Automatic Scaling of Resources in a Storm Topology

Evangelos Gkolemis, Katerina Doka(✉), and Nectarios Koziris

Computing Systems Laboratory, National Technical University of Athens, Athens, Greece
{vgol,katerina,nkoziris}@cslab.ece.ntua.gr

Abstract. In the Big Data era, the batch processing of large volumes of data is simply not enough - data needs to be processed fast to support continuous reactions to changing conditions in real-time. Distributed stream processing systems have emerged as platforms of choice for applications that rely on real-time analytics, with Apache Storm [2] being one of the most prevalent representatives. Whether deployed on physical or virtual infrastructures, distributed stream processing systems are expected to make the most out of the available resources, i.e., achieve the highest throughput or lowest latency with the minimum resource utilisation. However, for Storm - as for most such systems - this is a cumbersome trial-and-error procedure, tied to the specific workload that needs to be processed and requiring manual tweaking of resource-related topology parameters. To this end, we propose ARiSTO, a system that automatically decides on the appropriate amount of resources to be provisioned for each node of the Storm workflow topology based on user-defined performance and cost constraints. ARiSTO employs two mechanisms: a *static*, model-based one, used at bootstrap time to predict the resource-related parameters that better fit the user needs and a *dynamic*, rule-based one that elastically auto-scales the allocated resources in order to maintain the desired performance even under changes in load. The experimental evaluation of our prototype proves the ability of ARiSto to efficiently decide on the resource-related configuration parameters, maintaining the desired throughput at all times.

1 Introduction

In the Big Data era, data is being produced not only in large volumes, but also at an astounding rate. Now more than ever, organizations and companies worldwide heavily rely on the processing of the enormous amounts of data that continuously stream into their businesses to extract significant value out of them: identify new risks and opportunities, take educated decisions based on real-time facts, render their operations faster and more cost efficient and keep customers satisfied [7]. The traditional batch processing model is simply not enough any more, since it fails to cover one of Big Data's most important Vs, that of Velocity. Indeed, data needs to be processed fast, to support timely reaction to changing conditions in

D. Alistarh et al. (Eds.): ALGOCLOUD 2017, LNCS 10739, pp. 157–169, 2018.
https://doi.org/10.1007/978-3-319-74875-7_10

real time. This is of paramount importance in sectors such as trading, fraud detection, system monitoring, healthcare and many others [6,11].

A plethora of distributed stream processing engines have emerged as a remedy to the inability of batch processing systems to provide real-time, interactive responses [1,2,9,17]. Such systems are designed to analyze data in motion, as they stream through the server, contrarily to the traditional batch processing model where data are first stored and then subsequently processed by queries.

Distributed stream processing systems are deployed either on bare-metal clusters, or, most often, over Cloud infrastructures which provide virtual resources in a pay-as-you-go manner. The Cloud Computing model offers the ability to elastically allocate resources, i.e., expand and contract them to meet application needs while keeping the resource budget to a minimum. Whether deployed on physical or virtual infrastructures, distributed stream processing systems are expected to make the most out of the available resources, i.e., achieve the highest throughput or lowest latency with the minimum resource utilisation.

Resource provisioning is one of the most challenging tasks for streaming applications, as it is closely related to the rate of data arrival, which can not be controlled since data are generated by external sources. Over-provisioning of resources will unnecessarily increase resource utilization, hence the cost of running the application. Contrarily, under-provisioning may lead to the inability of the application to keep pace with the velocity of the incoming data stream or comply to the throughput or latency target desired by the user.

However, distributed stream processing engines lack mechanisms to assist the application provider to carefully and correctly provision the required resources, let alone provide on-the-fly adaptation to changing workload conditions [4]. Setting the resource-related parameters for such systems is a cumbersome trial-and-error procedure, tied to the specific workload that needs to be processed and requiring manual tweaking. Moreover, sudden changes in the initial setup, e.g., in the data arrival rate or underlying hardware performance, can not be accommodated automatically, but require manual scaling of the allocated resources.

To this end, we propose methods for the automatic provisioning and online scaling of distributed stream processing resources and design a system that implements them on top of Apache Storm, one of the most prevalent distributed streaming engines. Our system, called *ARiSto* (***A****uto-Scaling* ***R****esources* ***i****n* ***Sto****rm*), automatically and dynamically decides on the appropriate amount of resources to be provisioned for each node of the Storm workflow topology based on user-defined performance and cost/budget constraints. ARiSto employs two mechanisms:

- A *static* mechanism that is used one-off, when first deploying the topology. This mechanism relies on models of the cost and performance characteristics of the required tasks of the topology graph to decide on the (near-)optimal allocation of resources to each one of them.
- A *dynamic* mechanism that is used on-line, as the application executes, to elastically autoscale the allocated resources in order to maintain the desired performance even under changes in load. This rule-based mechanism relies

on the monitoring of each part of the topology to identify bottlenecks and dynamically adjusts the amount of allocated resources to comply to the user-defined performance and cost target.

There has been considerable work in the field of resource management in distributed stream processing systems, however they either emphasize on mechanisms for handling overload without obtaining additional resources on-demand, or focus more on addressing latency-related constraints. Contrarily, our work aims to provide optimal resource allocation and utilization based on high-level or low-level throughput guarantees. Moreover, our solution introduces a novel perspective to the automatic-scaling of distributed stream processing engines by providing re-usability, extensibility and faster initial decisions for the system scaling.

The contribution of this paper is summarized in the following:

- A framework for modeling the cost and the performance characteristics of streaming operators.
- A static, model-based mechanism for deciding on the right amount of resources to be allocated to each operator of the stream processing workflow according to the user-defined performance and cost constraints.
- A dynamic, rule-based mechanism for real-time auto-scaling of the allocated resources according to the monitored cost and performance.
- An open source prototype of our system ARiSto[1], which implements all the above using Apache Storm as the underlying stream processing engine.
- An experimental evaluation that proves the applicability of our methods, which are able to accurately predict the correct amount of resources required for the initial deployment of the streaming workflow and maintain the desired throughput even after sudden changes in load.

2 Preliminaries

Apache Storm [2] is the pedestal of our work. Thus, the understanding of its basic concepts and architecture [3] is of utter importance. In the following we present the basics of Storm and use the terminology introduced throughout the paper.

The core abstraction in Storm is the **stream**, an unbounded sequence of tuples that is created and processed in parallel in a distributed fashion. The tuple is a structure that can contain any kind of data from integers, byte arrays, etc. to custom objects.

A **spout** is a source of streams in a Storm application. Generally, spouts will read tuples from an external source and emit them into the Storm application. All processing in Storm applications is done in bolts. Bolts contain the processing logic and can do anything from filtering, functions, aggregations, joins, talking to databases, etc. A single bolt can perform simple stream transformations, while

[1] https://github.com/vgolemis/aristo.

more complex stream transformations often require multiple steps, i.e., multiple bolts. Bolts can have multiple streams as input and respectively emit more than one stream.

A **topology** is the logical graph representation of the streaming application. More precisely, it is a DAG (Directed Acyclic Graph) with spouts and bolts acting as graph vertices. Unlike a MapReduce job, which eventually finishes, a topology runs continuously, until killed. Part of defining a topology is specifying which streams feed each bolt. A **stream grouping** defines how these streams should be partitioned among the bolt's tasks.

Moving from the logical to the physical level, a subset of a topology is executed by a **worker** process, which runs in its own JVM. A running topology consists of many workers running on many machines within a Storm cluster. Each worker, tied to a specific topology, may run one or more **executors**, i.e., threads. An executor thread is spawned by a worker process and runs within the worker's JVM. Respectively, an executor may run one or more **tasks** for the same component (spout or bolt). A task performs the actual data processing. The number of tasks for a component is always the same throughout the lifetime of a topology, but the number of executors for a component can change over time. This means that the following condition holds true: #executors $\leq$ #tasks.

Another concept of Storm, related to the allocation of resources to each topology component is that of *parallelism* [10]. The parallelism is specified by modifying the following topology configuration parameters: (a) number of worker processes for the topology across all machines in the cluster, (b) number of executors per spout/bolt and (c) number of tasks per executor (the default is one).

All these parameters compose a configuration of the topology. Of these parameters, the number of workers and executors can be configured dynamically (i.e., after the topology is submitted). Contrarily, the number of tasks is static: additional tasks per executor do not increase the level of parallelism, since an executor always consists of one thread that it used for all of its tasks, i.e., tasks run serially on an executor. So, the only reason for having multiple tasks per executor thread is to give the flexibility to scale up the topology without taking the topology offline.

Apache Storm does not offer any mechanism to provide the optimal configuration (parallelism parameters) for user-defined topologies in an automatic manner [4]. There exist some rule-of-thumb guidelines for the parallelism of a topology, drawn from the experience of users:

- If a small number of executors per worker is defined, the cluster resources may not be fully utilized.
- If a large number of executors per worker is defined, resource contention and context switching issues may arise.
- Large number of streams create complexity in the network level.
- Performance is improved if neighbouring components are executed in the same worker because of data locality and reduced network time.
- Multiple workers per machine do not provide additional gains, only flexibility in case of worker failure.

However, these rules are quite vague and can not be used as-is for automatically defining the configuration parameters of a certain topology according to a user-defined optimization metric (e.g., throughput, latency). The user must still define the configuration herself. This can lead to a time-consuming iterative trial-and-error process in which the user runs the topology, manually monitors performance during runtime and evaluates if the configuration satisfies her expectations. Even after multiple iterations, the user may end up in a sub-optimal configuration, which additionally fails to adapt to changing workload conditions due to its static nature.

3 Architecture

ARiSto is a system implemented on top of Apache Storm, that provides automatic configuration of the parallelism parameters of a topology as well as auto-scaling of the provisioned resources in order to continuously comply to the user-defined throughput constraints, even after changes in load.

To this end, ARiSto employs two mechanisms: (a) a *dynamic*, on-line mechanism, that runs alongside the topology and reactively finds the optimal configuration according to the current conditions and (b) a *static*, off-line mechanism, that combines machine learning techniques with genetic algorithms to exploit knowledge from previous runs of the topology to predict the near-optimal configuration.

The common base for both mechanisms is the usage of information about the performance of each component of the topology. This information is retrieved by monitoring topology metrics. Low level component metrics (e.g., #executed tuples, #emitted tuples, latency, component capacity[2] etc.) for a given topology are periodically retrieved and converted to higher level combined metrics (e.g., execution rate, weighted average latencies, topology capacity, etc.) which can either train machine learning models or used in real-time.

The *dynamic* mechanism, given a user-defined target throughput for a topology, keeps track of its monitoring metrics and decides in real-time to adjust one or more parallelism parameters, if necessary. If it reaches the given target, it continues to periodically check to identify possible deviations from the target, due to changes in load or incoming data rate. In order to reach the given target, the mechanism needs to find the best configuration for the topology by resolving the bottlenecks. A bottleneck in a topology is a component that cannot reach the required execution rates, i.e., it cannot process the data at the rate at which they arrive from the previous processing stage. This can happen because the component uses all of its available resources and the only solution is additional parallelism to the specific component. The *dynamic* mechanism follows rules to

[2] The *capacity* metric, which is measured for each topology bolt and takes values between (0,1), represents the percentage of time that a bolt is active (i.e., processing tuples). If this metric approaches 1 the bolt works near its maximum capacity and needs additional parallelism.

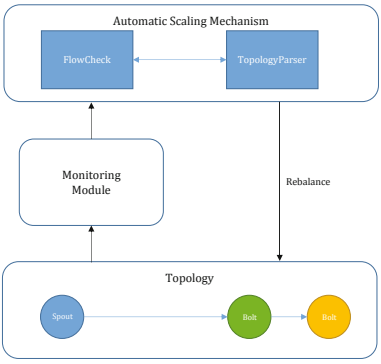

Fig. 1. The architecture of the *dynamic* component of ARiSto

decide how to modify the parallelism parameters of the topology and can be summarized as follows:

- If a component capacity exceeds a threshold th_{cc} then an executor is added for the component.
- If the number of executors per worker exceeds a threshold th_e then we increase the workers of the topology.
- If the capacity of individual components is normal but the total capacity of the topology exceeds a threshold th_{high} then we increase the workers of the topology.
- If the capacity of individual components is normal and the total capacity of the topology is below a threshold th_{low} then we decrease the workers of the topology.

After any change performed in the topology's parallelism parameters we grant a time window for the topology to stabilize (warm up phase). During this window we observe the throughput of the topology and proceed with further changes if required.

The architecture of the *dynamic* module of ARiSto is depicted in Fig. 1. The *TopologyParser* manages the high-level functionalities like invoking the *Monitoring Module*, reading the user configuration and initializing the data structures. After the initialization it is responsible for calling the *FlowCheck* periodically. The *Flowcheck* manages the low-level functionality of the mechanism like checking if the target is reached and changing the parallelism parameters if needed.

The *static* mechanism can be given user level constraints (money, time, etc.) and finds the near-optimal configuration(s) that comply to the user-defined constraints. To be able to predict the best configuration for any given topology the mechanism should be able to predict the performance for every possible configuration. To this end it relies on machine learning and heuristic techniques. The basis of the particular mechanism is machine learning models for the components of the topology. In order to create the models for every component of a topology, each component goes through offline profiling which aims to train the

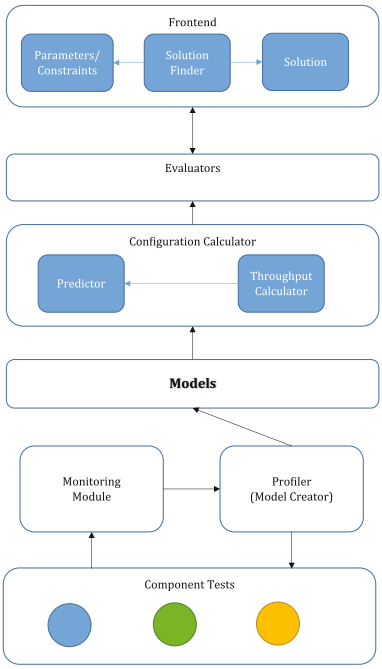

Fig. 2. The architecture of the *static* components of ARiSto

models. More extensive and focused training leads to more accurate models. The choice to create the models at the component level and not the topology level was made to render the mechanism flexible and expandable: Multiple topologies may use the same components, thus, the component models can be used by multiple topologies and enriched. In that manner, we create libraries of component models which are reusable and extendible. After the required models are created, they are used to predict the performance for any given configuration of a topology containing the specific components. In particular, a prediction is given by each model for its corresponding component and all of them are combined to produce the final performance prediction for the configuration of the topology. Subsequently, a heuristic algorithm based on genetic algorithms (NSGA-II in our case) that searches the space of all possible configurations is executed to find the near-optimal one.

Figure 2 presents the architecture of the *static* mechanism of ARiSto. The *Models* are created per component (using the WEKA [12] framework), initially through offline profiling by the *Profiler*. The *Predictor* relies on the *Models* to predict the throughput of individual topology components. The *Throughput Calculator* module calculates the total topology throughput by combining the predictions of individual components, given a specific topology configuration.

The *Evaluators* are those who build the high level constrained problem based on the user level constraints. The *Front-end* is the API we expose. The user

defines her configurations and constraints and the system returns the list of optimal configurations. The *SolutionFinder* is the module that creates the correct evaluator and performs the search in the configuration space.

4 Experimental Evaluation

ARiSto has been implemented in Java and is available under an open-source license. The experimental environment consists of 8 VMs (2 cores/4 GB RAM) in order to create a uniform cluster (1 master, 7 slaves). For the evaluation of our system we assembled 4 topologies with unique characteristics and structure. Our target is to cover a wide variety of different components and flows and observe the behavior of our mechanisms in these different workflows. These include custom made as well as scientific topologies (see Fig. 3):

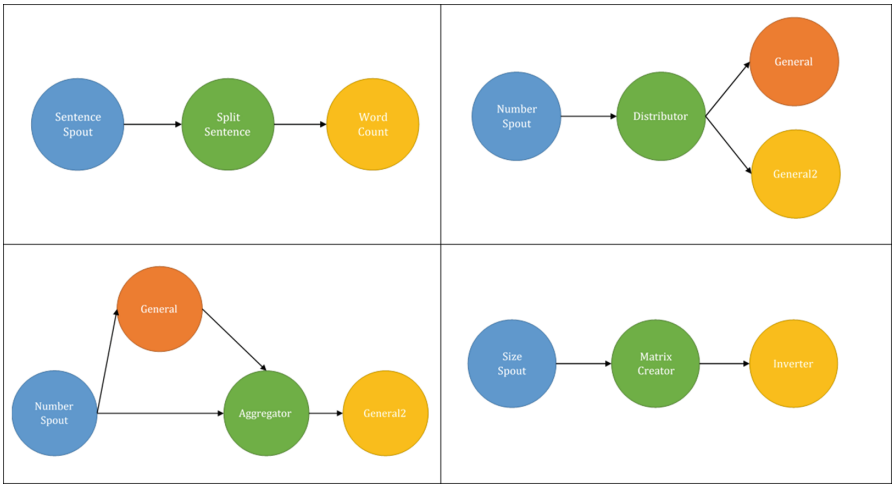

Fig. 3. The topologies of the experimental evaluation

- **WordCount**, one of the fundamental examples in distributed environments. (CPU intensive, Network intensive workflow)
- **CyberShake**, a scientific workflow used by the Southern California Earthquake Center to characterize earthquake hazards in a region. (Network intensive workflow)
- **Montage**, a scientific workflow created by NASA/IPAC stitches together multiple input images to create custom mosaics of the sky. (CPU intensive, Network intensive, Memory intensive workflow)
- **Matrix**, a custom-made workflow that performs matrix operations. It is customizable and easily expandable. (CPU intensive, memory intensive workflow)

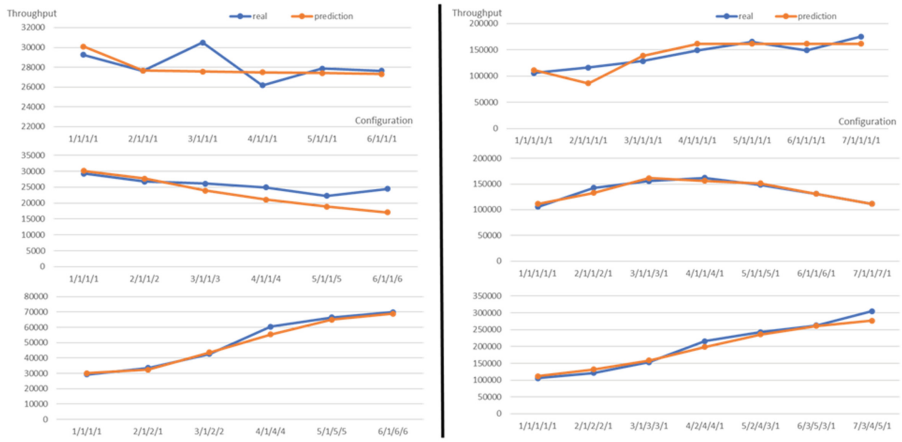

Fig. 4. Evaluating the total topology throughput when varying the #workers, #executors and both (vertically) for the Wordcount and the Montage Topologies (horizontally)

We evaluate ARiSto in terms of sensitivity to the parallelism-related configuration parameters, efficacy of the *static* mechanism and the accuracy of its predictor module.

Performance impact when varying the parallelism parameters. In the first set of experiments (first row of Fig. 4), we gradually increase the workers without changing anything else in the configuration. For the WordCount Topology we can see that by adding a worker to the initial configuration we lose in terms of performance. This happens because we lose the data locality (if all the components are executed in the same worker we avoid over-the-network data transfer). From that point on, additional workers do not improve the performance because it is bound by the main performance bottleneck i.e., the SplitSentence bolt. For the Montage Topology, as we provide more workers we can see a slight performance increase. This is due to the fact that the workflow is more cpu- than network-intensive.

In the second set (second row of Fig. 4), we gradually increase the number of executors for one of the components alongside the worker increase. For the WordCount Topology we increase the WordCount component executors which is not the bottleneck component. We can see that the topology performance immediately deteriorates. This happens because we increase the executors of a component which was not overloaded and we thus create additional network complexity for the SplitSentence component, which is the actual bottleneck. For the Montage Topology we increase the Aggregator component executors, which is one of the initial bottlenecks. We can see a gradual improvement in the topology performance but after several steps the performance starts to fall. This happens because we initially solved the cpu bottleneck (Aggregator), but by adding more executors of the Aggregator component we created a network bottleneck in the previous processing stage.

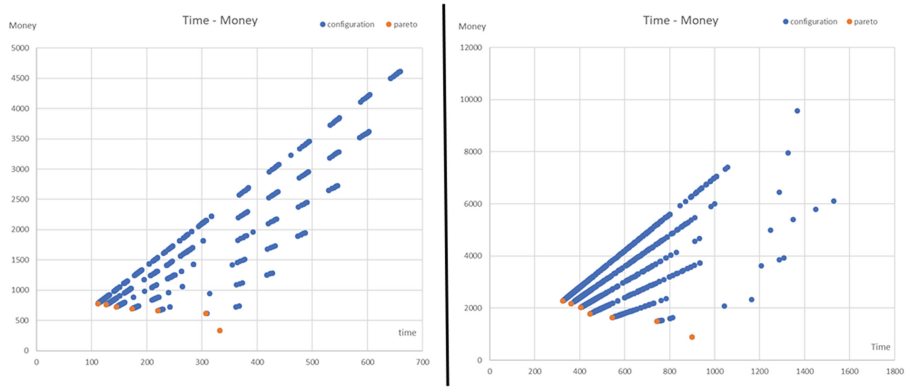

Fig. 5. The time-cost plot for all possible configurations for the Wordcount and the Montage Topologies (Color figure online)

In the third set (third row of Fig. 4), we gradually move to better configurations that simultaneously solve all the major bottlenecks that the topology is facing. For the WordCount we add executors for the SplitSentence component, which solve the cpu bottleneck and at the same time we add executors for the WordCount component, which improve the data locality and network complexity. For the Montage Topology we add executors for the Aggregator component, which solve the cpu bottleneck and at the same time we add executors for the General1 component which solve the network bottleneck and provides better data locality.

Efficacy of the static mechanism. For the 2 topologies presented above, all possible configurations are plotted in a time-cost plot (time and cost are two of the most important user constraints) in Fig. 5:

The orange dots are the configurations selected by ARiSto and they coincide with the pareto optimal configurations. Pareto optimal is a configuration that offers the best trade-off between time and cost.

The static mechanism is also evaluated based on its prediction accuracy. The accuracy is calculated as the difference between the predicted and actual performance for a given topology and configuration. The accuracy evaluations of Fig. 6 concern the CyberShake and Montage Topologies respectively. We can see that the mechanism has a discrepancy below 10% in approximately 60% of its predictions and below 20% in approximately 90% of its predictions.

The desirable behavior of the mechanism is to be very accurate in regions of global performance maxima. The regions of local/global minima as well as the majority of local maxima are not of such importance. The predictor we implemented generates very accurate predictions for the global maxima and the transitional regions, is less accurate for local maxima and purposefully underestimates the global minima and part of the local minima. This is the reason that we have a percentage of prediction with high discrepancy. The decision to underestimate

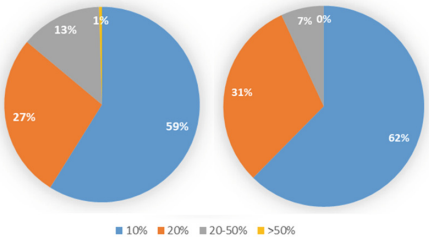

Fig. 6. The accuracy of the static mechanism for the CyberShake and Montage topologies

the local/global minima was made in order to have a faster search in the configuration space by the SolutionFinder module. When it searches a region that is dramatically below the performance the heuristic algorithm will not search again in the particular region. Additionally, solid and accurate transitional regions will help the algorithm find the local and global maxima.

5 Related Work

Distributed stream processing has been an active area of research over the last decade, when the need for real-time analytics over vast amounts of data became more prominent and called for scalable streaming engines that could handle them. Several such engines emerged, either proprietary such as S4 [8], MillWheel [13] and DataTorrent [5] or open source, such as Storm [2], Heron [17], Spark Streaming [9] and Flink [1]. None of these engined inherently provide auto-scaling capabilities to meet user-defined performance constraints with minimum amount of provisioned resources.

Works related to resource management in distributed stream processing systems mostly emphasize on mechanisms for handling overload: load-shedding [20], admission control [21], adaptive query planning [19], load balancing [22] and efficient initial operator placement [16] do not address overload by obtaining additional resources on-demand.

The concept of elastic scaling of distributed stream processing systems has been studied in recent works. However, they focus on the latency aspect of the system following different directions like optimizing operator movement during scale-in/out of the system to eliminate latency spikes [15] or using reactive strategies to enforce latency guarantees in scalable stream processing systems [18]. ARiSto aims to provide optimal resource allocation and utilization based on throughput guarantees. Moreover, our solution follows a novel, model-based approach which provides reusability, extensibility and faster initial decisions for the configuration of the resource-related parameters.

The work closest to ours is the very recently published Dhalion [14], a system deployed on top of Heron [17], which provides with self-regulating capabilities

through the execution of various Dhalion policies. One of them is to automatically scale up and down resources based on the input data and another to autotunes the topology by provisioning the necessary resources to meet a throughput SLO. Dhalion employs only rule-based methods, while ARiSto also follows a model-based approach, which can be used when first deploying a topology to speed up the process of auto-scaling.

6 Conclusions

In this paper we presented ARiSto, a system that provides automatic scaling of resources in Apache Storm in order to comply to the user-defined performance and cost constraints, by virtue of two mechanisms, a static and a dynamic one. The static mechanism relies on performance an cost models of the streaming operators that are involved in the workflow topology and based on genetic algorithms it decides on the exact amount of resources to be allocated to each one of them. The dynamic mechanism is able to adjust to current load conditions, maintaining the desired throughput/latency with minimum cost (i.e., resources). This is achieved by monitoring all topology entities, identifying the bottlenecks and adding/subtracting resources according to a set of rules. The evaluation of our prototype implementation showcases the ability of our system to efficiently decide on the resource-related configuration parameters, maintaining the desired throughput at all times.

References

1. Apache Flink. https://flink.apache.org/
2. Apache Storm. http://storm.apache.org/
3. Apache Storm Concepts. http://storm.apache.org/releases/1.0.0/Concepts.html
4. Auto-Scaling Resources in a Topology. https://issues.apache.org/jira/browse/STORM-594
5. DataTorrent. https://www.datatorrent.com/
6. Healthcare Looks to Real-Time Big Data Analytics for Insights. https://healthitanalytics.com/news/healthcare-looks-to-real-time-big-data-analytics-for-insights
7. Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse. https://www.infoq.com/articles/stream-processing-hadoop
8. S4 Distributed Data Platform. http://incubator.apache.org/s4
9. Spark Streaming. https://spark.apache.org/streaming/
10. Understanding the Parallelism of a Storm Topology. storm.apache.org/releases/1.0.0/Understanding-the-parallelism-of-a-Storm-topology.html
11. Use Cases for Real Time Stream Processing Systems. https://dzone.com/articles/need-for-using-real-time-stream-processing-systems
12. Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/
13. Akidau, T., et al.: Millwheel: fault-tolerant stream processing at internet scale. In: VLDB 2014, vol. 6, no. 11, pp. 1033–1044 (2013)
14. Floratou, A., et al.: Dhalion: self-regulating stream processing in heron. In: VLDB 2017 (2017)

15. Heinze, T., et al.: Latency-aware elastic scaling for distributed data stream processing systems. In: DEBS 2014, pp. 13–22. ACM (2014)
16. Kalyvianaki, E., Wiesemann, W., Vu, Q.H., Kuhn, D., Pietzuch, P.: SQPR: stream query planning with reuse. In: ICDE 2011, pp. 840–851. IEEE (2011)
17. Kulkarni, S., et al.: Twitter heron: stream processing at scale. In: SIGMOD 2015, pp. 239–250. ACM (2015)
18. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In: ICDCS 2015, pp. 399–410. IEEE (2015)
19. Markl, V., et al.: Robust query processing through progressive optimization. In: SIGMOD 2004, pp. 659–670. ACM (2004)
20. Tatbul, N., et al.: Load shedding in a data stream manager. In: VLDB 2003, pp. 309–320. VLDB Endowment (2003)
21. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., Fleischer, L.: SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 306–325. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89856-6_16
22. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic load distribution in the borealis stream processor. In: ICDE 2005, pp. 791–802. IEEE (2005)