10.1109/IPDPS.2009.5161036
34. Senderovich, A., Weidlich, M., Gal, A., Mandelbaum, A.: Queue mining - predicting delays in service processes. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8484, pp. 42–57. Springer (2014). https://doi.org/10.1007/978-3-319-07881-6_4, https://doi.org/10.1007/978-3-319-07881-6_4
35. Stonebraker, M., Çetintemel, U., Zdonik, S.B.: The 8 requirements of real-time stream processing. SIGMOD Record **34**(4), 42–47 (2005). https://doi.org/10.1145/1107499.1107504, http://doi.acm.org/10.1145/1107499.1107504
36. Sullivan, M., Heybey, A.: Tribeca: A system for managing large databases of network traffic. In: Douglis, F. (ed.) 1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998. USENIX Association (1998), https://www.usenix.org/conference/1998-usenix-annual-technical-conference/tribeca-system-managing-large-databases-network
37. Vengerov, D., Menck, A.C., Zaït, M., Chakkappen, S.: Join size estimation subject to filter conditions. PVLDB **8**(12), 1530–1541 (2015). https://doi.org/10.14778/2824032.2824051, http://www.vldb.org/pvldb/vol8/p1530-vengerov.pdf
38. Wu, Y., Tan, K.: Chronostream: Elastic stateful stream computation in the cloud. In: 2015 IEEE 31st International Conference on Data Engineering. pp. 723–734 (April 2015). https://doi.org/10.1109/ICDE.2015.7113328
39. Xu, J., Chen, Z., Tang, J., Su, S.: T-storm: Traffic-aware online scheduling in storm. In: IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014. pp. 535–544. IEEE Computer Society (2014). https://doi.org/10.1109/ICDCS.2014.61, https://doi.org/10.1109/ICDCS.2014.61

40. Xu, L., Peng, B., Gupta, I.: 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. pp. 22–31. IEEE Computer Society (2016). https://doi.org/10.1109/IC2E.2016.38, https://doi.org/10.1109/IC2E.2016.38
41. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: A fault-tolerant model for scalable stream processing. Tech. Rep. UCB/EECS-2012-259, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE (2012), http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.html