Elastic-PPQ: a Two-level Autonomic System for Spatial Preference Query Processing over Dynamic Data Streams

Gabriele Mencagli*, Massimo Torquati and Marco Danelutto

Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127 Pisa, Italy

Abstract

Paradigms like *Internet of Things* and the most recent *Internet of Everything* are shifting the attention towards systems able to process unbounded sequences of items in the form of data streams. In the real world, data streams may be highly variable, exhibiting *burstiness* in the arrival rate and *non-stationarities* such as trends and cyclic behaviors. Furthermore, input items may be not ordered according to timestamps. This raises the complexity of stream processing systems, which must support elastic resource management and autonomic QoS control through sophisticated strategies and run-time mechanisms. In this paper we present `Elastic-PPQ`, a system for processing spatial preference queries over dynamic data streams. The key aspect of the system design is the existence of two adaptation levels handling workload variations at different time-scales. To address fast time-scale variations we design a fine regulatory mechanism of load balancing supported by a control-theoretic approach. The logic of the second adaptation level, targeting slower time-scale variations, is incorporated in a *Fuzzy Logic Controller* that makes scale in/out decisions of the system parallelism degree. The approach has been successfully evaluated under synthetic and real-world datasets.

Keywords: Data Stream Processing, Sliding-Window Queries, Elasticity, Burstiness, Out-of-order Data Streams, Intel Knights Landing

1. Introduction

In our ever-more connected world we are assisting to an unprecedented diffusion of systems and devices able to generate massive streams of transient data transmitted at great velocity. An urgent challenge is to design computing and communication infrastructures supporting real-time processing of data streams in order to extract complex analytics for decision making [1].

The *Data Stream Processing* paradigm [2] has been proposed to cope with this issue. It consists in a programming model where applications are written as data-flow graphs of logic entities called *operators*, each one applying transformations on the input data and deployed on parallel and possibly distributed environments.

Real-world data streams may exhibit dynamic characteristics like hysteric data rates with abrupt surges and out-of-order arrivals, where stream elements (called *tuples*) may not be gathered in increasing order of timestamps. Due to such erratic nature, *a-priori* capacity planning and management of the resources needed to execute streaming applications is not always practical [3], and this makes difficult to maintain the desired *Quality of Service* (QoS) while achieving high resource utilization efficiency.

To circumvent this issue, some papers have investigated the problem of enhancing stream processing systems with elasticity mechanisms [4, 5, 6, 7]. The term *elasticity* has been adopted in the field of Cloud Computing [8, 9] to indicate the ability to change the number and the size of virtual machines to adapt to the workload level. Recently, the same term has been used in the data stream processing literature [3, 4, 5] to indicate systems where the *parallelism level* of some operators (i.e. the number of replicas of the same operator working on distinct input data in parallel) can be dynamically modified to optimize the application throughput in response to variations in the arrival rate.

Most of the past elastic approaches have been experimented assuming workload variations in the form of deterministic trends and non-stationarities observed at time-scales of tens of seconds [3] or even minutes/hours [9]. The work in [10] demonstrated that in elastic Clouds the presence of fast variabilities such as *burstiness* may be detrimental both for the system QoS and for the resource utilization rate. This negative effect also occurs in stream processing systems, where phases with arrivals occurring in clusters (temporal burstiness) are likely to generate a growing computational burden to the system.

Furthermore, this problem is exacerbated by the fact that scheduling strategies may generate load imbalance in

*Corresponding author, Phone: +39-050-221-3132
Email addresses: `{mencagli, torquati, danelutto}@di.unipi.it` (Gabriele Mencagli*, Massimo Torquati and Marco Danelutto)

case of bursty streams. In fact, although a bottleneck operator can be replicated to increase throughput, for stateful operators input tuples must be scheduled to the replicas by preserving the computation semantics. As an example, all the tuples whose timestamp is within a certain temporal range might need to be scheduled to the same replica. In this scenario scaling only of the number of replicas may be ineffective because load imbalance may prevent additional resources to be efficiently utilized.

In this paper we study these problems for *sliding-window preference queries*, a class of important queries used in multi-criteria decision making [11]. We propose `Elastic-PPQ` (Elastic Parallel Preference Queries), a parallel *autonomic* system for executing preference queries on multicores. To provide autonomic functionality the system is able to continuously monitor its achieved throughput and to take corrective actions. Since workload variabilities happen at different time-scales, we design two adaptation levels synergically interrelated. The innermost level neutralizes burstiness at time-scales of tens/hundreds of milliseconds, and consists in adaptive scheduling strategies that dynamically change the way to assign inputs to the parallel replicas in order to adjust load balancing just enough to absorb fast traffic surges. Variabilities at slower time-scales are handled by the outermost level of adaptation that triggers elasticity mechanisms able to change the parallelism level of the system. Since the performance achieved by `Elastic-PPQ` depends on the behavior of several of its components whose performances are strongly interrelated, the definition of a precise mathematical model of the QoS is not an easy task. For this reason we decided to design the logic of the outermost level in order to incorporate heuristics derived from the human knowledge and the "rule of thumb" experience. To do that we use the *Fuzzy Logic Control* paradigm [12], a powerful approach to design model-free controllers suitable for systems with complex dynamics [13].

This paper builds on our previous work published in [14]. The novel contributions are listed below:

- in this work, the burst-tolerant scheduling strategies previously published in [14] are synergically used with elasticity mechanisms able to scale the utilized resources dynamically. The fuzzy-logic adaptation level is described in this paper for the first time, and represents an original application of the Fuzzy Logic Control methodology applied in the data stream processing field for the first time;
- the scheduling approach designed for the second part of the system (described in Sect. 3.2) is substantially enhanced with respect to the first version in [14]. The improvement consists in the definition of a new type of tasks that can be scheduled to extract more parallelism during bursty periods, and thus to allow achieving a high throughput with better resource utilization (up to 45% improvement);
- `Elastic-PPQ` is evaluated on datasets exhibiting burstiness generated with different traffic models, widely used in teletraffic engineering to perform stress tests. Furthermore, we develop an extensive evaluation on real datasets from an existing application.

This paper is organized as follows. Sect. 2 introduces the background and the motivation. Sect. 3 presents the strategies in the innermost level of adaptation and Sect. 4 describes the outermost level. Sect. 5 provides a wide set of experiments. Finally, Sect. 6 reviews similar work and Sect. 7 gives the conclusions.

2. Background and Motivation

In this section we introduce our parallel approach to sliding-window preference queries and the motivations behind the design of `Elastic-PPQ`.

2.1. Sliding-window preference queries

Spatial preference queries process d-dimensional input tuples (e.g., sensor readings) where $d > 0$ is the number of attributes per tuple. Their goal is to extract from the set of inputs the subset of the best tuples selected according to a certain preference criterion specific of the query to be executed. Fig. 1(left) shows an example with $d = 2$ in the case of the *skyline query* [15]. In this query a tuple t_1 is considered better than t_2 (t_1 *dominates* t_2) if $t_1.a_i \leq t_2.a_i$ for both the attributes and there exists at least one attribute a_j such that $t_1.a_j < t_2.a_j$. This relation is known as *Pareto dominance* [15] and the output is the subset of all the non-dominated tuples (called *skyline*).

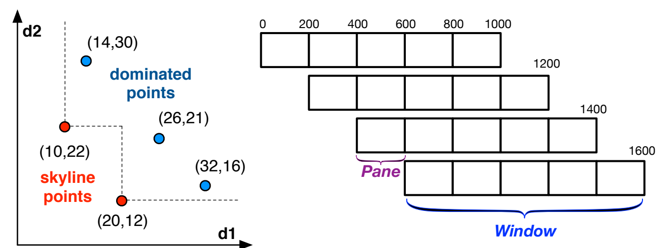


Figure 1: Example of skyline given a set of two-dimensional tuples (**left**), and consecutive sliding windows (**right**).

`Elastic-PPQ` executes preference queries according to the *sliding-window model* [2], where the best tuples are periodically computed by inspecting the inputs received in temporal windows of w time units that slide every $s < w$ time units, see Fig. 1(right). The output is a stream of skylines, one for each temporal window. For instance, with windows of $w = 1$ second that slide every 0.2 seconds the skyline query is applied over the tuples received in the last second by computing a new skyline every 0.2 seconds. To avoid recomputing the window skylines from sketch, we use the *pane-based model* [16, 17] that leverages the fact that consecutive windows have an overlapping region and so several tuples in common. Each window is divided into

non-overlapping ranges called *panes* of $L_p = GCD(w, s)$ time units. In the example each pane lasts $L_p = 0.2$ seconds (we have five panes per window).

The pane-based model consists in two phases. The *pane-level sub-query* (PLQ) processes input tuples by finding the best ones within each pane interval. The *window-level sub-query* (WLQ) computes the best tuples within each window by comparing the best tuples selected in the panes. Since the same pane is shared among multiple consecutive windows, this approach reduces the number of pair-wise tuple-to-tuple comparisons with respect to traditional sliding-window processing models [16, 18]. Furthermore, the model takes also advantage of the *incremental* computation of results. Pane results are incrementally computed by the PLQ phase every-time a tuple arrives by comparing it with the best tuples currently selected in the pane. Analogously, window results are updated in the WLQ phase as soon as a new pane result is available.

Fig. 2 describes the model in the case of the skyline query. The figure outlines the *pipeline parallelism* existing between the PLQ and WLQ phases that represent two *stages* connected in tandem. While the PLQ stage processes input tuples, the WLQ stage updates the window results using the results of the already computed panes.

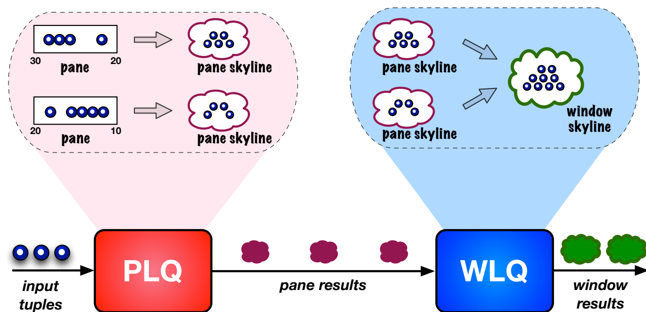


Figure 2: The PLQ stage computes the skyline of each pane, the WLQ stage computes the skyline of each window by merging the skylines of the panes.

2.2. Workload variabilities and solution overview

The arrival rate of data streaming applications may be affected by two variability factors that can have a direct, negative effect on the system throughput:

- *non-stationarities*: they consist in variations in the mean of the arrival rate like monotonous or step-like increasing/decreasing trends, flash-crowd effects, or periodic/cyclic variations [19];
- *burstiness*: it is a stochastic variability where the sequence of inter-arrival times tends to form clusters of closely time spaced arrivals. This effect can be originated by very dynamic sources that transmit intermittently, or introduced by network delays.

Non-stationarities are variabilities at a slower time-scale with respect to the one at which data arrive at the system [20]. In data centers and Clouds, non-stationarities happen at time-scales of minutes or even hours [21]. Slow time-scales instead are often considered in the order of few seconds in data stream processing applications receiving thousands of tuples per second [4, 5].

Burstiness may happen at multiple time-scales and is measured using second-order properties such as the index of dispersion or the Hurst parameter [22, 23, 10]. Fig. 3 illustrates three different traffic models (Poisson, Markov Modulated and Self-similar arrivals) all having the same average rate of $10K$ tuples/sec. The details about such models will be described later in this paper. In each row we plot on the left hand side the number of arrivals measured at a time-scale of 100 ms, which correspond to the red region in the plot on right hand side where we amplified the scale by a factor of 10. In the Poisson traffic burstiness disappears just with a fast time-scale of 100 ms, while in the other models it persists more or less intensively. Interestingly, the self-similar traffic has the characteristic to be *scale invariant* [19], i.e. the arrival process looks nearly the same at different time-scales, see Fig. 3(bottom).

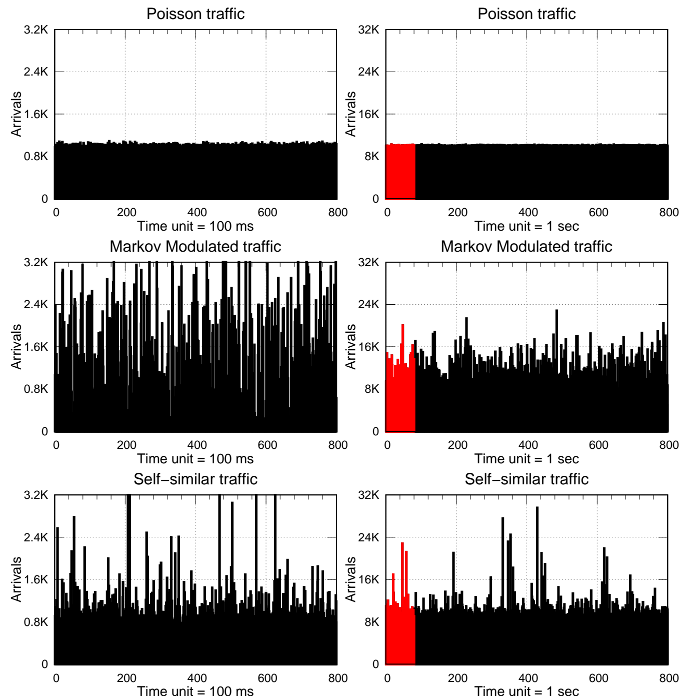


Figure 3: Burstiness at different time-scales: Poisson, Markov Modulated (fitted with an index of dispersion of $1K$) and Self-similar arrivals (with Hurst parameter of 0.75).

These variability factors must be addressed by adaptation mechanisms. An approach consists in the so-called *computational elasticity* [4, 5], which in streaming scenarios consists in the dynamic modification of the parallelism level of some application operators in response to workload fluctuations and changes in the resources availability. Apart from the ability to keep up with the workload, additional motivations in favor of elasticity are:

- *parallelism-level auto-tuning*: a sliding-window query can be run multiple times each receiving a data stream

with different temporal characteristics. Finding the right parallelism level for each run requires an intensive manual tuning which may be impractical;
- *sharing of cores and computing cycles*: elastic applications can automatically coexist with other running applications by adapting to the number of unused cores and to the available computing cycles. Such competition may also exist among different parts of the same application, e.g., the PLQ and WLQ stages in Fig. 2 can be internally parallel, and the selection of their parallelism level requires a proper coordination;
- *saving instantaneous power*: elasticity is beneficial in light of the importance of power efficiency. In fact, using more active replicas of an operator may increase the power demand, and so higher parallelism levels should be used when favorable trade-offs between power and performance can be achieved.

The strategy that monitors the system execution and triggers changes in the parallelism level is often carried out at regular time intervals [24]. If the interval is too long the system is slow to adapt to workload variations. If it is too short the system may not be fast enough to exploit the new parallelism level. In prior stream processing systems the used intervals are in the order of tens of seconds like in [4, 25]. Such granularity may be inappropriate to react to burstiness at fast time-scales like the one shown in the last two plots in the left hand side of Fig. 3.

Variabilities at different time-scales need to be handled by mechanisms with different responsiveness. In case of sliding-window queries, the scheduling strategies that distribute input data to the replicas play a decisive role in smoothing the impact of burstiness. Bursty periods may lead to uneven pane sizes in terms of number of tuples contained, and this may hamper load balancing. In `Elastic-PPQ` this problem is solved by an adaptive scheduling strategy able to identify large panes and to split them just enough to achieve the ideal instantaneous throughput. In addition, such strategies are used together with an overall controller that monitors the system execution, detects whether the used parallelism level needs to be decreased to improve resource utilization efficiency or increased to eliminate bottlenecks in the system.

In the next two sections we will study in detail these two adaptation levels.

3. Innermost Level of Adaptation

To increase the query throughput, the scheme in Fig. 2 is parallelized by making the PLQ and WLQ stages internally parallel by replicating their logic in multiple identical entities called *workers* [2]. Fig. 4 provides the illustration of the whole structure, with $n > 0$ workers in the first stage and $m > 0$ in the second one.

The *emitters* (denoted by `E`) are responsible for routing input data to the selected destination workers. *Collectors* (denoted by `C`) collect the results of the workers by eventually putting them in the right order.

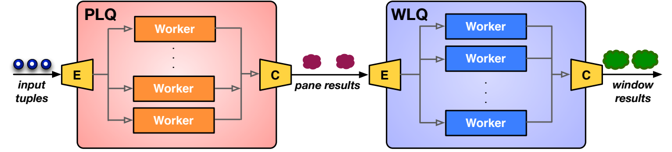


Figure 4: The structure of the system is a pipeline of two parallel stages (each featuring more identical workers).

To make the paper self-contained, we first recall the main features of the adaptive scheduling strategy of the PLQ stage, which has been published in [14]. Then, we will describe the scheduling performed in the second stage, which will be substantially enhanced with respect to the one originally described in the same paper.

3.1. PID-based scheduling in the PLQ stage

The scheduling is described in Alg. 1. It consists in two main parts: the management of out-of-order tuples through a punctuation mechanism, and the adaptive scheduling of tuples based on a pane splitting approach.

Algorithm 1: Adaptive splitting approach (PLQ)

Input: an input tuple t
Result: tuple t is dispatched to an activated PLQ worker
1: **procedure** PLQ_SCHEDULE(t)
2: ▷ Punctuation generation logic
3: **if** $t.t_s > t_{max}$ **then**
3.1: We set $t_{max} \leftarrow t.t_s$ and $K \leftarrow \max\{K, \overline{d}\}$, where $\overline{d}$ is equal to $\max_{t_i \in \mathcal{D}}\{t_{max} - t_i.t_s\}$
3.2: Set $\mathcal{D} \leftarrow \emptyset$, and a punctuation with timestamp $t_p \leftarrow \max\{t_p, t_{max} - K\}$ is sent to the activated workers
4: **else** $\mathcal{D} = \mathcal{D} \cup \{t\}$
5: ▷ Splitting threshold adaptation
6: **if** new sampling period τ **then**
6.1: From the statistics get the number of tuples scheduled to/processed by the workers during the last sampling period
6.2: Estimate the service rate $\mu(\tau)$ of the workers and the PLQ utilization factor $\rho(\tau)$
6.3: Update the *splitting threshold* $\theta(\tau)$ by using the PID control law (Fig. 5) with inputs $\rho(\tau)$ and setpoint $\overline{\rho}$
7: ▷ Scheduling of the tuple
8: **Let** p **be** the identifier of t's pane, i.e. $p = \lfloor t.t_s / L_p \rfloor$
9: **if** `Owner`[p] = `undefined` **then**
10: The `llw` becomes the owner of p and t is sent to it
11: **else**
12: **Let** w **be** `Owner`[p]
13: **if** `SentTuples`[p,w] $< \theta(\tau)$ **then** tuple t is sent to w
14: **else** the `llw` becomes the owner of p and t is sent to it

***K*-slack punctuations.** To handle out-of-order tuples `Elastic-PPQ` relies on a punctuation mechanism [26]. Input tuples are processed by the workers in the PLQ stage in the arrival order, and the PLQ emitter periodically transmits *punctuations* to them. A punctuation is a meta-tuple conveying a timestamp value. If the workers receive a punctuation with timestamp t_p, they are sure that no tuple with timestamp smaller than t_p can be received in

the future. To meet this constraint, the emitter eventually drops all the late tuples received after the punctuation transmission with timestamp smaller than t_p.

The precision of a punctuation mechanism is evaluated in terms of the number of dropped tuples. We adopt the *K-slack* algorithm [27] whose high precision has been demonstrated in past work [28]. The algorithm (at line 3 of Alg. 1) uses a variable t_{max} to store the highest timestamp seen in the stream history, and a variable K recording the maximum tuple delay. The idea is to generate a punctuation t_p when the first tuple with timestamp greater than $t_p + K$ has been received. So doing, all the panes $i = 0, 1, \dots$ such that $(i + 1) \cdot L_p \leq t_p$ can be considered "closed" and their results transmitted to the second stage.

It may be possible that no tuple is received within the temporal range of a pane. The PLQ emitter records the number of received tuples per pane, and in case of empty panes it dispatches a special `EMPTY` meta-tuple before the punctuation (not shown in Alg. 1 for brevity), whose role is to open an empty pane result in a destination worker. This aspect will be further discussed later in the paper.

A burst-tolerant scheduling. Tuples of different panes can be computed by different workers in parallel. However, in case of fast time-scale burstiness some "heavy" panes may contain more tuples than the others, and this may impair load balancing and thus the achieved throughput.

The solution described in [14] allows panes to be split. Let $\mathcal{P}_{i,j}$ the j-th partition of pane $\mathcal{P}_i$ and $\mathcal{R}_{i,j}$ its result. Suppose that tuple t is the first received tuple of $\mathcal{P}_i$. The emitter assigns t to currently least loaded worker (denoted by `llw`), i.e. the one having the smallest number of enqueued tuples (line 10 of Alg. 1). This worker, let say worker w, becomes the *owner* of the currently active partition of the pane. The emitter records the number of tuples of $\mathcal{P}_i$ transmitted to w. Until such number is lower than a *splitting threshold* θ, the tuples of $\mathcal{P}_i$ are scheduled to worker w that updates the result $\mathcal{R}_{i,w}$ (line 13). If the threshold is exceeded, a new worker becomes the owner that will receive the next tuples of the pane by creating a new result (line 14). Once received the punctuation closing the pane, the results of its partitions are sent to the WLQ stage. We define the *splitting factor* $\sigma_{avg} \geq 1$ as the average number of existing partitions per pane.

The choice of the threshold is critical. Low values lead to an aggressive splitting that may vanish the advantage of the pane-based approach while high thresholds may result in load imbalance. For this reason, the threshold value should be automatically adjusted based on the monitored throughput. Since this is a single-input single-output control problem (SISO), we adopt a Proportional-Integrative-Derivative regulator (called PID, see Fig. 5) to do this work because it represents a popular and well-established solution for SISO control loops [29]. The threshold during the sample τ is computed as follows: $\theta(\tau) = \alpha(\tau) \cdot \theta_{base}$, where $\alpha > 0$ is an adaptation parameter and θ_{base} is a statistic base threshold set to the average size of the last closed partitions plus the standard deviation. The PID computes an adjustment $\Delta\alpha(\tau)$ of the adaption parameter such that $\alpha(\tau) = \alpha(\tau - 1) + \Delta\alpha(\tau)$.

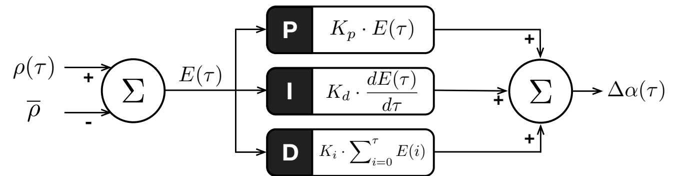


Figure 5: The PID regulator automatically changes the splitting threshold based on the measured PLQ utilization factor.

The input of the PID is the *utilization factor* $\rho(\tau) > 0$, an indicator computed as the ratio of the number of received tuples to the number of tuples that the stage is able to serve during the sampling period. If $\rho < 1$ the stage is not a bottleneck and the threshold can be increased to split less the panes. Otherwise, panes must be divided more to improve load balancing. The error $E(\tau)$ is the deviation between the utilization factor and a desired value $\overline{\rho}$ slightly smaller than one to optimize throughput with high resource utilization efficiency, e.g., $\overline{\rho} = 0.9$.

To determine the utilization factor during a sampling period τ, we need to estimate the average number of tuples that a worker would be able to process during the period by assuming it is never idle. As the paper [14] explains in detail, this can be estimated from the monitoring data, by measuring for each tuple computed during the sampling period τ its processing time and by taking the average processing time among all the tuples computed by the workers. Then, the PLQ utilization factor $\rho(\tau)$ is computed as the weighted average of the utilization factors $\rho_i(\tau) = \lambda_i(\tau)/\mu(\tau)$ of the workers, where $\lambda_i(\tau)$ is the number of tuples scheduled to the i-th worker, $\mu(\tau)$ is the worker service rate and the weights are the distribution probabilities λ_i/λ with $\lambda = \sum_i \lambda_i$.

Our approach has some interesting properties. First, it is not necessary that the load is always perfectly balanced among the $n > 0$ workers. If a worker receives a fraction greater than $1/n$ of the arrivals, this is acceptable as long as the service rate is high enough to prevent it from becoming a bottleneck. If the PID is unable to achieve $\rho < 1$ the scheduling behaves as follows:

Proposition 1. If the PID is unable to lower the utilization factor close to the setpoint $\overline{\rho}$, it will split the panes as much as possible, i.e. $\sigma_{avg} \approx n$.

Proof. The PID tries to minimize the utilization factor by choosing the distribution frequencies $\lambda_1/\lambda, \dots, \lambda_n/\lambda$. In the notation we omit the sampling period variable τ for simplicity. The optimization problem is the following:

$$\min_{\lambda_1,\dots,\lambda_n} \rho(\lambda_1, \dots, \lambda_n) = \sum_{i=1}^{n} \left(\frac{\lambda_i}{\lambda} \cdot \rho_i \right) = \frac{1}{\lambda \cdot \mu} \cdot \sum_{i=1}^{n} \lambda_i^2$$

The problem is constrained by the following condition

$\sum_{i=1}^{n} \lambda_i = \lambda$. According to the well-known Cauchy-Schwarz inequality we can write:

$$\sum_{i=1}^{n} \lambda^2 \geq \frac{1}{n} \cdot \left(\sum_{i=1}^{n} \lambda_i \right)^2 = \frac{1}{n} \cdot \lambda^2$$

This result can be used to derive a lower bound of the utilization factor, which is given by:

$$\rho = \frac{1}{\lambda \cdot \mu} \cdot \sum_{i=1}^{n} \lambda_i^2 \geq \frac{1}{\lambda \cdot \mu} \cdot \frac{\lambda^2}{n} = \frac{\lambda}{\mu \cdot n}$$

This lower bound can be obtained when $\lambda_i = \lambda/n$, i.e. when the splitting threshold is very low and the emitter evenly distributes to the workers the input tuples received during the sampling period. This means that the splitting factor turns out to be maximum, i.e. $\sigma_{avg} \approx n$. □

Counting the pane partitions. The WLQ stage must be able to determine when all the partitions of a pane have been received. To do that, each PLQ result is associated with the identifier of the pane and the counter of the partitions of that pane. Each time a new partition is created, all the workers already having a partition of that pane must update their counter with the new value, which is notified with an `UPDATE_CNT` meta-tuple sent by the emitter (this extends the actions at line 14 in Alg. 1).

3.2. Feedback-based scheduling in the WLQ stage

Also in the WLQ stage the scheduling must be designed in order to tolerate micro-congestion episodes happening at fast time-scales. The general idea of the scheduler (described in Alg. 2) is to spawn different types of tasks that the WLQ workers have to compute.

The emitter maintains an internal data structure called *window directory* (`WD`) storing a *descriptor* for each window. Descriptors are periodically purged when window results are finalized in order to save memory. The descriptor of a window $\mathcal{W}_k$ contains a queue (`pBuffer`) of pane partition results that belong to that window and that still have to be processed. We denote by Φ_i the set of the identifiers[1] of the windows that contain the pane $\mathcal{P}_i$.

The WLQ emitter adds each result $\mathcal{R}_{i,j}$ to the `pBuffers` of the windows that contain $\mathcal{P}_i$ (a pointer/reference is put in those buffers). For each of these windows a `WIN_UPDATE` task can be spawned. A task of this type updates the window result by taking into account the tuples in $\mathcal{R}_{i,j}$.

Only tasks of different windows can be executed in parallel. In fact, two tasks of the same window will update the same window result and this must be done in mutual exclusion. This constraint is enforced by using a `busy` flag in the window descriptor, set to true by the emitter if a `WIN_UPDATE` task of that window is still in execution

---

Algorithm 2: Feedback-based scheduling (WLQ)

Input: a result of a pane partition $\mathcal{R}_{i,j}$ or a feedback message `Fb`
Result: new tasks are scheduled to idle workers

1: **procedure** WLQ_SCHEDULE($\mathcal{R}_{i,j}$ or `Fb`)
2: ▷ Case 1: the input is a new result $\mathcal{R}_{i,j}$
3: **if** input is $\mathcal{R}_{i,j}$ **then**
4: **for each** $k \in \Phi_i$ **do** ▷ pane $\mathcal{P}_i$ belongs to window $\mathcal{W}_k$
5: Add $\mathcal{R}_{i,j}$ to `WD[k].pBuffer`
6: $m_{idle} \leftarrow$ `getNumIdleWorkers()`
7: $\mathcal{T} \leftarrow$ `FindTasks(`m_{idle}`)`
8: **for each** $tk \in \mathcal{T}$ **do**
8.1: If tk is a `WIN_UPDATE` task of window tk.win, the `busy` flag in the corresponding descriptor is set to `true`. If tk is the last task to execute of window tk.win, the `finalized` flag in tk is set to `true`
8.2: The task tk is sent to an idle worker
9: ▷ Case 2: the input is a feedback message
10: **if** input is `Fb` **then**
11: **if** `Fb.type` = `MERGE` **then** add `Fb.`$\mathcal{R}$ to `WD[Fb.win].pBuffer`
12: **else** set the flag `busy` in the descriptor of `Fb.win` to `false`
13: Mark the worker w sending the `Fb` message as idle
14: **if** worker w is still activated **then**
15: $\mathcal{T} \leftarrow$ `FindTasks(1)`
16: **if** $\mathcal{T} = \{tk\}$ **then**
16.1: If tk is a `WIN_UPDATE` task of window tk.win, the `busy` flag in the corresponding descriptor is set to `true`. If tk is the last task to execute of window tk.win, the `finalized` flag in tk is set to `true`
16.2: Task tk is sent to worker w
17: **else** worker w becomes idle

---

in a worker. The emitter never dispatches a `WIN_UPDATE` task for a window with the busy flag set to true. In this way it may be possible that although some pane results are enqueued in the internal `pBuffers`, some workers are actually idle because no further tasks can be scheduled.

To find further parallelism opportunities, we enhance the scheduling with `MERGE` tasks. A task of this type consists in a pair of two results of pane partitions. Its execution produces a single result containing the best tuples among the ones contained in the two results. `MERGE` tasks can be executed to do useful work when all the open windows are busy but there are still idle workers to utilize.

To utilize the workers more effectively, the emitter avoids sending tasks to workers that are running previously scheduled tasks. To do that the emitter: *i)* marks each worker with a `ready` flag set to true if the worker is idle, false otherwise; *ii)* after the computation of a task, workers send special messages called *feedbacks* that notify the emitter when they are ready to compute again. The ready flag of a worker is set to false by the emitter when a task is scheduled to it, and the emitter resets it to true when it receives the feedback message from that worker.

As shown in Alg. 2, at the arrival of a result from the PLQ stage the WLQ emitter updates the `pBuffer` of the windows and tries to schedule at most $m_{idle} \geq 0$ tasks (line 7), where m_{idle} is the number of idle WLQ workers (line 6). Alg. 3 first tries to find `WIN_UPDATE` tasks of different windows, otherwise it looks for `MERGE` tasks.

Each feedback message incorporates information related

[1] The set Φ_i can be statically defined for most of the sliding window semantics including the time-based sliding windows studied in this paper. See [18, 16] for further details.

Algorithm 3: `FindTasks`

Input: maximum number $m_{idle} \geq 0$ of tasks that must be found
Result: a set of tasks $\mathcal{T}$

1: **procedure** FindTasks(m_{idle})
2: ▷ Looking for `WIN_UPDATE` tasks
3: **for each** $\mathcal{W}_k \in$ `WD` **do**
4: **if** $\mathcal{T}$.size $= m_{idle}$ **then return** $\mathcal{T}$
5: **if** `WD[k].busy` $\neq$ `true` **AND** `WD[k].pBuffer` $\neq \emptyset$ **then**
5.1: Extract a result from the `pBuffer` and add a new `WIN_UPDATE` task to $\mathcal{T}$
6: ▷ Looking for `MERGE` tasks
7: **for each** $\mathcal{W}_k \in$ `WD` **do**
8: **if** $\mathcal{T}$.size $= m_{idle}$ **then return** $\mathcal{T}$
9: **if** `WD[k].pBuffer` ≥ 2 **then**
9.1: Extract two results from the `pBuffer` and add a new `MERGE` task to $\mathcal{T}$
10: **return** $\mathcal{T}$

to the completed task, such as its type and the involved window. If it was a `WIN_UPDATE` task, the emitter clears the busy flag of the window (line 12 of Alg. 2). If it was a `MERGE` task, the message contains the result of the task which is appended to the `pBuffer` of the window (line 11). Then, the emitter tries to schedule a new task to the same worker (line 16), otherwise its ready flag is set to true.

4. Outermost Level of Adaptation

We add to the adaptive scheduling strategies described in the previous section an overall controller in charge of dynamically adjusting the number of workers available in the two stages. In the description we will assume that each worker entity is executed by a dedicated thread in the `Elastic-PPQ` run-time system. Thereby, we will use the terms parallelism level and thread level interchangeably.

4.1. Elasticity mechanisms

The complete design of `Elastic-PPQ` is illustrated in Fig. 6. The parallel structure in Fig. 4 is extended by introducing a *Controller* entity responsible for periodically evaluating the logic that triggers *reconfigurations* of the number of activated workers. We first describe the reconfiguration mechanisms in general. Then, we show in detail how they are used in the two stages.

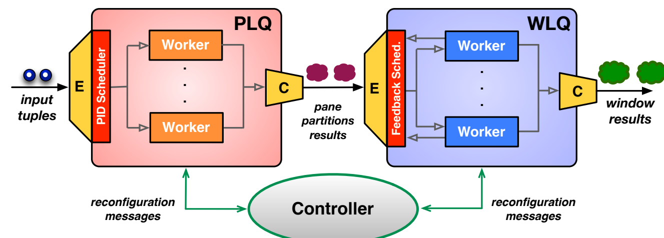


Figure 6: Final design of the system: innermost level of adaptation encapsulated in the emitter entities (E), and the outermost level implemented by a *controller*.

Reconfiguration messages are transmitted from the controller to the emitters and can be of two types: `REMOVE` and `ADD`. A `REMOVE` message deactivates a subset of the workers used by the stage. Since thread creation may come with considerable overhead in most of the operating systems, threads once created are never destroyed but the system tries to recycle them. After the arrival of a `REMOVE` message, the emitter stops sending new tuples/tasks to the workers to be deactivated and sends a `TURNOFF` message to them with the effect to put their threads to sleep.

The second type of reconfigurations are `ADD` messages. Before sending a message of this type, the controller determines whether there exist enough sleeping threads for executing the new workers. If this is not the case, new threads are spawned. The system uses a global maximum thread level $N > 0$ by default equal to the number of physical cores available[2]. After receiving the `ADD` message, the emitter updates its scheduling with the new list of activated workers and wakes up all the workers that have to be re-activated.

Elasticity in the PLQ stage. In the first stage an increase in the parallelism level requires that the emitter simply starts to schedule tuples to the new workers. After a `REMOVE` message instead, the PLQ emitter executes the actions in Alg. 4. The removed workers are marked as deactivated by the emitter, and a `TURNOFF` message is sent to them. For all the open panes (i.e. panes with at least one received tuple and that are not closed by a punctuation yet), the emitter checks whether their owner is still activated or not. If not, the least loaded worker (denoted by w in Alg. 4) is designated as the new owner. There are two possible cases: *i)* w has already a partition of the pane and no further action is needed; *ii)* w does not have a partition of the pane, and the emitter transmits to w an `EMPTY` meta-tuple opening a result of that pane with the counter set to the updated number of existing partitions.

Algorithm 4: Worker removal in the PLQ stage

Input: set of identifiers $\mathcal{Z}$ of the PLQ workers to deactivate
Result: the workers with identifiers in $\mathcal{Z}$ are deactivated

1: **procedure** PLQ_Removal($\mathcal{Z}$)
2: **for each** $w \in \mathcal{Z}$ **do**
3: send a `TURNOFF` message to w and mark it as deactivated
4: **for each** not closed panes p **do**
5: **if** `Owner[`p`]` $\in \mathcal{Z}$ **then**
6: **Let** w **be** the current `llw` worker
7: **if** w already has a partition of p **then**
8: `Owner[`p`]` $\leftarrow w$
9: **else**
10: `Owner[`p`]` $\leftarrow w$ and an `EMPTY` meta-tuple is sent to w

A PLQ worker receiving a `TURNOFF` message must forcibly close and transmit to the WLQ stage all the partition results that is currently holding. A subtle problem is that such panes can be further split by the PID-based scheduling, and thus the number of partitions may change in the future. The WLQ stage must be able to determine

[2] Alternatively, the maximum thread level can be set to the available number of virtual cores (Simultaneous Multithreading).

when all the partitions of a pane have been received, and the values of the counters must always be coherent to do that. For this reason, the counters of all the forcibly closed results must be reset to a special value (i.e. zero) before being transmitted. In this way the WLQ stage is able to correctly count the number of results per pane, since all the non-zero counters will have the same value equal to the final number of existing partitions of the pane.

Elasticity in the WLQ stage. In the second stage reconfigurations are handled easily since workers do not maintain any internal data structure across the processing of different input tasks. In case of an `ADD` reconfiguration, the emitter only updates the list of activated workers that are initially idle (see Alg. 2). A `REMOVAL` message is handled as follows:

- the emitter stops to schedule tasks to the deactivated workers and sends a `TURNOFF` message to them;
- once received a `TURNOFF` message, the workers are deactivated and the corresponding threads go to sleep.

4.2. Fuzzy logic controller

Autonomic strategies map system behavioral conditions onto reconfiguration actions. Our system presents interrelated dynamics among its sub-parts. For example, if the first stage is a bottleneck while the second one has a thread level just sufficient to process the tasks at the current speed, an increase in the parallelism of the first stage will likely reflect in a higher input pressure to the second stage. Therefore, in addition to increase the thread level in the first stage the strategy might try to increase in anticipation the parallelism level of the second stage.

While conventional Control Theory methods can be easily used to automatize the adjustment of control variables in small parts of the system (like scheduling parts), the application of such methods to large systems is more difficult. In fact, standard techniques like Model Predictive Controllers [29] need accurate mathematical models that are complex to be defined for systems like ours with multiple components featuring interdependent dynamics. For this reason we adopt the methodology of *Fuzzy Logic Controllers* [12, 13] (shortly, FLC). Instead of being based on a system model, FLCs use a set of logic rules that provide a linguistic description of the interactions existing between the different system parts and allow an inference process that extracts the actions to be taken.

Since it has never been applied to stream processing systems in the past, the use of the FLC methodology deserves to be explained in detail by focusing on its three main phases: *fuzzification*, *inference rules* and *defuzzification*.

Fuzzification. Our FLC has three crisp inputs whose values are monitored by the run-time system: the PLQ utilization factor $\rho_1 \in [0, +\infty)$, the splitting factor $\sigma_{avg} \in [0, N]$, and the WLQ utilization factor $\rho_2 \in [0, +\infty)$ (this last computed similarly to what we did in Sect. 3.1 for ρ_1). Each crisp input is associated with a *linguistic variable* with terms expressed in natural language. The first variable `RHO_PLQ` corresponds to the ρ_1 crisp input and takes three possible *linguistic terms*: *slow* means that the first stage is currently too slow with respect to its arrival rate; *fast* means that it is unnecessarily fast; finally, *acceptable* means that its processing speed is very close to its input speed. The variable `RHO_WLQ` corresponding to the ρ_2 crisp input has the same terms and fuzzification of `RHO_PLQ` because the two crisp inputs ρ_1 and ρ_2 take their values in the same domain and have the same definition (e.g., values greater than one identify a congestion scenario).

Each term L is a fuzzy set with a *membership function* $\mu_L(x) \in \{0, 1\}$ assigning to each value x a grade of membership. Fig. 7(left) shows the fuzzy sets adopted for the `RHO_PLQ` variable, analogous sets are defined for `RHO_WLQ`. We chose trapezoidal-shaped membership functions for the *fast* and *slow* terms, while *acceptable* has a triangular-shaped membership. Such shapes are very common in fuzzy control. Non-linear functions like the Gaussian are more complex, and we experienced no appreciable advantage in using them. Each function has a set of parameters chosen according to the following principles:

- if the stage speed is at least two times higher than its arrival rate (i.e. $\rho_{1,2} \leq 0.5$) the term *fast* has membership grade 1 while the other terms have grade zero. This means that the controller will be able to apply the maximum reduction of the parallelism level in case the stage speed is at least two times higher than its input rate (in the inference rules part we will explain the reason for this choice);
- a desired situation is when the stage speed is slightly higher than its input speed. In this case the stage utilizes the minimum resources to avoid being a bottleneck. This reflects in a utilization factor slightly smaller than 1 (we chose 0.9 as our default value). Therefore, if $\rho_{1,2} = 0.9$ the term *acceptable* has membership grade 1 while the others have zero grade;
- if the stage speed is at least 30% slower than its input speed (i.e. $\rho_{1,2} \geq 1.3$), the stage is *slow* with grade 1 while the other terms have grade zero. This value has been chosen to make the center edge of the *acceptable* membership function (at 0.9) equidistant between the right foot and left foot of the *fast* and *slow* functions respectively, which makes their shapes symmetric (a common rule in FLCs [13]).

The fuzzification phase assigns to a crisp input a fuzzy value obtained according to the membership grades of all the linguistic terms. For example, if $\rho_1 = 0.6$ the fuzzification result is $\overline{\rho}_1 = 0.750/\text{fast} + 0.250/\text{acceptable} + 0.000/\text{slow}$. The meaning is that the crisp value is converted into the term *fast* with membership grade 0.75, *acceptable* with grade 0.25 and zero grade for the term *slow*. Fig. 7(right) shows the fuzzy sets of the `SPLITTING` variable corresponding to the σ_{avg} crisp input, with two terms (*moderate* and *intensive*) with trapezoidal-shaped membership functions. An approach to choose the shape of

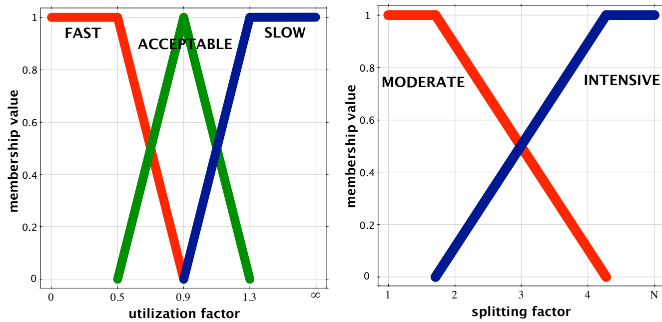


Figure 7: Fuzzy membership functions of the linguistic variables `RHO_PLQ` and `RHO_WLQ` (left) and `SPLITTING` (right).

membership functions is to compute some statistics of the values assumed by the crisp input and to divide the range of observations in function of the data percentiles [30]. In our case the right shoulder and the left shoulder of the two functions correspond to a splitting factor of 1.5 and 4.5 respectively. Such values represent the 25% and the 75% quartile of all the splitting factor statistics gathered by running `Elastic-PPQ` in our past experience.

Inference rules. The strategy is developed as a set of rules in the form *if condition then action*. The antecedent is a sequence of propositions "variable is term" connected by a conjunctive fuzzy *and* operator. In the consequents we use two linguistic variables `PAR_VAR_1` and `PAR_VAR_2` which indicate a relative variation of the parallelism level of the first and the second stage respectively. Five terms are defined: *decrease*, *slight_decrease*, *unchanged*, *slight_increase* and *increase*.

A problem is the so-called *settling time*, i.e. the number of intervals needed to reach the target parallelism level [3]. If the controller performs large variations the settling time is minimized at the expense of large oscillating decisions. Instead, a controller that applies small modifications (e.g., of one or few workers) can feature a long settling time that may be unsuitable to react to fast workload variabilities. In our approach the controller changes the number of workers at most of 50% of the current value. Therefore, the parallelism level grows/decays following an exponential rule, which allows reaching the target parallelism level in few intervals. We observe that if the utilization factor is 0.5 the number of workers can be halved to reach an utilization factor of 1. This explains why we apply the maximum reduction of the parallelism level when the utilization factor of the stage becomes less or equal than 0.5.

To model this behavior we adopt the zero-order Takagi-Sugeno approach [31] where the output terms correspond to constant values (see correspondence in Tab. 1). This model does not require general membership functions for the output terms and complex defuzzification methods (like in Mamdani controllers [13]) and this makes easier the definition of the rules modeling the herustics that we have designed for `Elastic-PPQ`, as it will be explained in the defuzzification part.

decrease	slight decrease	unchanged	slight increase	increase
0.5	0.75	1	1.25	1.5

Table 1: Constant values associated with each linguistic term of the output variables.

The rules are listed in Tab. 2. Some of them are straightforward, like rule no. 1. Others deserve more explanations:

- rules no. 7 and 8 state that we slightly increase the parallelism level of the first stage if its speed is acceptable, the splitting factor is intensive, and the second stage is not slow. The use of some additional PLQ workers is beneficial to divide less the panes by reducing the tuples comparisons in the WLQ [14];
- rule no. 9 avoids increasing the PLQ parallelism level if the WLQ stage is slow. In this way we give more priority to the acquisition of new workers by the second stage, which is impairing the query throughput;
- rules no. 10 and 11 try to prevent a future bottleneck. In rule n. 10, the parallelism level in the WLQ stage is slightly decreased in case the strategy needs simultaneously to increase the parallelism degree in the first stage. In this scenario, it is likely that the load to the second stage will become higher because the first stage will be able to produce results faster. Similarly, rule no. 11 slightly increases the number of workers in the second stage if its current speed is acceptable but we are increasing the parallelism in the first stage.

Defuzzification. The *activation degree* of each rule is evaluated starting from the membership grade of the input terms. Propositions in the antecedent are connected by the *and* operator (T-norm): according to the Fuzzy Logic semantics [12], the activation degree is obtained by computing the minimum between the membership grades of all the terms in the antecedent. According to the zero-order Takagi-Sugeno model, each output term has a scalar value (see Tab. 1) instead of a general membership function. The final defuzzified crisp outputs of the FLC are obtained by taking the weighted average of the outputs of the rules, where the weights are the activation degrees. This leads to a direct interpretation of rules. As an example, rule no. 1 halves the parallelism level of the two stages if they are both *fast* with membership grade 1, otherwise the reduction produced by the rule is proportionally reduced according to its activation degree. Finally, the FLC rounds the parallelism levels to the nearest integer before sending reconfiguration messages to the system.

5. Evaluation

`Elastic-PPQ` has been implemented using the FastFlow parallel programming framework targeting shared-memory machines [32]. FastFlow is a `C++11` library that allows the programmer to build graphs of streaming operators. Each

Rules no.	If			Then	
	RHO_PLQ	SPLITTING	RHO_WLQ	PAR_VAR_1	PAR_VAR_1
1	FAST	-	FAST	DECREASE	DECREASE
2	FAST	-	ACCEPTABLE	DECREASE	UNCHANGED
3	FAST	-	SLOW	DECREASE	INCREASE
4	ACCEPTABLE	MODERATE	FAST	UNCHANGED	DECREASE
5	ACCEPTABLE	MODERATE	ACCEPTABLE	UNCHANGED	UNCHANGED
6	ACCEPTABLE	MODERATE	SLOW	UNCHANGED	INCREASE
7	ACCEPTABLE	INTENSIVE	FAST	SLIGHT_INCREASE	DECREASE
8	ACCEPTABLE	INTENSIVE	ACCEPTABLE	SLIGHT_INCREASE	UNCHANGED
9	ACCEPTABLE	INTENSIVE	SLOW	UNCHANGED	INCREASE
10	SLOW	-	FAST	INCREASE	SLIGHT_DECREASE
11	SLOW	-	ACCEPTABLE	INCREASE	SLIGHT_INCREASE
12	SLOW	-	SLOW	SLIGHT_INCREASE	INCREASE

Table 2: Inference rules adopted by the Fuzzy Logic Controller.

operator performs a loop that: *i)* gets a data item (through a memory pointer to a data structure) from one of its input queues; *ii)* executes a code working on the data item and possibly on a state maintained by the operator; *iii)* puts a memory pointer to the result item into one or multiple output queues selected according to a user-defined policy. Operations on FastFlow queues are based on non-blocking lock-free synchronizations enabling fast processing in high-frequency streaming scenarios [33].

The application graph can be built by composing and nesting customizable parallel patterns. One is the *pipeline*, which allows the tandem composition of operators working on different data items in parallel. Another is the *farm* pattern, where the same computation is replicated and takes place on different data items in parallel. `Elastic-PPQ` has been designed as a pipeline of two farm patterns by following the general picture in Fig. 6.

5.1. Experimental setup

`Elastic-PPQ`[3] can be used by the user with some predefined preference queries and it can also be extended to execute other user-specified queries. In this case, the user must provide the implementation of a relatively small set of C++ classes with the implementation of few methods that apply a user-defined preference criterion. In the analysis we consider:

- the *skyline query*, which computes the set of non-comparable tuples using the Pareto dominance [15];
- the *top-δ dominant query* [34], which finds the smallest $k \leq d$ such that there are more than $\delta > 0$ k-dominant skyline points.

The queries use the Block Nested Loop algorithm [15] and, to increase the computational requirements, we will use data streams of high-dimensional tuples (we use eight floating-point attributes per tuple).

`Elastic-PPQ` is implemented on top of `FastFlow`[4] version 2.1 and compiled with `gcc 4.8.5` with the `-O3` optimization flag. The controller is implemented using the `FuzzyLite` library[4] version 5.0.

The used architecture is shown in Fig. 8. It is the last-generation Intel Xeon Phi model 7210 (codename Knights Landing, KNL) [35]. KNL is a highly parallel many-core processor equipped with 32 tiles, interconnected by an on-chip mesh network, each one with two cores working at 1.3 GHz. Each core has 32 KB L1d private cache and the two cores on the same tile share a L2 cache of 1 MB. The system is configured with 96 GB of DDR4 RAM and has 16 additional GB of high-speed on-package MCDRAM configured in cache mode [35].

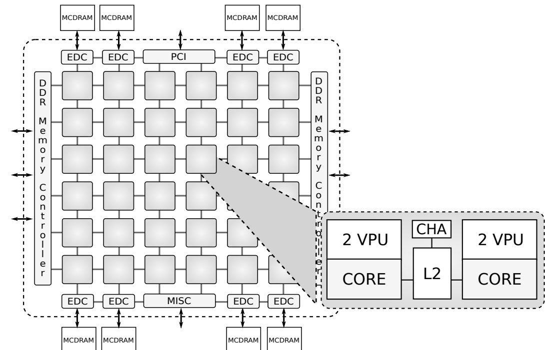


Figure 8: Intel Knights Landing internal architecture.

Owing to the busy-waiting synchronizations adopted by the FastFlow run-time support [32], we decided to map at most one thread per core. In case the controller decides to allocate more workers, the output thread levels of the controller are reduced proportionally to meet the global constraint on the maximum total thread level $N > 0$. Solutions based on mapping more threads onto the same core (by exploiting Simultaneous Multithreading) will be analyzed in our future work.

5.2. Preliminary experimental analysis

In this analysis we use synthetic data streams. Burstiness is injected using two different methodologies. The first is a light-tailed model based on a class of Markov

[3]The source code has been made publicly available at the following url: `https://github.com/ParaGroup/Elastic-PPQ/`.

[4]FastFlow: `http://calvados.di.unipi.it/`, FuzzyLite: `http://www.fuzzylite.com/cpp/`

Modulated processes called Markovian Arrival Processes [23] (MAP), where the process has a "*fast*" and a "*slow*" state. In each state the inter-arrival times are generated using an exponential distribution with rate λ_f and λ_s respectively. The values of the two rates and the transition probabilities are the result of the fitting procedure described in [22], in order to achieve a target average rate λ and a desired index of dispersion $\mathcal{I} \geq 1$. This parameter is of fundamental importance, as it permits the burstiness level to be arbitrarily adjusted, i.e. the higher the index of dispersion (of some thousands of units) the higher is the variability and correlation of inter-arrival times, which turns out in a more severe burstiness level.

The second traffic model exhibits self-similarity and the *scale invariant* property hinted in Sect. 2.2. The Hurst parameter $\mathcal{H} \in [0.5, 1)$ is used to measure the level of self-similarity: values close to one correspond to time-series with a very long-range dependence [19]. We use the heavy-tailed Pareto distribution to model the amount of arrivals per time slot (of one millisecond by default).

Finally, all the synthetic data streams have been disordered by adding a uniformly distributed random delay to all the timestamps, and by generating tuples according to the delayed timestamps.

Artificial scenario. We study an artificial scenario where `Elastic-PPQ` executes the skyline query on windows of $w = 1$ second and slide $s = 0.1$, thus the panes have length of 100 milliseconds. Fig. 9 shows the results of this experiment where the x-axis corresponds to the elapsed execution time in second. The input rate is shown in Fig. 9(top) with a black solid line (the corresponding y-axis is on the left hand side of the figure). The initial rate is of $40K$ tuples/sec and is changed every 30 seconds (note the stepwise shape of the curve).

Fig. 9(top) shows the PLQ thread level (blue solid line) and the splitting factor per second (small bars in the plot). The y-axis for these measures is on the right hand side of the figure. Not surprisingly, the controller increases the thread level in the instants where the input rate changes. Fig. 9(middle) shows the utilization factor. We note that:

- in the first 30 seconds the PLQ stage has 3 workers, and the panes are evenly split (the splitting bars are close to the blue line of the thread level in Fig. 9(top));
- after second 30 the rate grows of 25% and the PLQ stage starts becoming a bottleneck, see the high utilization peaks in Fig. 9(middle) at second 30. The controller reacts by allocating a new worker. From 32-60 seconds the PLQ stage is able to keep again its utilization close to the setpoint by using a splitting factor $\sigma_{avg} \approx 3.5$, i.e. the bars in Fig. 9(top) are slightly lower than the thread level after second 30;
- after second 60 the rate grows again of 40%. The PID adapts by increasing the splitting factor, which becomes maximum ($\sigma_{avg} = 4$) in the instants just after second 60. However, the PLQ stage is still a bottleneck and the system needs to allocate additional workers: initially we add a further worker which is not sufficient, finally at second 70 the controller decides to use 6 workers and the system succeeds in achieving the ideal throughput again, see see Fig. 9(bottom);
- at second 90 a further increase in the input rate is handled transparently to the controller by the PID-based scheduler, which is able to track the utilization setpoint by increasing the splitting factor. In this phase there is no change in the thread level;
- at second 120 and 150 the input rate decreases. The effect is a drop in the utilization factor, handled by the controller with a decrease in the thread level.

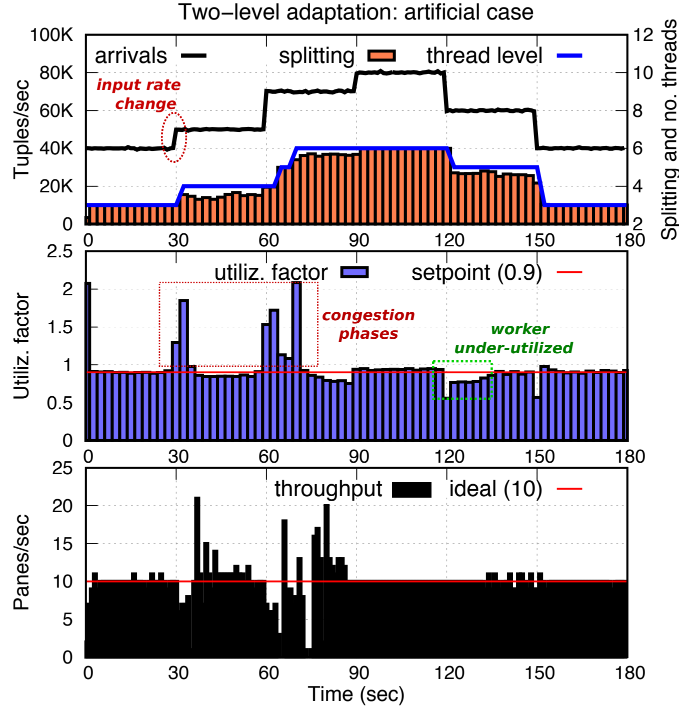
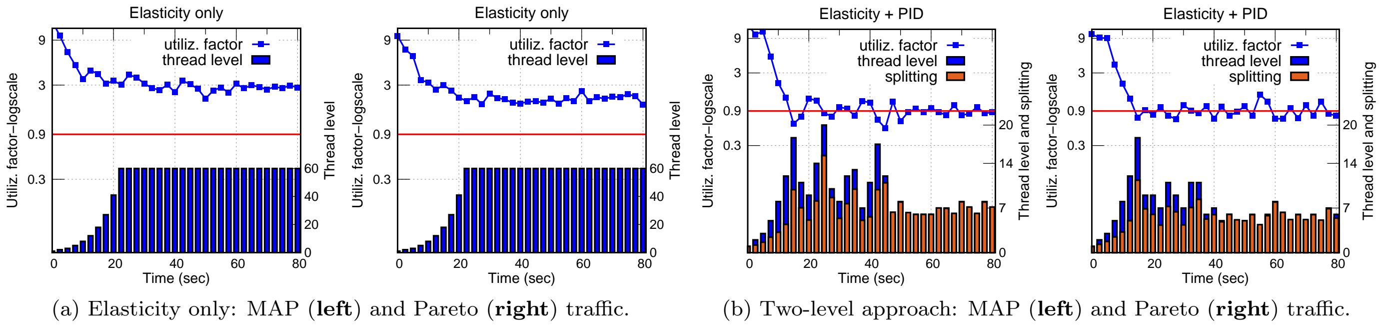


Figure 9: First evaluation of the two-level autonomic approach on an artificial scenario: analysis of the PLQ stage behavior.

Fig. 9(bottom) shows the throughput. The ideal number of panes that the system can complete is of $1/L_p = 10$ panes/sec (red line). Most of the time the system is able to keep the throughput near to its ideal value. Interestingly, phases with dips in throughput correspond to time intervals where the utilization factor is higher than one, while the throughput is always ideal when the utilization is close to its setpoint 0.9. This confirms that the utilization metric presented in Sect. 3.1 is an effective approach to measure the congestion. Time periods with measured throughput greater than the ideal one, see the peaks in Fig. 9(bottom), happen after the removal of a congestion phase, when the system drains all the pending tuples enqueued when it was a bottleneck.

Handling bursty scenarios. In this part we show why the elasticity mechanisms may be not enough, and the reasons for the combined used with the PID-based scheduler.

Fig. 10 illustrates an experiment with the skyline query



(a) Elasticity only: MAP (**left**) and Pareto (**right**) traffic. (b) Two-level approach: MAP (**left**) and Pareto (**right**) traffic.

Figure 10: Analysis of the PLQ stage in bursty scenarios: utilization factor, number of PLQ workers, and average splitting factor measured per controller interval of 2.5 seconds (PID sampling period of 250 ms).

and panes of 100 ms. Again we focus on the PLQ stage. We use two synthetic data streams with average rate of $80K$ tuples/sec. In the first, inter-arrival times are generated using a MAP distribution with $\mathcal{I} = 1K$ which is a case of remarkable burstiness [14, 22]. The second is a self-similar traffic obtained with a Pareto distribution with Hurst parameter of 0.6. Both the data streams are out-of-order with average delay of 200 ms.

Fig. 10a shows the utilization factor measured at each controller interval. In this case we force the scheduling to avoid splitting panes. The PLQ stage is initially a bottleneck with a high utilization. The blue bars represent the number of workers used at each interval. The controller reacts by increasing the parallelism level which rapidly reaches the maximum (in this case 60). In the initial part of the execution the use of additional threads is beneficial since the utilization factor lowers. However, in both the traffic models after the first 20 seconds any further increase in the parallelism level is useless and the utilization factor stabilizes around a value of 3 in the MAP traffic, which means that the PLQ stage is able to withstand only 1/3 of the input bandwidth. The reason is that additional workers in this strategy are useful only to exploit parallelism among panes, which may be relatively small and insufficient to achieve the ideal throughput.

Fig. 10b shows the results using our two-level approach. Here the system is able to keep the utilization factor close to the desired setpoint of 0.9. The figure uses two stacked bars per interval. The plot can be read as follows: for each interval the yellow bar corresponds to the splitting factor while the blue bar is the thread level used in response to the utilization factor (blue point) measured at the previous interval. An increase in the thread level is triggered when the utilization was higher than the setpoint and vice-versa. In case of large variations in the thread level (mostly in the first execution phase) we observe that the PID-based scheduler is not always able to maximize the splitting when the PLQ stage is a bottleneck, because the controller interval is not long enough and more time would be needed to further lower the splitting threshold. However, in those cases the PID-based scheduler does its best and the utilization factor would have been worse without its help.

A question arises: why is it not convenient to always split all the panes evenly among the workers? To examine this issue, Fig. 11(left) shows the results of some experiments with the MAP traffic. We run the system several times, each run with a different static thread level (from 1 to 12). In each run the splitting factor is forced to be equal to the number of utilized workers. For each thread level we report the average utilization factor measured throughout the execution. By using more workers, we lower the utilization factor because the pane partitions are smaller and the processing time per tuple is reduced. However, higher splitting factors lead to a greater *filtering ratio*, defined as the number of tuples selected per pane to the total number of pane tuples. In the skyline query, the selected tuples are the ones in the skylines of the pane partitions. In the experiment, without splitting panes only 27% of the input tuples reach the second stage, while with a high splitting factor of $\sigma_{avg} = 12$ the ratio is of 58%.

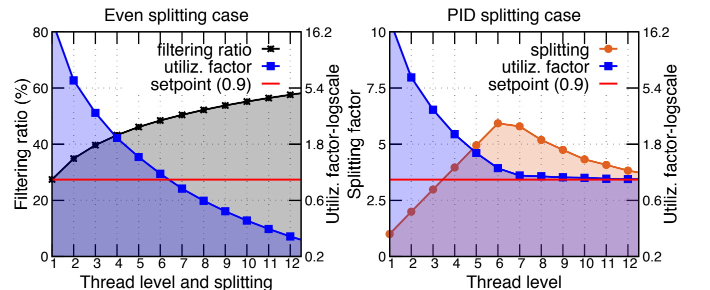


Figure 11: Analysis of the splitting approach: even splitting case (**left**) and PID-based splitting (**right**).

Fig. 11(right) shows the results with the PID. When the thread level is not high enough the PID splits as much as possible. With 7 workers the bottleneck can be eliminated and the PID splits the panes long enough to reach the setpoint. E.g., with eight workers we have $\sigma_{avg} = 5.18$ and the ratio is of 43% instead of 54% with $\sigma_{avg} = 8$.

Evaluation of the WLQ scheduling. In this part we evaluate the feedback-based scheduling by focusing on the optimizations presented in Sect. 3.2.

Fig. 12 shows an experiment executed under a MAP traffic ($\mathcal{I} = 1K$ and average rate of $30K$ tuples/sec).

We use windows of $w = 1$ second with slide of 200 ms ($L_p = 200$ ms). We focus on the second stage by comparing two scheduling strategies: `fb-default` is the one proposed in [14] where only `WIN_UPDATE` tasks are dispatched to the workers; `fb-opt` is the new strategy (Sect. 3.2) that allows the WLQ emitter to further schedule `MERGE` tasks.

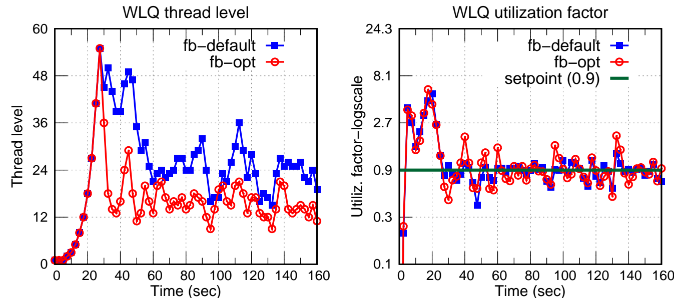


Figure 12: Analysis of different variants of the feedback-based scheduling in the WLQ stage: controller interval of 2.5 seconds.

Fig. 12(left) shows the number of WLQ workers chosen by the controller per interval, and Fig. 12(right) reports the measured utilization factor. The general outcome is that both the strategies are able to maintain the utilization factor close to the setpoint. However, with the `fb-default` strategy the WLQ stage utilizes more workers on average. To understand this phenomenon, we can look at the results in Fig. 13 where we plot for each interval the fraction of time that the activated WLQ workers have passed without doing useful work, i.e. they are idle. In the hypothetical case of an optimal controller, the idle time should be zero, meaning that the controller always allocates the right number of resources per interval. This condition is not realistic for two reasons:

- first, there are intervals where the input rate drops and the WLQ stage becomes under-utilized. During such intervals the idle time is inevitably high;
- second, it is possible that although some results of pane partitions are pending, no further `WIN_UPDATE` tasks can be scheduled because the windows are currently busy. In this scenario the stage is a bottleneck despite some workers remain idle for a certain time.

The second point is addressed by the `fb-opt` strategy: if all the windows are busy but some results of pane partitions are pending in the WLQ emitter, they can be merged by assigning useful `MERGE` tasks to some idle workers. The effect is that the workers are better exploited and the idle time is 45% lower on average, as Fig. 13 shows.

`MERGE` tasks are 37% of the total number of tasks. We note that in the first 25 seconds (see Fig. 12) the utilization factor and the number of workers is higher. The reason is that the WLQ stage starts with only one worker and a certain time is needed to drain all the tuples received in the initial phase, when the stage was a bottleneck. However, in a long running this initial phase is negligible.

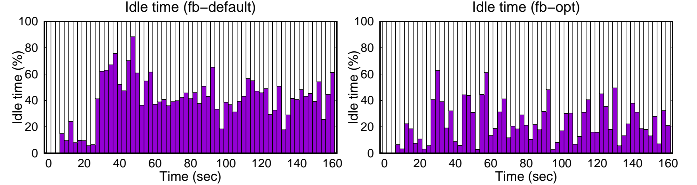


Figure 13: Idle time of the WLQ workers with/without the scheduling of `MERGE` tasks.

5.3. *Elastic-PPQ at work*

In this part we study the behavior of the whole system with the following synthetic and real-world data streams:

- a stream (`RW-DS`) with non-stationarities (increasing/decreasing trends). Inter-arrival times are exponentially distributed with the mean changed every five seconds according to a lognormal random walk model;
- seven bursty streams, four MAP traffic traces (`MAP-DS`) and three self-similar flows (`SS-DS`). The maximum index of dispersion of $4K$ is considered a very severe burstiness in past research papers [22], while the highest Hurst parameter of 0.8 represents a high level of self-similarity [19];
- four real streams (`Game1-4`) obtained from the Real-time Locating System used for the DEBS 2013 grand challenge [36]. Each dataset is a stream of eight-attributed tuples received from the sensors embedded in the shoes of players during a match played in the soccer stadium of Nuremberg, Germany.

In the synthetic data streams the tuple delay is uniformly distributed with mean 200 ms. Instead, the real data streams have their own out-of-order characteristics shown in Tab. 3, and they have been accelerated two times with respect to the original traces. The table also shows the average rate and the peak to mean ratio per stream.

	Name	Speed		Delay	
		tuples per sec	peak to mean	avg(ms)	max(sec)
Synthetic	`RW-DS`	11,816	4.17	200	∼ 0.4
	`MAP-DS` $\mathcal{I} = 1K$	30,331	1.57		
	`MAP-DS` $\mathcal{I} = 2K$	29,659	1.75		
	`MAP-DS` $\mathcal{I} = 3K$	28,873	2.01		
	`MAP-DS` $\mathcal{I} = 4K$	30,189	2.03		
	`SS-DS` $\mathcal{H} = 0.6$	25,582	2.24		
	`SS-DS` $\mathcal{H} = 0.7$	26,552	2.06		
	`SS-DS` $\mathcal{H} = 0.8$	26,028	3.83		
Real	`Game1`	16,923	1.44	17.1	71.1
	`Game2`	19,272	1.16	14.1	85.5
	`Game3`	17,470	1.13	13.2	38.3
	`Game4`	23,106	1.12	12.2	27.2

Table 3: Characteristics of the utilized data streams.

Detailed evaluation of `Game1`. Fig. 14 describes the system behavior with data stream `Game1`. We use the skyline query with the sliding windows used at the end of the previous section. As Fig. 14(left) shows, the arrival rate fluctuates intensively over time. At the end of the execution we measured an average utilization factor of the two

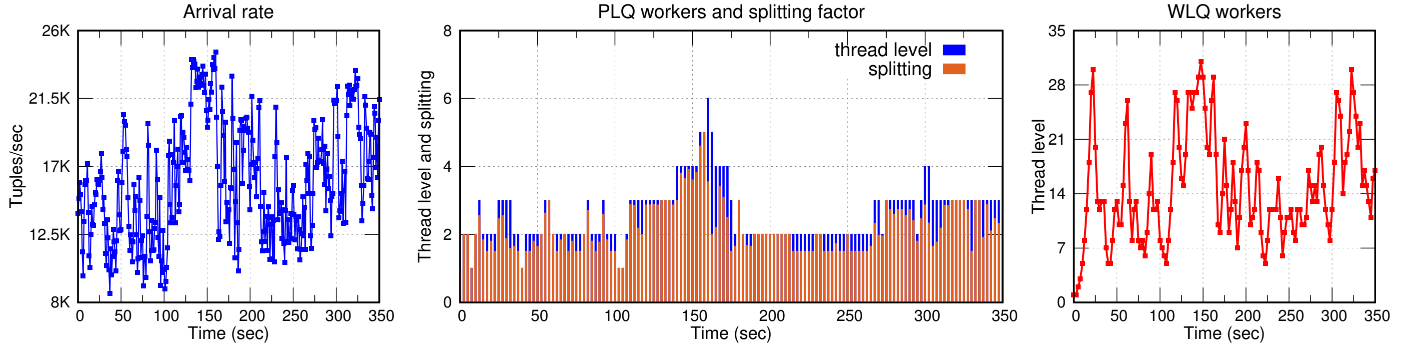


Figure 14: Detailed analysis of the `Game1` workload trace: arrival rate per second (**left**), PLQ thread level and splitting factor (**middle**), number of threads of the WLQ stage (**right**). Controller interval of 2.5 seconds (PID sampling period of 250 ms).

stages equal to 0.882 and 0.912, which are both close to the utilization setpoint. Fig. 14(middle) shows the number of PLQ workers activated by the controller and the average splitting factor applied by the PID at each controller interval. Two considerations can be made. First, the stage tweaks its thread level in light of changes in the arrival rate, with the highest level near second 150 corresponding to the peak rate. Second, the splitting factor is not always maximum because the controller may overprovision the thread level and using the maximum splitting is often useless to optimize throughput. Interestingly, the rate variability in the last 50 seconds is amortized by adjusting the splitting factor without changing the thread level. Finally, Fig. 14(right) shows the workers assigned to the WLQ stage, which also keeps up with the arrival rate. They are more than the first stage, because the WLQ stage receives results of pane partitions (tens per second) and the task processing time is several orders of magnitude greater than the processing time of single tuples in the first stage.

To have an idea of the QoS perceived by the users, Fig. 15 illustrates the number of window results produced per second. We compared the throughput of the two-level approach against the one of a non-elastic processing, where the number of workers in the two stages is fixed to 6 and 38. Such values are chosen by executing various runs with different static thread levels and by finding the minimum ones that avoid the two stages being a bottleneck throughout the execution[5]. This static configuration, which is unrealistic in general because not known *a-priori*, represents an optimal case from the performance viewpoint, thus a valuable baseline to compare our autonomic approach with.

As the figure shows, the average throughput is close to the maximum bandwidth of five windows per second in both cases (difference less than 1%). This is achieved owing to the low overhead of our approach. In fact, the strategy always runs for a negligible fraction of the controller interval and it is always overlapped with the processing

[5]The sum of the two thread levels may exceed the maximum no. of cores. This is another reason in favor of elasticity.

Figure 15: Throughput (windows completed per second) of the system under the `Game1` real data streams.

flow. Instead, the scheduling logic is executed directly by the two emitters. However, it requires few instructions to update the counters and scheduling internal variables, while the PID is evaluated every sampling period and not for every tuple, thus its cost is amortized. We measured an overhead in the average emitters service time bounded by 8%. The main effect of elasticity is a slightly less stable throughput. We measured the *coefficient of variation* (CV), the ratio of the standard deviation to the mean of throughput. Cases with a CV less than 1 are low-variance. In our case the CV is 0.86 and 0.66 in the two scenarios, thus the variability increase is limited demonstrating the effectiveness of our autonomic approach.

Summary of results for the other data streams. The results are provided in Tab. 4 for the skyline query, while Tab. 5 shows the results of the Top-δ query with $\delta = 100$. The second query is slightly more computationally demanding [14]. The tables report some parameters: the average thread level of the PLQ and WLQ stages and the average splitting factor of the first stage. In all cases the measured throughput is very close to the theoretical maximum bandwidth: Δ_{th} measures the additional time spent to complete the experiment with respect to the generation time of the whole stream. A limited increase is tolerated, since a small latency is always paid to complete the processing of the last window results after the completion of the stream transmission. As it is evident, each experiment is completed in nearly the same time spent in generating all the tuples $\Delta_{th} < 3.1\%$. In terms of throughput variability, with higher values of the index of dispersion

	Stream	Threads		Split.	Throughput	
		PLQ	WLQ	σ_{avg}	Δ_{th} (%)	CV
Synthetic	RW-DS	1.77	12.6	1.48	0.40	1.60
	MAP-DS $\mathcal{I} = 1K$	4.28	41.0	3.37	1.39	0.82
	MAP-DS $\mathcal{I} = 2K$	4.16	38.8	3.63	1.77	0.95
	MAP-DS $\mathcal{I} = 3K$	4.30	37.2	3.54	2.37	1.29
	MAP-DS $\mathcal{I} = 4K$	6.01	36.8	4.86	2.64	1.85
	SS-DS $\mathcal{H} = 0.6$	3.05	31.5	2.99	1.02	0.86
	SS-DS $\mathcal{H} = 0.7$	3.47	34.7	3.24	1.11	1.17
	SS-DS $\mathcal{H} = 0.8$	3.50	30.9	3.05	2.19	2.30
Real	Game1	3.82	15.65	2.74	1.21	0.86
	Game2	2.30	25.3	2.06	1.76	1.15
	Game3	1.98	21.0	1.89	0.23	0.78
	Game4	2.48	28.9	2.34	1.19	1.12

Table 4: Summary of results with the skyline query.

	Stream	Threads		Split.	Throughput	
		PLQ	WLQ	σ_{avg}	Δ_{th} (%)	CV
Synthetic	RW-DS	2.61	17.0	2.08	0.44	1.43
	MAP-DS $\mathcal{I} = 1K$	5.43	49.5	4.31	1.92	0.96
	MAP-DS $\mathcal{I} = 2K$	5.26	46.28	4.53	2.14	1.24
	MAP-DS $\mathcal{I} = 3K$	5.65	45.3	4.81	2.68	1.58
	MAP-DS $\mathcal{I} = 4K$	6.95	43.06	5.13	3.06	1.99
	SS-DS $\mathcal{H} = 0.6$	3.71	38.6	3.41	1.32	0.97
	SS-DS $\mathcal{H} = 0.7$	3.83	41.8	3.27	1.87	1.34
	SS-DS $\mathcal{H} = 0.8$	4.01	37.9	3.68	2.74	2.97
Real	Game1	4.11	27.7	2.98	1.45	0.96
	Game2	2.81	33.5	2.32	2.27	1.23
	Game3	2.59	31.7	2.09	0.57	0.93
	Game4	3.10	35.3	2.62	1.56	1.96

Table 5: Summary of results with the top-δ query.

and the Hurst parameter the CV increases.

The approach has also the benefit of reducing the power consumption without impairing throughput. Fig. 16 shows the consumption (in percentage) compared with the one with all the cores of the machine working at full speed. As we can observe, the power reduction is significant especially in the real-world data streams. It should be noted that also a small power saving is meaningful, because of the long-running nature of stream processing queries.

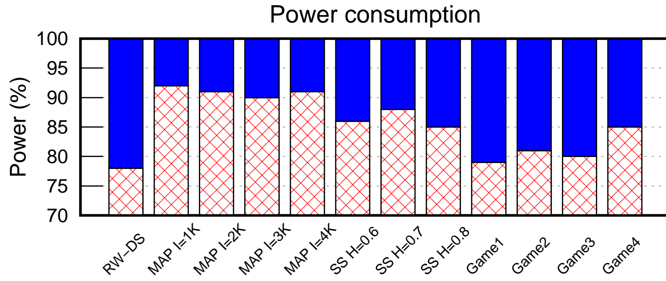


Figure 16: Power consumption with the autonomic approach (skyline query).

Sensitiveness to configuration parameters. The length of the controller interval $T_{control}$ and of the PID sampling period T_{pid} may influence the system behavior. To analyze this aspect we repeated the `Game1` experiment with the skyline query using different lengths of $T_{control}$ and T_{pid}. Fig. 17(left) depicts the relative error between the measured PLQ utilization factor and the setpoint. Each line represents a different choice of the controller interval. Within each interval the PLQ stage uses a stable number of workers, and the PID adapts the splitting threshold to get closer to the setpoint. The more sampling periods we have within each interval the better is the PID accuracy while longer periods make the PID less reactive. As an empirical rule we found that the number of sampling periods within each interval must be in the order of tens in order to get an error of 1-2%.

A similar consideration can be made for the controller interval. If it is too long the system responds slowly to deviations from the desired behavior that may become large. Concretely, the system may be not able to adapt to changes in the arrival rate within the interval like the

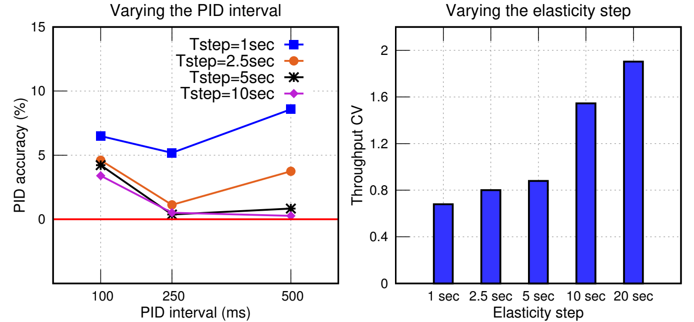


Figure 17: PID accuracy with various lengths of the controller interval and the PID sampling period (**left**), coefficient of variation of throughput (**right**).

ones in Fig. 14(left). The effect is that query results are produced haltingly, which is not ideal for the real-time analysis by the users. To confirm this, Fig. 17(right) reports the throughput CV with different controller intervals where longer intervals imply greater instability.

5.4. Robustness to interferences with other applications

In this final part we study how `Elastic-PPQ` effectively adapts to the external load originated by other applications running on the same machine.

We repeated the experiment with the data stream `Game1`. During the execution, we injected an external load obtained by running the `stress-ng` utility for the Linux OS[6]. This tool is designed to stress a computer system in various selectable ways using a wide range of specific stress tests (called stressors) that exercise CPU resources (floating-point and control flow units), caches, memory and I/O. We chose a selection of 15 tests that stress CPU resources and the cache hierarchy: `bsearch`, `cache`, `hsearch`, `icache`, `lockbus`, `lsearch`, `malloc`, `matrix`, `memcpy`, `qsort`, `str`, `tsearch`, `vecmath`, `wcs`, `zlib`. They consist in well-known sorting algorithms, linear-algebra computations, large string manipulations, and so forth.

Fig. 18 shows the results with different intensity levels of the external load. The level is measured as the ratio of the

[6] https://github.com/ColinIanKing/stress-ng

number of injected threads that continuously run random stressors to the total amount of virtual cores available in the machine. As we can see from the figure, a higher interference turns out in a higher thread level needed by our system, see Fig. 18(left). Fig. 18(middle) shows the total thread level (PLQ + WLQ stages) utilized during each interval of 2.5 seconds in two cases of a low and a high external load intensity. The total thread level in both cases keeps up with the changes in the arrival rate, see Fig 14(left). However, workers perform slowly in the case with severe interference, because fewer computational cycles are available and due to the mutual interference (competition for using processor's units and private caches) among threads executed on virtual cores in the same physical core. The results confirm that our system adapts to the external load by allocating more workers. Except in the case of very high intensity levels, no appreciable throughput degradation is measured (see Fig. 18(right)) at the expense of a higher throughput variability.

6. Related Work

A long research endeavor has been made to define paradigms helping the design of autonomic and elastic applications in the Cloud. Cloud-based applications are usually defined as composition of services and components that are linked and utilized by the execution flow that transforms inputs into corresponding outputs. Description languages and models [37, 38] have been proposed with the goal of enabling the dynamic composition of services/components in order to adapt to changing users' requirements. In addition to mechanisms that dynamically link/remove components or services, an intensive research effort has been spent to provide applications with a fully autonomic logic able to derive automatically which is the best reconfiguration plan allowing the applications to achieve the current users' requirements without human intervention. Formal descriptive languages [39, 40] have been proposed in order to deduce the behavior that must be adopted in new contexts that have not earlier been examined by the application developer.

Differently from the Cloud domain, in Data Stream Processing the use of elastic solutions is more recent and not yet established as the norm. The work in [7] presents the development of an elastic stream processing framework for IoT environments. The authors designed an elastic system able to deal with changing rates by significantly reducing the operating costs. The approach reacts to situations where system QoS measurements exceed or violate predefined crisp thresholds, and this may lead to instability in case of very erratic workload. In fact, the approach is mainly studied for long-term steady load variations, and it is not designed for handling fast time-scale burstiness.

A very interesting work has been recently published in [6]. The proposed methodology is able to control the number of replicas in streaming operators. The authors proposed two algorithms to be applied at different time-scales. However, they have a different role with respect to the innermost and outermost adaptation levels in our approach. The short-term algorithm is used to determine whether an operator is a bottleneck in the present time based on deterministic thresholds. Again, such crisp thresholds are not easy to determine and values near to the boundaries may under- or over-estimate the bottleneck detection. The mid-term algorithm instead performs predictions of the future workload based on a Markov model, and it has been shown to be effective mainly for arrival rates exhibiting non-stationarities like trends or cyclic variations.

The idea of using model-driven controllers has been recently employed in [41] for generic sliding-window queries. When different windows are assigned to different operators for processing, the same tuple can be transmitted to multiple operators and this may increase the scheduling overhead. The approach decides how to group windows in batches assigned to the same operator in order to trade off scheduling overhead with processing latency. Although interesting for the use of control-theoretic controllers, in this work inter-arrival time variabilities at different time-scales have not been studied systematically though the approach can be extended in this sense.

Some papers like the ones in [42, 3, 43] focus on the state transfer in elastic stateful operators. In those cases, resource scaling needs actions to migrate the state with-

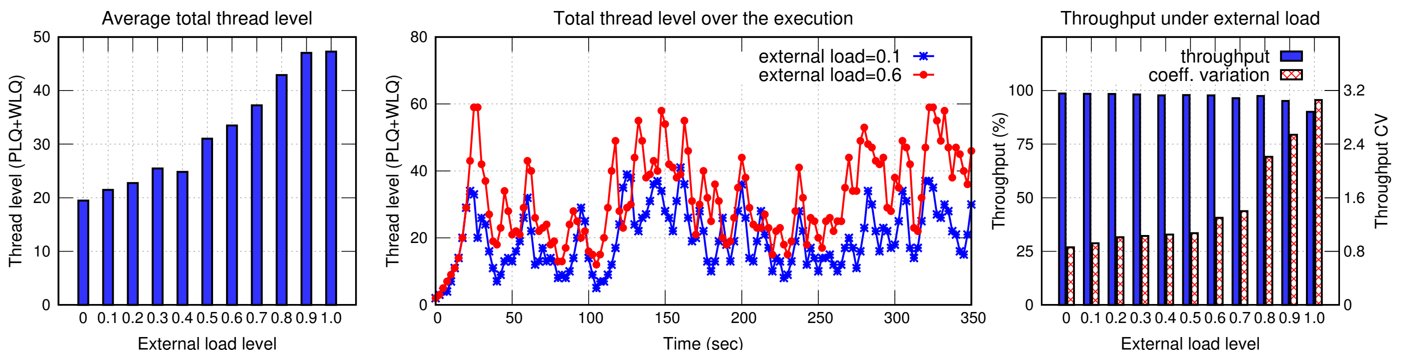


Figure 18: Effect of the external load on the `Game1` data stream: average total thread level needed by `Elastic-PPQ` (**left**), total thread level over the execution (**middle**), and throughput provided under different levels of external load intensity (**right**).

out altering the computation semantics. As shown in [44], such actions may generate latency spikes and throughput dips, and thus are suitable to handle long-/medium-term variations in the arrival rate. Instead, in `Elastic-PPQ` the pane splitting approach is aimed at avoiding such overhead by allocating pane partitions without blocking the tuple computations, and this revealed very useful to provide seamless processing over dynamic data streams.

As demonstrated in this paper, frequent sharp rises in the traffic volume cannot always be neutralized with resource scaling only. The impact of such kind of burstiness in traditional Cloud computing infrastructures has been studied from the consumer's perspective in [10]. This demonstrated that burstiness may be detrimental for the elastic behavior of Cloud systems. The deterioration can be pronounced either in under-provisioning or over-provisioning scenarios, resulting in more QoS violations in the former case or in a sub-optimal operating cost in the latter one. A recent work [20] focuses on the impact of burstiness on the dynamic resizing of data centers. The approach is based on an optimization problem solved to compute the optimal resource allocation that guarantees probabilistic bounds on the QoS parameters. The approach accounts for fast time-scale burstiness by properly increasing the resource provisioning with respect to the one needed under the equivalent non-bursty workload. As we have seen in this paper, over-provisioning is not always sufficient to handle burstiness, and finer regulatory mechanisms are needed to control load balancing.

7. Conclusions and Future Work

In this paper we propose `Elastic-PPQ`, a system for parallel processing of sliding-window preference queries. The approach features adaptation mechanisms suited to address variabilities happening at different time-scales. Short-term burstiness is addressed through adaptive scheduling techniques. Long/medium-term arrival rate variations instead are handled by a fuzzy logic controller. The experiments demonstrated that our system is able to keep up with the variability by optimizing throughput with good resource utilization efficiency and power saving.

Our work can be extended in several ways. Additional control knobs can be considered, such as the possibility to apply load shedding (randomly discard input tuples to reduce the stream pressure) and CPU frequency scaling to reduce power consumption. Since the complexity of fuzzy logic rules increases significantly with the number of knobs, machine learning techniques can be used to automatize this process. Furthermore, another research direction is to study the applicability of our two-level autonomic approach to support a wider class of sliding-window queries.

Acknowledgments

This work has been partially supported by the EU H2020 project RePhrase (EC-RIA, ICT-2014-1).

References

[1] J. Gama, Knowledge Discovery from Data Streams, 1st Edition, Chapman & Hall/CRC, 2010.

[2] H. Andrade, B. Gedik, D. Turaga, Fundamentals of Stream Processing, Cambridge University Press, 2014, cambridge Books.

[3] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463. doi:10.1109/TPDS.2013.295.
URL http://dx.doi.org/10.1109/TPDS.2013.295

[4] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463. doi:10.1109/TPDS.2013.295.
URL http://dx.doi.org/10.1109/TPDS.2013.295

[5] S. Schneider, H. Andrade, B. Gedik, A. Biem, K.-L. Wu, Elastic scaling of data parallel operators in stream processing, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12. doi:10.1109/IPDPS.2009.5161036.
URL http://dx.doi.org/10.1109/IPDPS.2009.5161036

[6] N. Hidalgo, D. Wladdimiro, E. Rosas, Self-adaptive processing graph with operator fission for elastic stream processing, Journal of Systems and Software 127 (2017) 205 – 216. doi:http://doi.org/10.1016/j.jss.2016.06.010.
URL http://www.sciencedirect.com/science/article/pii/S0164121216300796

[7] C. Hochreiner, M. Vögler, S. Schulte, S. Dustdar, Elastic stream processing for the internet of things, in: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), 2016, pp. 100–107. doi:10.1109/CLOUD.2016.0023.

[8] R. d. R. Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. de Bona, T. Ferreto, Autoelastic: Automatic resource elasticity for high performance applications in the cloud, IEEE Transactions on Cloud Computing 4 (1) (2016) 6–19. doi:10.1109/TCC.2015.2424876.

[9] G. Galante, L. C. E. d. Bona, A survey on cloud computing elasticity, in: 2012 IEEE Fifth International Conference on Utility and Cloud Computing, 2012, pp. 263–270. doi:10.1109/UCC.2012.30.

[10] S. Islam, S. Venugopal, A. Liu, Evaluating the impact of fine-scale burstiness on cloud elasticity, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, ACM, New York, NY, USA, 2015, pp. 250–261. doi:10.1145/2806777.2806846.
URL http://doi.acm.org/10.1145/2806777.2806846

[11] L. H. U, N. Mamoulis, K. Mouratidis, Efficient evaluation of multiple preference queries, in: 2009 IEEE 25th International Conference on Data Engineering, 2009, pp. 1251–1254. doi:10.1109/ICDE.2009.213.

[12] C. C. Lee, Fuzzy logic in control systems: fuzzy logic controller. i, IEEE Transactions on Systems, Man, and Cybernetics 20 (2) (1990) 404–418. doi:10.1109/21.52551.

[13] R.-E. Precup, H. Hellendoorn, A survey on industrial applications of fuzzy control, Computers in Industry 62 (3) (2011) 213 – 226. doi:http://dx.doi.org/10.1016/j.compind.2010.10.001.

[14] G. Mencagli, M. Torquati, M. Danelutto, T. D. Matteis, Parallel continuous preference queries over out-of-order and bursty data streams, IEEE Transactions on Parallel and Distributed Systems PP (99) (2017) 1–1. doi:10.1109/TPDS.2017.2679197.

[15] S. Borzsony, D. Kossmann, K. Stocker, The skyline operator, in: Data Engineering, 2001. Proceedings. 17th International Conference on, 2001, pp. 421–430. doi:10.1109/ICDE.2001.914855.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams, SIGMOD Rec. 34 (1) (2005) 39–44. doi:10.1145/1058150.1058158.
URL http://doi.acm.org/10.1145/1058150.1058158

[17] C. Balkesen, N. Tatbul, M. T. Özsu, Adaptive input admission and management for parallel stream processing, in: Proceedings of the 7th ACM International Conference on Distributed Event-

based Systems, DEBS '13, ACM, New York, NY, USA, 2013, pp. 15–26. doi:10.1145/2488222.2488258.
URL http://doi.acm.org/10.1145/2488222.2488258

[18] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 311–322. doi:10.1145/1066157.1066193.
URL http://doi.acm.org/10.1145/1066157.1066193

[19] D. G. Feitelson, Workload Modeling for Computer Systems Performance Evaluation, 1st Edition, Cambridge University Press, New York, NY, USA, 2015.

[20] K. Wang, M. Lin, F. Ciucua, A. Wierman, C. Lin, Characterizing the impact of the workload on the value of dynamic resizing in data centers, in: 2013 Proceedings IEEE INFOCOM, 2013, pp. 515–519. doi:10.1109/INFCOM.2013.6566826.

[21] M. Lin, A. Wierman, L. L. H. Andrew, E. Thereska, Dynamic right-sizing for power-proportional data centers, IEEE/ACM Trans. Netw. 21 (5) (2013) 1378–1391. doi:10.1109/TNET.2012.2226216.
URL http://dx.doi.org/10.1109/TNET.2012.2226216

[22] G. Casale, N. Mi, L. Cherkasova, E. Smirni, How to parameterize models with bursty workloads, SIGMETRICS Perform. Eval. Rev. 36 (2) (2008) 38–44. doi:10.1145/1453175.1453182.
URL http://doi.acm.org/10.1145/1453175.1453182

[23] R. Gusella, Characterizing the variability of arrival processes with indexes of dispersion, IEEE Journal on Selected Areas in Communications 9 (2) (1991) 203–211. doi:10.1109/49.68448.

[24] M. R. Nami, M. Sharifi, Autonomic computing: A new approach, in: First Asia International Conference on Modelling Simulation (AMS'07), 2007, pp. 352–357. doi:10.1109/AMS.2007.20.

[25] Y. Drougas, V. Kalogeraki, Accommodating bursts in distributed stream processing systems, in: Parallel Distrib. Proc., 2009. IPDPS 2009. IEEE International Symposium on, 2009, pp. 1–11. doi:10.1109/IPDPS.2009.5161015.

[26] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, D. Maier, Out-of-order processing: A new architecture for high-performance stream systems, Proc. VLDB Endow. 1 (1) (2008) 274–288. doi:10.14778/1453856.1453890.
URL http://dx.doi.org/10.14778/1453856.1453890

[27] S. Babu, U. Srivastava, J. Widom, Exploiting k-constraints to reduce memory overhead in continuous queries over data streams, ACM Trans. Database Syst. 29 (3) (2004) 545–580. doi:10.1145/1016028.1016032.
URL http://doi.acm.org/10.1145/1016028.1016032

[28] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, C. Fetzer, Quality-driven processing of sliding window aggregates over out-of-order data streams, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, ACM, New York, NY, USA, 2015, pp. 68–79. doi:10.1145/2675743.2771828.
URL http://doi.acm.org/10.1145/2675743.2771828

[29] S. Shevtsov, M. Berekmeri, D. Weyns, M. Maggio, Control-theoretical software adaptation: A systematic literature review, IEEE Transactions on Software Engineering PP (99) (2017) 1–1. doi:10.1109/TSE.2017.2704579.

[30] D. Malchiodi, W. Pedrycz, Learning Membership Functions for Fuzzy Sets through Modified Support Vector Clustering, Springer International Publishing, Cham, 2013, pp. 52–59. doi:10.1007/978-3-319-03200-9_6.
URL https://doi.org/10.1007/978-3-319-03200-9_6

[31] T. Takagi, M. Sugeno, Fuzzy identification of systems and its applications to modeling and control, IEEE Transactions on Systems, Man, and Cybernetics SMC-15 (1) (1985) 116–132. doi:10.1109/TSMC.1985.6313399.

[32] M. Danelutto, M. Torquati, Structured parallel programming with "core" fastflow, in: V. Zsók, Z. Horváth, L. Csató (Eds.), Central European Functional Programming School, Vol. 8606 of LNCS, Springer, 2015, pp. 29–75. doi:10.1007/978-3-319-15940-9_2.
URL http://dx.doi.org/10.1007/978-3-319-15940-9_2

[33] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, An efficient unbounded lock-free queue for multi-core systems, in: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 662–673. doi:10.1007/978-3-642-32820-6_65.
URL http://dx.doi.org/10.1007/978-3-642-32820-6_65

[34] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, Z. Zhang, Finding k-dominant skylines in high dimensional space, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 503–514. doi:10.1145/1142473.1142530.
URL http://doi.acm.org/10.1145/1142473.1142530

[35] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y. C. Liu, Knights landing: Second-generation intel xeon phi product, IEEE Micro 36 (2) (2016) 34–46. doi:10.1109/MM.2016.25.

[36] C. Mutschler, H. Ziekow, Z. Jerzak, The debs 2013 grand challenge, in: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, DEBS '13, ACM, New York, NY, USA, 2013, pp. 289–294. doi:10.1145/2488222.2488283.
URL http://doi.acm.org/10.1145/2488222.2488283

[37] T. Baker, A. Taleb-Bendiab, M. Randles, A. Hussien, Understanding elasticity of cloud services compositions, in: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 231–232. doi:10.1109/UCC.2012.58.
URL http://dx.doi.org/10.1109/UCC.2012.58

[38] Y. Karam, T. Baker, A. Taleb-Bendiab, Security support for intention driven elastic cloud computing, in: 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation, 2012, pp. 67–73. doi:10.1109/EMS.2012.17.

[39] Y. Karam, T. Baker, A. T. Bendiab, Support for self-optimized organizational patterns in socio-driven elastic clouds, in: 2013 IEEE International Systems Conference (SysCon), 2013, pp. 235–243. doi:10.1109/SysCon.2013.6549888.

[40] T. Baker, A. Taleb-Bendiab, M. Randles, Support for adaptive cloud-based applications via intention modelling, in: The Third International Symposium on Web Services, Zayed University, Dubai, UAE, 2010, 2010, pp. 235–243. doi:10.1109/SysCon.2013.6549888.

[41] R. Mayer, M. A. Tariq, K. Rothermel, Minimizing communication overhead in window-based parallel complex event processing, in: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, ACM, New York, NY, USA, 2017, pp. 54–65. doi:10.1145/3093742.3093914.
URL http://doi.acm.org/10.1145/3093742.3093914

[42] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, P. Valduriez, Streamcloud: An elastic and scalable data streaming system, IEEE Trans. Parallel Distrib. Syst. 23 (12) (2012) 2351–2365. doi:10.1109/TPDS.2012.24.
URL http://dx.doi.org/10.1109/TPDS.2012.24

[43] V. Cardellini, M. Nardelli, D. Luzi, Elastic stateful stream processing in storm, in: 2016 International Conference on High Performance Computing Simulation (HPCS), 2016, pp. 583–590. doi:10.1109/HPCSim.2016.7568388.

[44] T. De Matteis, G. Mencagli, Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, ACM, New York, NY, USA, 2016, pp. 13:1–13:12. doi:10.1145/2851141.2851148.
URL http://doi.acm.org/10.1145/2851141.2851148