A preventive auto-parallelization approach for elastic stream processing

Roland Kotto Kombi
Univ Lyon
INSA de Lyon
CNRS UMR5205
LIRIS
F69621, Villeurbanne
Email: roland.kotto-kombi@liris.cnrs.fr

Nicolas Lumineau
Univ Lyon
University Claude Bernard
CNRS UMR5205
LIRIS
F69621, Villeurbanne
Email: nicolas.lumineau@liris.cnrs.fr

Philippe Lamarre
Univ Lyon
INSA de Lyon
CNRS UMR5205
LIRIS
F69621, Villeurbanne
Email: philippe.lamarre@liris.cnrs.fr

***Abstract*—Nowadays, more and more sources (connected devices, social networks, etc.) emit real-time data with fluctuating rates over time. Existing distributed stream processing engines (SPE) have to resolve a difficult problem: deliver results satisfying end-users in terms of quality and latency without over-consuming resources. This paper focuses on parallelization of operators to adapt their throughput to their input rate. We suggest an approach which prevents operator congestion in order to limit degradation of results quality. This approach relies on an automatic and dynamic adaptation of resource consumption for each continuous query. This solution takes advantage of i) a metric estimating the activity level of operators in the near future ii) the AUTOSCALE approach which evaluates the need to modify parallelism degrees at local and global scope iii) an integration into the *Apache Storm* solution. We show performance tests comparing our approach to the native solution of this SPE.**

I. Introduction

With the proliferation of data stream sources (connected devices, sensors, social networks, etc.), specific issues related to data volume and velocity have emerged. To address these Big Data issues, some parallel and distributed solutions have been developed [1]–[5].

Streams are potentially infinite sequences of items with fluctuating rates and distribution over time. Users query these streams through the definition of *continuous queries* [6], which compute some results as soon as they receive new items. For example, an application looking for traffic jam detection consumes raw data streams from speed cameras. Users can specify continuous queries implicitly by using a declarative query language [6] or explicitly by building a graph of operators [1]–[3]. The continuous query is turned into a direct acyclic graph (DAG) of operators, denoted *workflow* or *topology*. When processing multiple continuous queries at the same time, some techniques [7] like query rewriting can identify operators shared by many queries in order to execute only once the same logical operator. We consider here selected workflows after multiple query optimization. Each operator is potentially processed in parallel [7], [8] by a set of threads. This workflow is then distributed on a cluster of machines. Considering our previous example on traffic jam detection, sensor data are collected by an operator, which groups them by location to compute average values every 15 minutes. These averages are sent to an operator, which filters critical values and raises an alert if necessary. Finally, a persistence operator stores values into a remote database.

As we consider streams with rates that fluctuate continuously, a SPE has fluctuating processing costs over time. Thus, the resources required (CPU, RAM and bandwidth) vary too. It appears then unavoidable that SPEs have to adapt dynamically their usage of resources (*i.e.* number of allocated processing units) to their processing requirements. Indeed, an oversized usage implies large overheads due to massive network traffic [9], [10]. On the contrary, an undersized usage leads to *congestion* [11] of the system.

In order to adapt usage of resources, a SPE can manage two aspects: scheduling and parallelism of operators. The first solution consists in changing scheduling of threads associated to an operator. This allows a thread to be moved from one machine to another. The aim of scheduling management is to keep to a target concerning usage of resources on each machine [2], [9], [10], [12]. For example, a scheduling strategy based on load balancing [2] between machines allocates a maximal number of available machines and spreads treatments over them. On the contrary, a scheduling strategy based on network traffic management [10] tends to concentrate treatments on a subset of machines. The second solution affects operator parallelism and defines how many threads handle the incoming load of an operator at runtime. Under a certain threshold, the more there are, the less each item waits before being processed.

In this paper, we aim at tackling the congestion issue. Actually, an increase of input rate may lead an operator to its congestion. The impact in a congestion is an unacceptable end-to-end latency leading to system failure.

Some works [1], [2], [9], [10], [12] suggest scheduling strategies which remove effective congestion of operators only under the restrictive assumption that there is an optimal scheduling plan according to a fixed set of operator instances. We focus then on *auto-parallelization* strategies, changing dynamically the set of operator instances in order to tackle this issue.

Recent works [13]–[15] suggest approaches for elastic

stream processing, which tend to optimize performance and resource usage. We consider as resources, the CPU, RAM and bandwidth of processing units. To our knowledge, these approaches do not aim at anticipating operator congestion as they are curative. Moreover, some of them require user action.

We present a preventive approach relying on a metric which estimates operator activity in the near future. This metric is computed dynamically from the recent history for each operator. Our *auto-parallelization* approach, named AUTOSCALE, can increase (*scale-out*) or decrease (*scale-in*) the parallelism degree of each operator. AUTOSCALE takes into account the local and the global context to identify necessary and coherent modifications of parallelism degrees. It improves the performance and controls the stability of the system.

In the remainder of this article, Section II presents the context of execution and the challenges raised by congestion management. In Section III, we discuss related works in more detail. Then, in Section IV, we describe metrics and the AUTOSCALE approach. Finally, we present an experimental evaluation comparing AUTOSCALE to the native strategy of Apache Storm in Section V.

II. Background and Challenges

A. Background

In our context, we shall consider three continuous queries Q_1, Q_2 and Q_3 taking S_1, S_2 and S_3 as input streams. As shown on Figure 1, these queries are represented by workflows W_1, W_2 and W_3 which correspond to their respective execution plans.

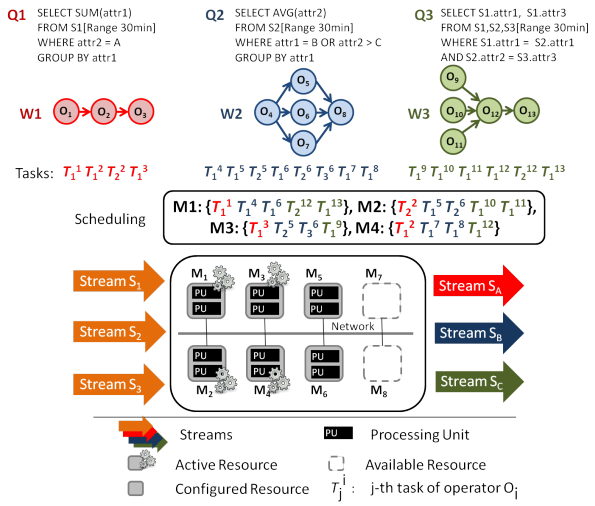


Fig. 1. Execution context

W_1 is a linear topology, W_2 is a diamond topology and W_3 is a star topology. Each of these topology represents an elementary topology and the most complex topologies can be considered as a composition of these topologies[12].

To execute these topologies, each operator is mapped to a set of *threads*. The number of threads executing an operator is called its *parallelism degree*. On Figure 1, $\mathcal{O}_2$ is linked to threads T_1^2 and T_2^2, so its parallelism degree is two.

In order to run, these threads are then assigned to the processing unit of machines M_1 to M_8 according to a scheduling plan. In the example illustrated on Figure 1, threads of topology W_1 are distributed on machines M_1 to M_4.

Still on this example, we consider three machine roles. First, machines M_1 to M_4 are active because they process threads assigned on them. Then, M_5 and M_6 are configured but inactive because there is no thread assigned on them. Finally, M_7 and M_8 are available resources but they are not configured so out of the scheduler's reach.

B. Assumptions

We consider that each continuous query is composed of stateless or stateful operators with respect to some assumptions. First, considering a set of continuous queries running simultaneously, we assume that there are enough available resources to process all queries (**H1**).

Then, we consider only SPEs able to manage states of stateful operators (**H2**) like several well-known solutions [2], [3], [16].

Concerning the execution of each query, we consider that all operators can be processed in parallel by multiple threads and scheduled potentially on different machines. The global incoming load is divided evenly between threads executing the same operator (**H3**).

These threads are assigned to machines according to a scheduling strategy. We assume that this strategy revises operator placement periodically. Moreover, this strategy assigns at most one thread for each operator on a given processing unit (**H4**).

C. Challenges

Each operator applies a function defined by a user on each input. For stateless operators, an input corresponds to an item, while for stateful ones, an input matches to a set of items. Depending on the time complexity of its function and available CPU, an operator can process, on average, a certain number of inputs per time unit. This number is called the *capacity* of the operator. This capacity limits the input rate that an operator can handle with a single thread. Given a thread executing an operator, congestion may occur when input rate is greater than capacity. There are two possible cases of this: a load balancing issue between threads executing the same operator or a critical change in global input rate. The first case is out of our scope and has been studied in [17]. We focus then on the second case in our context. To limit the impact of a critical input rate, two solutions are possible: changing the current affectations of critical operators or increasing their parallelism degree.

Faced with effective congestion, the scheduler can move operators from overused nodes to less busy ones. While this provides operators with more available CPU, capacity is improved only if the new node has more available resources.

In order to prevent congestion, a SPE should be able to detect when an operator's input rate reaches or exceeds its

capacity. Indeed, a detection based on resource consumption is only a solution for effective congestion. It is not satisfactory because treatment quality deteriorates before the system has time to reconfigure itself. Even if no item is lost, overall latency suffers from the congestion of one or more operators.

Yet, it is not easy to decide when to increase (*scale-out*) or to decrease (*scale-in*) the parallelism degree of an operator. Actually, for any one operator, if its input rate exceeds its capacity, the operator tends towards congestion. However, congestion is effective only if the input rate remains equal to or greater than capacity for a significant time. In any other case, triggering a scale-out too early will lead to the activation of one or more unnecessary threads. This will affect system stability and generate large reconfiguration overheads. It is then crucial that a relevant strategy takes system stability into account. Moreover, it is important that this strategy reduces the parallelism degree of underused operators. It should fit global capacity to processing needs and, depending on the scheduling strategy, free unnecessary processing units which are then available for other queries.

To sum up, automatic and dynamic adaptation of operators capacities requires that SPEs can detect potential congestion before it becomes effective in most cases. Moreover, a relevant strategy should not overreact because of reconfiguration overheads. Finally, the system should fit operators capacities to their processing needs in order to consume only necessary resources. Finding a satisfactory compromise between these issues is a major challenge for elastic stream processing. Indeed, with the growing popularity of *pay-as-you-consume* solutions[18] and the emergence of *Green IT*, it is crucial that the current generation of SPEs takes treatment elasticity into account.

III. Related Works

The performance of a SPE and the quality of its results rely mainly on its reactivity to fluctuations in its execution context. The problem is complex because it requires adaptation of resource usage, while avoiding massive reconfiguration overheads that affect SPE stability.

SPEs must integrate a strategy to confront large fluctuations in input rate. This strategy is either curative or preventive. In the first case, the SPE reacts exclusively to existing congestion of operators, while in the second case, the SPE is able to anticipate them. In both cases, if the SPE triggers some mechanisms to remove or avoid automatically a congestion at runtime, it is considered to be automatic. In other cases, it is not an automatic approach.

Some solutions [9], [10] are based on the state of resources and the network traffic to determine an optimal operator scheduling. This allows global latency of a topology to be reduced as it avoids large network overheads. However, these solutions rely exclusively on scheduling management (see Section II) and are curative. In particular, they do not adapt the parallelism degree. This adaptation then requires additional solutions.

Solutions based on maximal usage of resources [3], [16] serve as a guarantee, according to hypothesis H1, that continuous queries can be processed without loss of quality. Unfortunately, these solutions are inappropriate in a context simultaneously. Moreover, from an economical and energetic point of view, it is not advisable to spread treatments without taking load fluctuations into account.

In [11], authors suggest an algorithm enabling scale-in or scale-out on demand. This approach relies on an external intervention or a script not explained by the authors to detect automatically a need for more resources. Actually, there is no information about the transition from an available machine to a configured one, thus making this approach inefficient with a constant number of configured resources. Finally, this approach is curative and cannot prevent operator congestion.

Some solutions [13], [14] make it possible to adapt dynamically and automatically the parallelism degree of operators with a constant number of configured resources. Nevertheless, they rely on the detection of effective operator congestion. Even if they can reduce congestion duration, they cannot anticipate it. Likewise, in [15], authors suggest a solution based on a learning algorithm. According to this method, the system is able to select the parallelism degree maximizing performance gain after a learning phase. However, this method is based on active resource monitoring (CPU and RAM), making anticipation of operator congestion a complex process.

Finally, concerning anticipation of operator congestion, some works [19], [20] based on queuing theory have been suggested. As presented in [15], [19], queuing theory relies on a detailed system model and do not fit to dynamic modification of execution context such as frequent fluctuations in input rate or add of processing units at runtime. We choose then time series analysis algorithms [21] as forecasting method.

IV. AUTOSCALE APPROACH

A. Goal

As discussed in Section II, we aim at suggesting a strategy able to prevent operator congestion. In AUTOSCALE, we seek to define an auto-parallelization strategy which fits usage of resources dynamically and automatically in order to prevent congestion and free unnecessary resources. We consider that an adequate usage of resources, with respect to given treatments, yields results with acceptable item loss and minimum latency.

B. Monitoring formalization

We introduce the formalization of monitored values used by our auto-parallelization strategy.

Let $\mathcal{T} = (\mathcal{O}, \mathcal{V})$ the topology of a continuous query represented as a direct acyclic graph where $\mathcal{O}$ is the set of operators and $\mathcal{V}$ the set of streams. We consider that each operator $\mathcal{O}_i$ can be processed in parallel by a set of threads. The parallelism considered in this paper is *data parallelism* [7]. The number of threads, denoted $deg(\mathcal{O}_i)$, defines the parallelism degree of operator $\mathcal{O}_i$.

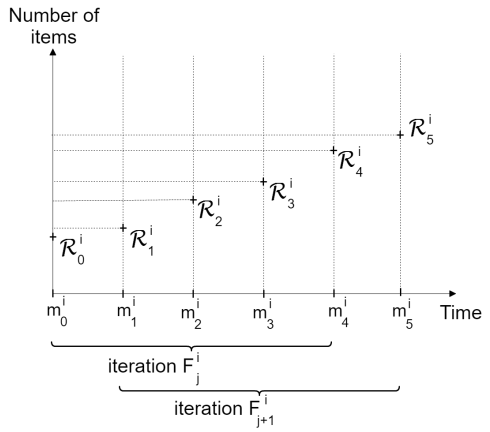


Fig. 2. Monitoring window

Let $\mathcal{F}$ a set of monitoring sliding windows $\mathcal{F}_i = \{(F_j^i)\}_{j\in\mathbb{N}^+}$. As illustrated on Figure 2, each window $\mathcal{F}_i$ is associated with the operator $\mathcal{O}_i$ and is composed of iterations F_j^i. Each F_j^i is defined by a duration Δ and collects measurements collected during this interval. These measurements are collected according to a predefined set of timestamps $\mathcal{M}_i = \{m_1^i, m_2^i, ..., m_n^i\}_{n\in\mathbb{N}^+}$. For each operator $\mathcal{O}_i$, we collect measurements taking into account items received and processed in the interval $[m_{k-1}^i, m_k^i[$ with k=1,...,n. It is worth noting that a master process (*e.g. JobTracker for Hadoop*) serves as guarantee that measurements are collected synchronously on each processing unit. Moreover, as mentioned in Section II, we consider that the scheduling strategy runs periodically, so we assume that this period corresponds to the size of the interval $[m_{k-1}^i, m_k^i[$. To improve monitoring effectiveness, the size of this interval should be greater than the time required to pre-process and store measurements in a standard database management system, which is around a second. In addition, the monitoring window size should be greater than the size of the greatest processing window of a stateful operator belonging to a given topology. It ensures that all metrics presented in the remainder of this section can be computed for both stateless and stateful operators. Finally, scale-in and scale-out actions performed by AUTOSCALE have no impact on monitored values as we take a grace period into account after each reconfiguration.

Let $\mathcal{R}^i$ the set, potentially infinite, of items received by operator $\mathcal{O}_i$. We consider $\mathcal{R}^{i,j}$ as the subset of items received by $\mathcal{O}_i$ during the iteration F_j^i, and $\mathcal{R}_k^i$ the subset of items received between $[m_{k-1}^i, m_k^i[$. In the example presented on Figure 2, $\mathcal{R}^{i,j}$ is the sum of measurements $\mathcal{R}_0^i$ to $\mathcal{R}_4^i$, and $\mathcal{R}^{i,j+1}$ the sum of measurements $\mathcal{R}_1^i$ to $\mathcal{R}_5^i$.

In addition to the number of items received, we collect the processing latency per item of the operator observed during F_j^i, denoted $Lat_{F_j^i}$. This does not include the time an item may spend in pending queues.

C. Estimation of operator activities at local scope

Now that we have defined how each operator is monitored, we introduce metrics computed from monitored values to estimate the activity of each operator.

1) Metrics: The aim is to estimate if the number of items an operator has to process is compatible with its current capacity. If it is not, the operator is considered as a potential source of congestion. The input size an operator has to process during an iteration is defined as the number of items received during this iteration added to the pending items received during previous iterations. The effective input size during the iteration F_j^i is then:

$$Input_j^i = |\mathcal{R}^{i,j}| + pending_{F_{j-1}^i} \tag{1}$$

Because of the value $|\mathcal{R}^{i,j}|$ in formula (1), this effective input size can only be computed at the end of the iteration F_j^i. To anticipate operator congestion, we need to estimate $|\mathcal{R}^{i,j}|$ at the end of F_{j-1}^i. We estimate the number of items received during F_j^i with a linear regression based on measurements collected during F_{j-1}^i. Let f_{j-1}^i the affine function computed by linear regression and based on pairs (m_k^i, $\mathcal{R}_k^i$) collected during F_j^i. The estimation of items received during F_j^i is then:

$$|Estim\mathcal{R}^{i,j}| = \sum_{m_k^i \in \mathcal{M}_i} \lceil f_{j-1}^i(m_k^i) \rceil \tag{2}$$

We choose linear regression to compute estimations arbitrarily. Up to now, the choice of this quite simple solution has not been contradicted by experimental results.

The expected input size during F_j^i is then defined as follows:

$$EstimInput_{F_j^i} = |Estim\mathcal{R}^{i,j}| + pending_{F_{j-1}^i} \tag{3}$$

Now that we have an approximation of the future input size of each operator $\mathcal{O}_i$ during F_j^i, we have to estimate their respective capacities on F_j^i. We consider operator capacity, the average number of items this operator is able to process during an iteration of size Δ.

$$Capacity_{F_j^i} = \frac{1}{Lat_{F_j^i}} \times \Delta \times deg(\mathcal{O}_i) \tag{4}$$

We use covariance to estimate the capacity of $\mathcal{O}_i$ during F_j^i.

$$EstimCapacity_{F_j^i} = Capacity_{F_{j-1}^i} + \epsilon_i \tag{5}$$

where ϵ_i is the covariance between the capacity during F_{j-1}^i and the capacities observed during previous iterations.

The above estimations allow us to define a control metric evaluating the activity of an operator at its local scope. The notion of *Local Activity Level*, denoted LAL, represents intuitively the balance between parallelism degree and future input size. It is defined by:

$$LAL_{F_j^i} = \frac{EstimInput_{F_j^i}}{EstimCapacity_{F_j^i}} \tag{6}$$

Let θ_{min} and θ_{max} two thresholds delimiting respectively a low and a high activity level. For a given operator, the interpretation of LAL value is as follows:

- If $LAL_{F_j^i} \leq \theta_{min}$, the local activity of the operator is 'low' because operator capacity is too great in comparison to $EstimInput_{F_j^i}$.
- If $\theta_{min} < LAL_{F_j^i} \leq \theta_{max}$, the local activity of the operator is 'medium' because the operator is able to process all items during F_j^i.
- If $\theta_{max} < LAL_{F_j^i} \leq 1$, the local activity of the operator is 'high' because the operator has just the capacity to process items waiting to be processed during F_j^i.
- If $LAL_{F_j^i} > 1$, the local activity of the operator is then 'critical' because the operator is not able to process now $EstimInput_{F_j^i}$ with its estimated capacity $EstimCapacity_{F_j^i}$.

2) Principle: The AUTOSCALE approach determines local activities. Each operator is mapped to a vertex of an attributed graph. Each attribute corresponds to a local metric as presented on Figure 3. We denote this attributed graph the *instantaneous graph of local activities* (IGLA). An example of IGLA is illustrated on Figure 3 for query $Q1$ introduced in Section II.

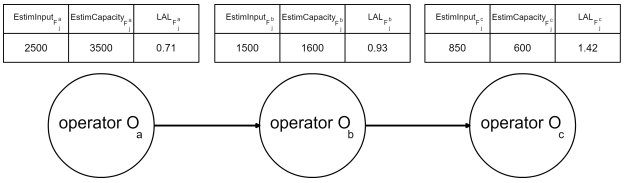


Fig. 3. An example of IGLA

D. Qualitative estimation of scale-in/out requirements at global scope

As soon as the IGLA has been built, the AUTOSCALE approach checks its global consistency. For this, we introduce a metric that quantifies the effect of local activities on next operators.

1) Metrics: Let us consider a monitored operator $\mathcal{O}_i$ during an iteration F_j^i. We have measured the total number of items it has processed, denoted $process_{F_j^i}$, and the total number of items it has emitted during F_j^i, denoted $output_{F_j^i}$. We compute its selectivity factor $SF_{F_j^i}$ during the iteration F_j^i as follows:

$$SF_{F_j^i} = \frac{output_{F_j^i}}{process_{F_j^i}} \tag{7}$$

According to estimation of the future input size $EstimInput_{F_j^i}$ and the current capacity $Capacity_{F_j^i}$ of the operator $\mathcal{O}_i$, we confine the estimated number of processed elements during the next iteration, denoted $EstimProcess_{F_j^i}$, to the following value:

$$EstimProcess_{F_j^i} = min(EstimInput_{F_j^i}, Capacity_{F_j^i} \times \Delta) \tag{8}$$

It is a fact that an operator can at most process the number of items corresponding to its capacity per time unit multiplied by the duration of an iteration. With this estimation and the selectivity factor $SF_{F_j^i}$, we estimate then number of items emitted by $\mathcal{O}_i$ on F_{j+1}^i, denoted $EstimOuput_{F_{j+1}^i}$ with respect to the following formula:

$$EstimOutput_{F_{j+1}^i} = EstimProcess_{F_j^i} \times SF_{F_j^i} \tag{9}$$

According this value, it is now possible to obtain a complementary estimation of the future input size of next operators. Concretely, let us consider a child operator $\mathcal{O}_c$ receiving its inputs from a parent operator $\mathcal{O}_p$. The value $EstimOutput_{F_{j+1}^p}$ is intrinsically different from $EstimInput_{F_j^c}$ since it is not based on items already received by $\mathcal{O}_c$ as illustrated on Figure 4. Indeed, $EstimOutput_{F_{j+1}^p}$ is computed from items received and processed by the previous operator, in this example, $\mathcal{O}_p$. Intuitively, it offers a better anticipation of critical variations of the global input rate.

2) Global consistency strategy: So, still considering an operator $\mathcal{O}_c$ receiving its inputs from an operator $\mathcal{O}_p$, we have at our disposal two distinct estimations of the future input size of $\mathcal{O}_c$: its local estimation $EstimInput_{F_j^c}$ and the global estimation $EstimOutput_{F_{j+1}^p}$. The choice of the estimation to consider depends on which aspect the SPE should favor.

If the SPE serves as guarantee that the capacity of each operator remains great enough to absorb its inputs, the maximal value between $EstimInput_{F_j^c}$ and $EstimOutput_{F_{j+1}^p}$ is considered to adjust the parallelism degree of $\mathcal{O}_c$. According to available estimations and H1, it ensures that each operator is able to process all incoming items. Nevertheless, it prevents scale-in from being performed until local and global estimations confirm that it does not lead to a potential congestion.

On the contrary, if the SPE aims at using only necessary resources, the minimal value between $EstimInput_{F_j^c}$ and $EstimOutput_{F_{j+1}^p}$ is used. With this combination strategy, the SPE decreases operator capacity as soon as it is locally or globally advisable. However, the drawback of this strategy is that it affects system stability. Indeed, decreasing operator capacity to save resources as soon as possible also means increasing them whenever input size increases significantly.

For both combination strategies, we consider the globally consistent estimation as the result of a function $combine$, which takes both estimations as input and returns the globally consistent estimation according to the SPE's goal.

3) Principle: As presented in Algorithm 1, we perform a BFS on the IGLA from child operators of sources. Indeed, local estimations of source operators cannot be contradicted by the global context. We check whether any parent of the operator has a critical activity. If the function $activity()$ returns that the current operator has a critical parent at local scale and the method $unchecked()$ returns that it has not been already

Algorithm 1 Global consistency checking

Require: an operator $\mathcal{O}_i$, the set $parents$ of parent operators of $\mathcal{O}_i$, the local-based IGLA
Ensure: the parallelism degree of $\mathcal{O}_i$ according to the global context

```
current ← currentDeg(O_i);
for all parent in parents do
  if activity(parent) == 'critical' ∧ unchecked(O_i) then
    EstimParentOutput ← Σ EstimOutput_{F_{j+1}^{P_i}};
    EstimInput_{F_j^i} ← combine(EstimInput_{F_j^i}, EstimParentOutput);
    next ← deg_j(O_i);
    if current > next then
      setScaleIn(IGLA, O_i, next);
    end if
    if current == next then
      setNothing(IGLA, O_i, current);
    end if
    if current < next then
      setScaleOut(IGLA, O_i, next);
    end if
    checked ← checked ∪ {O_i};
  end if
end for
```

checked, we compute the globally consistent estimation of its future input size and replace its local estimation. This replacement propagates the effect of critical estimation to all operators processing items emitted by an operator. Then, we compare the current parallelism degree given by function $currentDeg()$ and the adequate parallelism. According to this comparison, we map each operator to a modification of parallelism degree.

TABLE I
DECISION MATRIX FOR IGLA COMPUTATION

Operator activity \ Evolution trend of inputs	Decreasing or constant	Increasing
low activity	*scale-in*	*nothing*
medium activity	*nothing*	*nothing*
high activity	*nothing*	*scale-out*
critical activity	*scale-out*	*scale-out*

Now that we have a globally consistent estimation of input size, we define the notion *Global Activity Level*, denoted GAL, of an operator $\mathcal{O}_i$ defined as follows:

$$GAL_{F_j^i} = \frac{combine(EstimInput_{F_j^i}, \sum EstimOutput_{F_{j+1}^{P_i}})}{EstimCapacity_{F_j^i}} \tag{10}$$

where $\sum EstimOutput_{F_{j+1}^{P_i}}$ is the sum of the estimated outputs of all parent operators of $\mathcal{O}_i$. According to Table I, we can evaluate at a global scope if a scale-in or a scale-out is needed. We can map each operator to a modification of its parallelism degree. This decision takes into account the global activity of a given operator and the evolution trend of its input size. As a reminder, the affine function f_j^i is computed with linear regression to estimate the future input size of an operator (see formula (2)). The derivative value of this function is used to evaluate the evolution trend of input size. If this value is strictly positive, input size is considered as increasing. Otherwise it is estimated as decreasing or constant.

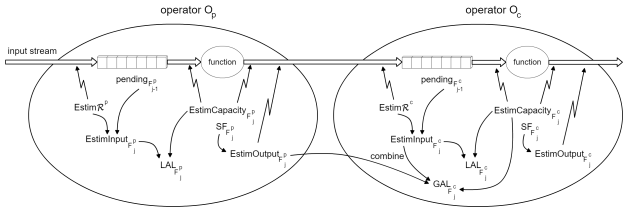


Fig. 4. Estimations at local and global scope

To summarize, AUTOSCALE estimates activity at local and global scope for each operator as illustrated on Figure 4. At local scope, AUTOSCALE computes an estimation of future input size by monitoring received data and pending queues. The input size estimation is divided by the estimated operator capacity to give a value of its local activity level. To propagate local estimations to next operators, the estimated output is computed, relying on estimation of items processed and the operator selectivity factor. For child operators, this estimation is combined with their local estimation of input size to help the SPE reach its goal as stated above.

E. Quantification of scale-in/out modifications

We then identified operators requiring scale-in or scale-out. We need now to evaluate the appropriate parallelism degree of each operator requiring a modification. Let $deg_{j-1}(\mathcal{O}_i)$ the parallelism degree of $\mathcal{O}_i$ during the iteration F_j^i. Let $maxP_{\mathcal{O}_i}$, the maximal parallelism degree of $\mathcal{O}_i$, we consider that its appropriate parallelism degree is defined as follows:

$$deg_j(O_i) = \begin{cases} min(maxP_{\mathcal{O}_i}, deg_{j-1}(\mathcal{O}_i) + 1), \\ \text{if activity is 'high'} \\ min(maxP_{\mathcal{O}_i}, \lceil deg_{j-1}(\mathcal{O}_i) \times GAL_{F_j^i} \rceil), \\ \text{otherwise} \end{cases} \tag{11}$$

We distinguish the specific case where an operator has a high activity and an increasing input rate. Indeed, the value of $GAL_{F_j^i}$ is smaller than 1, but a scale-out is recommended (see Table I). In this case, we simply increment the parallelism degree of the operator by 1.

V. EVALUATION

A. Design and Implementation

1) Overview of Apache Storm: Apache Storm[2] is an open-source SPE, allowing users to define continuous queries

as graphs of operators, called *topologies*. Users define each operator in a high-level programming language such as Java, Python or Clojure.

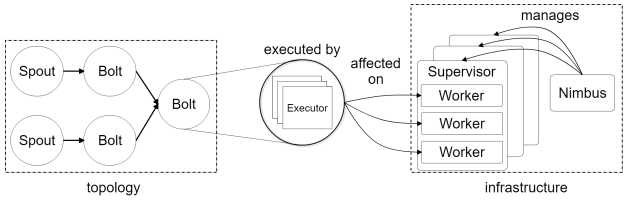


Fig. 5. Storm architecture

Operators, named *components* in Storm terminology, belong to one of two categories: *spouts* or *bolts*. A spout is a connector to a raw stream source and represents an entry of a topology. It distributes stream items to components to which it is connected and can process filtering operations if required. Bolts consume items from any component and compute a result for each item received (*stateless* bolt) or for a set of items (*stateful* bolt).

Each component is executed in parallel by *executors*. An executor is an instance of an operator. Each executor is assigned to a processing unit by the scheduler (see Figure 5). The number of executors for a given spout/bolt is revised at runtime only at user request[11].

Concerning the execution support, Storm relies on two types of processing nodes: *Nimbus* and *supervisor*. The Nimbus acts as a JobTracker for Hadoop [22]. As illustrated on Figure 5, each supervisor manages a pool of *workers*, *i.e.* processing units, and monitors executors assigned on them.

In this article, we consider the scheduling strategy presented in [12]. This strategy aims at optimizing resource usage of active supervisors and using only the necessary supervisors according to resource requirements defined by users.

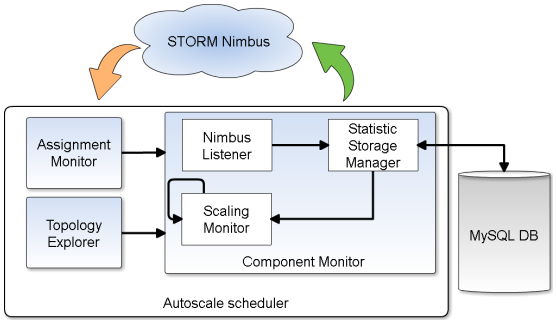


Fig. 6. AUTOSCALE architecture

2) Design of the AUTOSCALE *approach:* We have implemented AUTOSCALE over Storm 1.0.2. We have chosen this solution over other efficient SPEs such as Apache Spark Streaming[16] due to data management. Yet, Spark Streaming systematically groups items into batches, called *Resilient Distributed Datasets* (RDD). Nevertheless, if RDD sizes are large compared to incoming volumes, detection of congestion and over-consumption of resources is delayed. Thus, RDD size must be managed dynamically in addition to parallelism degrees.

AUTOSCALE is composed of three modules:

- *Component Monitor*: this module listens to the Nimbus, collects and post-processes statistics concerning each executor. It stores monitoring data in a database (see Figure 6). These data are periodically retrieved by a *Scaling Manager*, which then evaluates whether a modification is needed for each operator as described above.
- *Assignment Monitor*: this module collects information on executor assignments and supervisor states.
- *Topology Explorer*: for each submitted topology, this module builds static knowledge in order to explore topologies efficiently. For example, it builds lists of parents and children for each operator.

The AUTOSCALE scheduler includes the auto-parallelization strategy described above and the resource-aware scheduling strategy. It implements the *IScheduler* interface of the Storm API.

B. Experimental Protocol

Our test cluster is composed of 7 VMs. Each VM has at its disposal a dual-core CPU Intel(R) Xeon(R) E5-2620 running at 2.00GHz, 4Gb of RAM and 40Gb of hard disk space. A machine runs the Nimbus daemon and is dedicated to cluster coordination. Each supervisor manages 4 workers. On the Nimbus host, a MySQL database is also deployed in order to store historical data as illustrated above.

To validate our approach, we choose to show its impact on three elementary topologies: a linear, a star and a diamond topology. Each elementary topology is composed of two types of bolts: *intermediate* bolts with low latency and *sink* bolts with high latency.

In this section, we choose to present only some results relative to the linear ad a complex topology. However, more detailed results for diamond and star topologies are also available on our website[1]. Moreover, implementation, datasets and topologies can be downloaded for reproductibility.

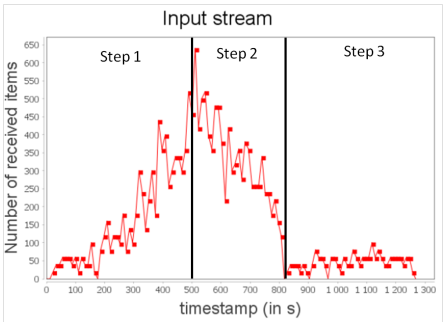


Fig. 7. 3-step stream

We built a 3-step synthetic stream with the following characteristics: 1) distribution with a small standard derivation 2)

[1]https://liris.cnrs.fr/ rkottoko/autoscale/v2/

significant increase and decrease in load. Indeed, as illustrated on Figure 7, input load is constant at a low rate. Load then increases progressively before stabilizing at a high rate. Finally, rates decrease markedly until it reaches the initial low rate. We also add small and irregular fluctuations in order to simulate fluctuations in a real stream. To comply with good Storm practices, we implemented the replay of out-of-time items.

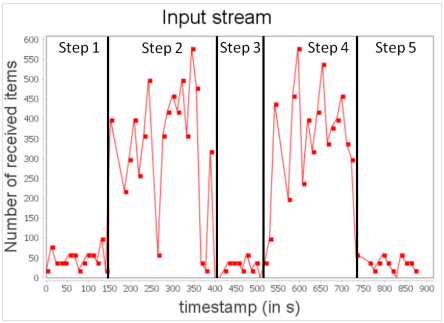


Fig. 8. 5-step stream

We then apply a 5-step stream with sudden input rate peaks (see Figure 8) to test the reactivity of our approach. This second stream moves from a low input to a very high one without a progressive transition as presented above. The input rate then decreases suddenly before increasing again. We summarize the main experimental parameters in Table II.

TABLE II
MAIN PARAMETERS

window size	60s
monitoring frequency	10s
θ_{min}	0.3
θ_{max}	0.8
processing timeout	30s
combine strategy	max

C. Results

1) 3-step stream and linear topology: We compare AUTOSCALE to the native scheduler according to two configurations. We summarize experimental configurations for the linear topology in Table III:

TABLE III
PARAMETERS OF OPERATORS FOR LINEAR TOPOLOGY

	intermediate	sink
average latency	2ms	80ms
min degree	1	1
expert degree	1	8
max degree	8	8
CPU load	20.0	80.0
memory load	256Mb	512Mb

With the configuration *ConfMin*, the initial number of executors per bolt corresponds to minimal degrees (see Table III). Intuitively, the configuration *ConfMin* is adapted to small incoming loads but cannot handle large ones. With the configuration *ConfExpt*, initial numbers of executors correspond to expert degrees (see Table III). Expert degrees have been chosen with full knowledge of stream variation and operator latencies. This configuration can in fact handle maximal loads without wasting resources.

For each configuration, we measured the global latency of the topology (configuration performance) and the number of dephased items (results quality). Concerning system reactivity and the usage of resources, we observed parallelism degrees of each bolt.

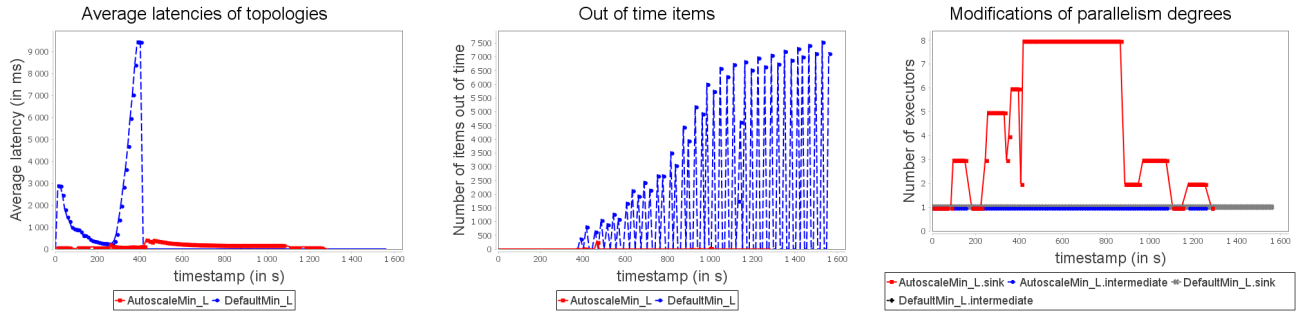
With ConfMin, we observe that the incoming load cannot be handled, thus leading to the complete congestion of the topology. Indeed, the topology is not able to process items completely. As soon as congestion occurs, new items emitted by the spout are dephased and replayed indefinitely until a user intervenes (see Figure 9a). On the contrary, our auto-parallelization strategy increases dynamically and automatically the parallelism degree of critical operators in order to adjust their capacities to future incoming loads. When the stream rate decreases, the parallelism degree decreases accordingly. It also prevents overusing resources that are no longer necessary.

With ConfExpt (see Figure 9b), we start with a configuration able to handle large loads. Nevertheless, this configuration overuses resources when the stream rate is low. It corresponds to the start and the end of the synthetic stream. Our auto-parallelization strategy reduces the parallelism degree when operators do not need large capacities. In this case, just as with ConfMin, the parallelism degree is adapted dynamically. AUTOSCALE then achieves equivalent performance with approximately 37.5% less CPU and memory resources. The significant increase in topology latency with AUTOSCALE is due to a scale-in from three to one supervisor, which re-routes multiple items and implies this significant overhead.

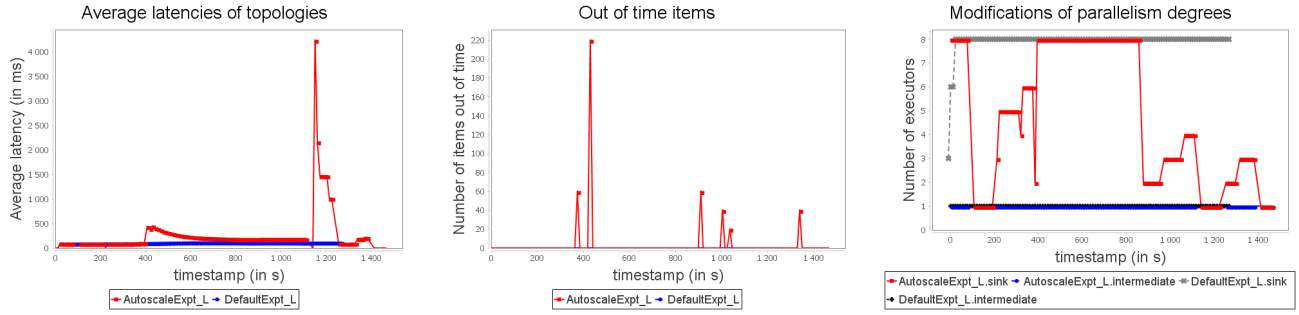
2) 5-step stream and linear topology: We can see on Figure 10, that with ConfMin, Storm is unable to handle the sudden increase in input rate and that topology is completely congested. Even the decrease in input rate is not enough to restore normal operator activity. Indeed, due to replay of out-of-time items more and more emissions are carried out by the spout, with the result that pending queues remain full. On the contrary, the AUTOSCALE approach reacts in multiple stages to adapt the capacity of each operator to fluctuations in input rate. Even the intermediate bolt, which has a very low latency, performs a scale-out as a precaution thanks to the global context. As a result of this adaptation, operators can consume their respective pending queues fast enough to benefit from the decrease in input rate. AUTOSCALE then adapts dynamically the parallelism degree. Finally, a reconfiguration is performed as soon as a new peak appears.

3) 3-step stream and advertising topology: Finally, we test our approach on an advertising topology mainly inspired from a topology used in [12] and available on Github[2] to validate

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



(a) Comparison between Storm (Default) and AUTOSCALE for the Linear topology with *ConfMin* and a 3-step stream.



(b) Comparison between Storm (Default) and AUTOSCALE for the Linear topology with *ConfExpt* and a 3-step stream.

Fig. 9. Experimental results for the Linear topology

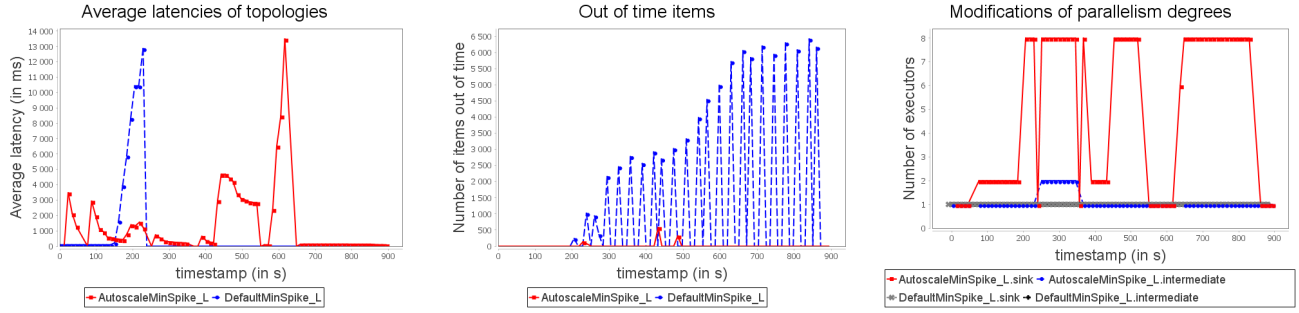


Fig. 10. Comparison between Storm (Default) and AUTOSCALE for the Linear topology with *ConfMin* and 5-step stream.

our approach. We essentially modify the source to be able to reproduce the same stream with different configurations and add two operators (ip projection and ip processor) to obtain a complex topology. Moreover, we apply the input stream variation illustrated on Figure 7.

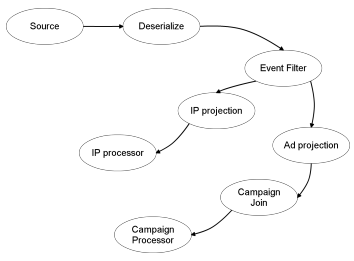
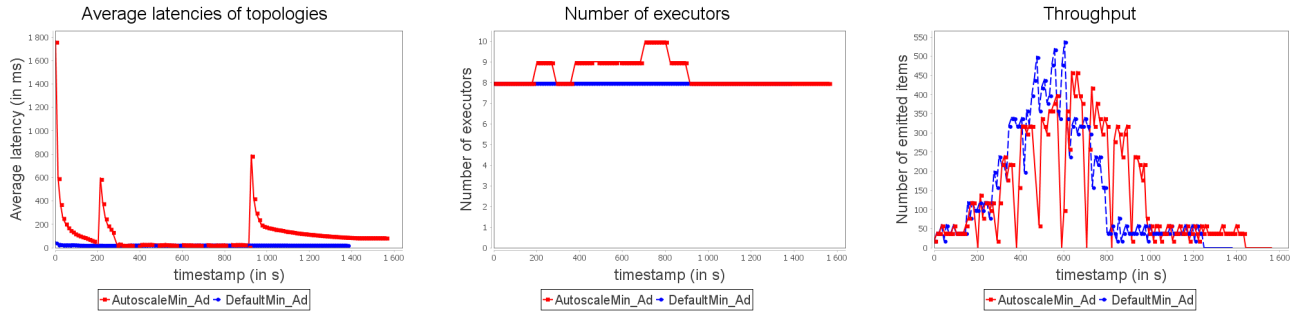


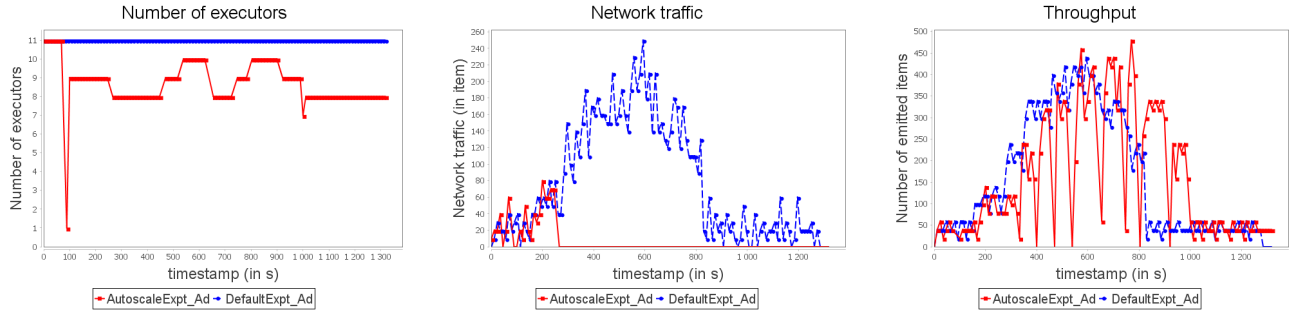
Fig. 11. Advertising topology for stream benchmarking

This topology takes as input, logs representing an event linked to an advertisement on a web page. Each log is first deserialized before being transmitted to an event filter. Two projection operators receive items from this filter, one looking for user IP addresses and the other for information on the ad. A join with a static dataset is performed to link the ad to a promotion campaign. Finally, IP and campaign processors increase users and campaign counts to update a remote database. The main interest of this topology is the significant selectivity of a filter operator (see Figure 11), as this implies that a large increase in input rate will have a minor impact on final operators even if they have large latencies in comparison with other operators.

We observe that even if the topology is not congested, the AUTOSCALE approach performs some scale-outs in order to adapt operator capacity to their respective input rates as illustrated on Figure 12a. This is due to the *combine* strategy, which takes into account the maximum between local and



(a) Comparison between Storm (Default) and AUTOSCALE for the Advertising topology with *ConfMin* and a 3-step stream.



(b) Comparison between Storm (Default) and AUTOSCALE for the Advertising topology with *ConfExpt* and a 3-step stream.

Fig. 12. Experimental results for the Advertising topology

global estimations as the globally consistent one. Therefore, when a slow operator begins to accumulate some items on its pending queue, the AUTOSCALE approach performs a scale-out to avoid congestion. Nevertheless, AUTOSCALE performs a similar throughput even if there are some unnecessary reconfigurations in one case. We can then estimate overheads induced by AUTOSCALE to 12% in comparison to actual needs in terms of CPU and memory requirements.

If a user bases his/her choice of parallelism degree exclusively on latencies, he/she will start the topology with some unnecessary executors (see Figure 12b). The AUTOSCALE approach then performs scale-in to fit capacities of operators to their respective processing needs. It is important to notice that the dynamic adaptation made by AUTOSCALE, combined with the scheduler, allows all treatments to be collected on a single supervisor. With AUTOSCALE, Storm is then able to handle biggest amount of data without generating network traffic, which is a large overhead factor, as explained in [10], and using 50% less resources.

VI. Conclusion

We suggest an approach that adapts dynamically and automatically the parallelism degree of operators according to stream rate fluctuations. Conducted experiments show that this approach limits operator congestion. Indeed, even a resource-aware strategy cannot prevent operator congestion if the issue is not exclusively linked to operator placement. We highlight through different test cases that an auto-parallelization strategy limits operator congestion significantly by adapting automatically and dynamically the parallelism degree of operators. Moreover, when compared to a configuration aiming at absorbing input rate peaks, AUTOSCALE allows the SPE to deliver similar performance with up to 37% less CPU and memory resources. Also, a dynamic change in the parallelism degree of operators significantly decrease network traffic when the input stream rate is low. When applied to a topology with selective operators, experiments show an overhead that does not exceed 12% of CPU and memory resources even with a strategy favoring scale-out as soon as possible. Nevertheless, the reactivity of AUTOSCALE is limited by monitoring configuration. This is because, if incoming load increases massively between consecutive timestamps, AUTOSCALE may not re-configure the system within an acceptable duration. Yet, AUTOSCALE periodically analyzes historical data before taking decisions. Thus, addition of monitoring of CPU and RAM resources is an effective solution for dealing with sudden operator congestion. Our future works deal with that limitation and the management of limited resources according to processing needs.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *In CIDR*, 2005, pp. 277–289.

[2] "Storm." [Online]. Available: https://storm.apache.org/

[3] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, Dec 2010, pp. 170–177.

[4] "Apache flink." [Online]. Available: https://flink.apache.org/

[5] "Google cloud dataflow." [Online]. Available: https://cloud.google.com/dataflow/

[6] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00778-004-0147-z

[7] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2528412

[8] K.-U. Sattler and F. Beier, "Towards elastic stream processing: Patterns and infrastructure." in *BD3@VLDB*, ser. CEUR Workshop Proceedings, G. Cormode, K. Yi, A. Deligiannakis, and M. N. Garofalakis, Eds., vol. 1018. CEUR-WS.org, 2013, pp. 49–54. [Online]. Available: http://dblp.uni-trier.de/db/conf/vldb/bd32013.html#SattlerB13

[9] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, 2013, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/2488222.2488267

[10] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014, pp. 535–544.

[11] Xu and G. Peng, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. IEEE International Conference on Cloud Engineering (IC2E), 2016*, 2016.

[12] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. H. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, 2015, pp. 149–161. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814808

[13] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[14] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.295

[15] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 318–321. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611314

[16] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: A fault-tolerant model for scalable stream processing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-259, Dec 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.html

[17] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, 2015, pp. 80–91. [Online]. Available: http://doi.acm.org/10.1145/2675743.2771827

[18] "Amazon ec2." [Online]. Available: https://aws.amazon.com/ec2/

[19] R. Das, G. Tesauro, and W. E. Walsh, "Model-based and model-free approaches to autonomic resource allocation," IBM Research Report, Tech. Rep. RC23802, Nov 2005. [Online]. Available: http://domino.watson.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/f5e3b7f574b24bad852570c1005e35a9!OpenDocument&Highlight=0,tesauro

[20] A. Senderovich, M. Weidlich, A. Gal, and A. Mandelbaum, *Queue Mining – Predicting Delays in Service Processes*. Cham: Springer International Publishing, 2014, pp. 42–57. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07881-6_4

[21] G. Box, *Box and Jenkins: Time Series Analysis, Forecasting and Control*. London: Palgrave Macmillan UK, 2013, pp. 161–215. [Online]. Available: http://dx.doi.org/10.1057/9781137291264_6

[22] "Hadoop." [Online]. Available: http://hadoop.apache.org/