Auto-scaling Techniques for Elastic Data Stream Processing

Thomas Heinze[1], Valerio Pappalardo[1], Zbigniew Jerzak[1], Christof Fetzer[2]

[1] *SAP AG*
Chemnitzer Str. 48, 01069 Dresden, Germany
{firstname.lastname}@sap.com

[2] *TU Dresden, Systems Engineering Group*
Noethnitzer Str. 46, 01187 Dresden, Germany
christof.fetzer@tu-dresden.de

***Abstract*—An elastic data stream processing system is able to handle changes in workload by dynamically scaling out and scaling in. This allows for handling of unexpected load spikes without the need for constant overprovisioning. One of the major challenges for an elastic system is to find the right point in time to scale in or to scale out. Finding such a point is difficult as it depends on constantly changing workload and system characteristics. In this paper we investigate the application of different auto-scaling techniques for solving this problem. Specifically: (1) we formulate basic requirements for an auto-scaling technique used in an elastic data stream processing system, (2) we use the formulated requirements to select the best auto scaling techniques, and (3) we perform evaluation of the selected auto scaling techniques using the real world data. Our experiments show that the auto scaling techniques used in existing elastic data stream processing systems are performing worse than the strategies used in our work.**

I. Introduction

Data stream processing systems are designed to process high velocity big data. They are used in various scenarios, including: algorithmic trading [1] or monitoring of key performance indicators in the manufacturing domain [2]. The processing of information using data streaming systems is achieved by executing a set of standing queries over unbounded streams of data. Unlike in batch-oriented systems [3], [4], query results in data stream processing systems are returned continuously as soon as they become available. To ensure the required low latency and high throughput, data streaming systems are typically implemented as distributed systems.

A classical distributed data stream processing system [5], [6] uses a fixed number of hosts, which is chosen to meet the expected maximal workload. However, due to the fact that peak loads only occur from time to time, in average the system is mostly underutilized. Ideally, the system should automatically acquire new processing nodes or release existing nodes to match the workload. Such systems are called elastic [7]. Elastic scaling systems are especially beneficial for the end user when running within a public cloud environment with a pay per use model.

Various authors studied the problem of designing elastic data stream processing systems. Most of them focused on the underlying problems like an efficient operator state management [8] or the coordination of large processing clusters for an elastic streaming system [9], [10].

An important problem faced by an elastic streaming system is to decide when the system has to scale in or out. Auto-scaling techniques [11] derive scaling decisions based on an algorithmic analysis of current workload parameters. They try to avoid situations of unexpected overload and limit the number of scaling decisions needed to meet a given utilization target. Many different auto-scaling techniques [11] have been proposed in the context of cloud databases or application layer scenarios. However, to the best of our knowledge no other work has yet investigated the usage of different auto-scaling techniques within data stream processing systems. Data streaming use cases create new challenges for auto-scaling techniques: (1) an unpredictable event workload prohibits the usage of fixed workload models (2) fast changing workload requires the auto-scaling technique to respond within seconds (3) due to inhomogenous load changes between hosts, the system needs to decide for each host individually. Therefore, established auto-scaling techniques need to be carefully investigated and adapted in order to be used within an elastic data stream processing system.

For this paper we have implemented three auto-scaling strategies: (1) global thresholds, (2) local thresholds as well as (3) reinforcement learning. They are based on different established algorithmic techniques [11] used within non-streaming elastic cloud systems. The selection of the above techniques is based on the fulfillment of the requirements presented in Section III. The chosen techniques were implemented on top of a state of the art elastic streaming system prototype [10] presented in Section II. Using real-world data in the evaluation (see Section V) we compare to which extent the selected techniques can fulfill the requirements of elastic data stream processing systems. We show that reinforcement learning is the best alternative, being robust and adaptive. This is a clear contrast to existing related work relying on global [9] or local thresholds [8].

II. Background

The presented auto-scaling techniques are implemented on top of an elastic data streaming processing engine *FUGU* [10]. It consists of a centralized management component dynamically allocating a varying number of hosts. *FUGU* is executed on top of a commercial, state of the art distributed streaming engine.

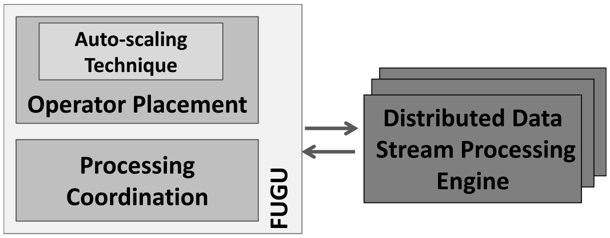

Fig. 1: Architecture of *FUGU*

The centralized management component (see Figure 1) serves two major purposes: (1) it derives placement decisions using an operator placement component and assigns operators to hosts, including decisions to allocate new hosts or release existing hosts and (2) it coordinates the construction of the operator network within the distributed CEP engine.

For the operator placement *FUGU* constantly monitors all running operators in the system and measures CPU, RAM and network consumption for each of them periodically. Based on these measurements the used auto-scaling technique decides if a host/system is over- or underloaded. For overloaded hosts a subset of their operators is selected using a subset sum algorithm [12] and re-assigned to not overloaded hosts.

The assignment of operators itself is done using a bin packing approach [13], where hosts are modeled as bins and operators as items. An operator can only be assigned to a host, if its CPU load is less than the available capacity on the host. As sub-constraints, enough network and RAM capacity need to be available on the host. The bin packing problem is solved using a First Fit heuristic [13]. It first sorts the operators based on the CPU load and assigns them in decreasing order on any host with enough capacity. The heuristic is enhanced with a priority for preferring certain hosts in case neighboring operators of the selected operator are placed there. The placement is executed in an incremental way. All operators marked for movement by the auto-scaling technique are used as input for the bin packing approach, the position of the remaining operators is fixed.

For each migration of a stateful operator, its state needs to be transfered to the new host [14]. Therefore, the state of the moved operator is extracted [8] and moved to the new destination of the operator. Before starting the operator on the new host, the state is replayed. During the migration the predecessor operator is paused to avoid the loss of events. A more detailed description of the used state migration protocol is given in [14].

Each host works autonomously and only knows the centralized manager. If different operators of the same query are executed on different hosts, each host deploys its operators individually and asks the centralized manager for the location of successor and predecessor operators. As a response, *FUGU* provides valid locations of the requested operators and, thereby, allows the construction of point to point connections between individual operators. *FUGU* also informs the processing nodes about changes of the topology due to operator movements.

An auto-scaling technique is used as a decision kernel within *FUGU*. Periodically, *FUGU* triggers the auto-scaling algorithm with up to date system information. As input for the auto-scaling technique the current utilization of each host is used. The output of the auto-scaling technique can be either: scale out, no action or scale in.

III. Requirements for an auto-scaling technique

For selecting possible approaches from the set of available auto-scaling techniques [11] we propose a set of requirements for an auto-scaling technique in an elastic data streaming engine:

Workload Independence: the solution has to be independent from the workload characteristics; we make no assumption on the input workload, for example, we do not want to adjust the solution to a specific workload pattern or stochastic model of the workload.

Adaptivity: the system has to be able to adapt online to changing conditions like different workload characteristic, therefore, the system should be learning using online feedback.

Configurability: the scaling strategy has to be easy to set up and configure by an end user.

Computational feasibility: the algorithm has to be computationally feasible in a soft real-time environment. In contrast to a classical database or a webserver, the workload for streaming system changes within seconds. State of the art streaming systems [8], [9], [10] can scale out and in within seconds to accommodate such changes. An auto-scaling strategy has to have a computational complexity small enough to be able to response in such small timescales.

Existing auto-scaling approaches can be categorized based on the underlying algorithmic techniques into five major groups [11]: (1) threshold-based approaches, (2) time series analysis, (3) reinforcement learning, (4) queuing theory and (5) control theory. Predicting workload using a time series analysis based on historical data or workload pattern is not feasible, because usually event rates and the data distribution for a data stream processing system change in an unpredictable way. Therefore, we decided to exclude time series analysis algorithms from the paper. A queuing theory approaches are based on a detailed system model and have a limited adaptivity [15]. To find such a model for an elastic streaming system is very complicated due to the unknown workload characteristics. For the remaining three classes we selected either existing techniques for data stream processing systems or approaches used in other domains, but fulfilling the mentioned requirements:

Threshold-based approaches: Threshold-based rules derive placement decisions using simple rules. Therefore, the user needs to define an upper as well as a lower bound for the system utilization. The decisions can be taken very fast, independent of the workload and they are configurable by a user. However, the approach is not adaptive, the thresholds

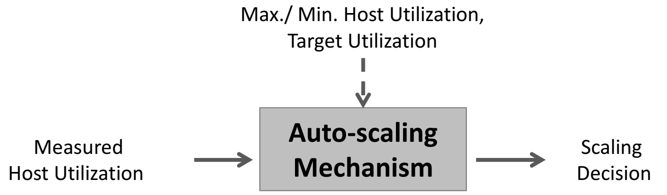

Fig. 2: Input/Output of the different auto-scaling techniques

are fixed for the complete runtime of the system. We present a threshold-based approach as a baseline, because it has been used by existing scale out data stream processing systems: Guilsano et al. [9] defined an upper threshold for the load variance between all hosts and Fernandez et al. [8] defined an upper as well as lower threshold for each individual host. Cervino et al. [16] provided an algorithm, which determined the required number of virtual machines based on the current input rate and a maximal capacity of an individual host.

Reinforcement learning: Reinforcement learning describes a set of approaches, which decide based on the current state of the system for the action, which is expected to yield the biggest improvement of the current system state. As a response, the data stream processing system provides a feedback describing the result of the used scaling action. The feedback allows the algorithm to change its behavior and choose another action next time.

Control theory: In general, a controller is a good candidate for elastic data streaming: it can be designed to be independent of the workload, it responses very fast to a changing input and the system can become adaptive by using a feedback loop. This approach could be an alternative to the reinforcement learning presented in this paper. A comparison between both approaches is left for future work.

IV. Implemented Auto-scaling techniques

In the following we describe the selected auto-scaling techniques. The goal of the auto-scaling technique in our system is to maximize the system utilization and at the same time to guarantee a low end to end latency. Both objectives are conflicting: if the utilization is maximized, the end to end latency will increase due to more frequent migration decisions. In contrast, optimizing for a stable end to end latency will limit the achieved utilization by avoiding overload situations and too frequent migration decisions.

We have implemented three different variants of auto-scaling mechanisms: (1) a local threshold-based approach, (2) a global threshold-based approach and (3) an adaptive policy using reinforcement learning.

All three auto-scaling techniques are configured using only three parameters: (1) a maximal host utilization, (2) a minimal host utilization, and (3) a target utilization. All other parameters, e.g., a learning rate for the reinforcement learning, are used with pre-defined values (if not mentioned differently).

A. *Local Threshold-based Approach*

When using a threshold-based approach as soon as the auto-scaling technique is triggered the system utilization is checked against an upper and lower threshold. In case the upper threshold is exceeded for a set of n successive measurements the system is marked as overloaded.

A host is marked as underloaded, if the utilization drops below the lower threshold for n consecutive measurements. In this situation *FUGU* tries to find a new host for all currently running operators on the underloaded host. In case it is successful, the host is released. In case not enough free capacity on the remaining hosts exists, the release is canceled and no operator movement is done.

Certain actions are taken to avoid too frequent scaling decisions: (1) after each scaling decision the affected host is not touched for a certain period of time, called grace period, (2) scaling decisions are only done after n consecutive violations of the threshold, (3) scaling out decisions the load to move will be chosen larger so as to get done to the target utilization.

B. *Global Threshold-based Approach*

In contrast to the local thresholds, the global threshold-based approach is defined based on the average load of all running hosts. In case this average load exceeds an upper limit the system is marked as overloaded and for each individual host an amount of load to move is calculated. In case the lower threshold is exceeded the host with the minimal load is released and all its operators are redistributed.

A global threshold will trigger less frequent scaling decisions. This has the advantage of less operator movements, but also results in more overload situations for individual hosts (see Section V), which in turn negatively impact the latency.

C. *Reinforcement learning*

The reinforcement learning approach [15] builds up a lookup table for deciding which action to take based on historical feedback. A table entry describes for a given utilization value the expected reward value for different actions. Possible actions include scale out, scale in and no action. Based on the current state, always the action with the highest reward is chosen.

The lookup table is updated using an online learning based on the reward received by the system. Therefore, the so-called Sarsa(0) algorithm [15] is used, which updates the reward for the previous state and action $Q(s_{i-1})$ based on an immediate reward r and the expected future reward for the current state $Q(s_i)$:

$$Q(s_{i-1}) = (1-\alpha(s_{i-1}))Q(s_{i-1}) + \alpha(s_{i-1})(r + \beta \cdot (Q(s_i)))),$$

where $\alpha(s_{i-1})$ and β are learning factors describing the influence of the immediate and the future reward respectively. The immediate reward is calculated based on the utilization error, which describes the difference between measured utilization and target utilization. If by an action the utilization error between $Q(s_{i-1})$ and $Q(s_i)$ decreases, the immediate reward is larger than 1. Similarly, if the error increases, the immediate reward is smaller than 1. Within this work we use as learning rate $\alpha(s_i)$ a monotonic decreasing function, which decreases with the number of visits of the state [15]:

$$\alpha(s_i) = \ imp \cdot \ (\frac{dur}{dur + visits(s_i)}),$$

where $visits(s_i)$ describes the number of learning steps the system has done for the current state, imp is the impact of the learning and dur shows how long the system is learning. The default values used in our system are $imp = 0.2$ and $dur = 80$ (see Section V-C). The learning of the system is accelerated by a soft monotonicity constraint [15], which requires that the system behaves in a monotonic way. If the system decides for the current threshold and latency combination to scale out, it should not derive a scale in decision for an even larger utilization and latency value. This monotonicity is enforced after each update of the reinforcement table.

We have extended this basic algorithm to meet the characteristics of an elastic data streaming engine:

- *Local Policy:* within our prototype each host an individual lookup table is maintained, which is created on the allocation of the host and deleted on the host release. As a consequence, for each host an own scaling strategy based on its own load characteristics is learned.
- *Initialisation:* we initialize the lookup table based on a local elasticity policy. Below the predefined lower threshold the highest reward is assigned to scale in, between the lower and the upper threshold the highest reward for no action and above the upper threshold for scale out.
- *Learning algorithm:* certain (wrong) decisions of the reinforcement learning can not be enforced by the underlying bin packing, e.g., in case a host with a load of 0.8 should be released, typically not enough resources are available on the remaining hosts to move the operators of this host. In this situation the scaling decision are canceled and the system state is not changed. However, we modified the reinforcement to receive as immediate feedback on the next learning step a negative penalty to reduce the reward of the taken action.
- *Grace period:* similar to the threshold-based approach we enforce a fixed grace period. The learning for a previous scaling decision is done using the (stable) utilization after the grace period finished.

The following example demonstrates the implementation of our reinforcement learning approach. In Table Ia a part of the lookup table for an individual host after the initialization is shown using a target utilization of 0.6. A row represents the expected rewards for a given utilization value for the three possible actions. Assuming *FUGU* now triggers the reinforcement learning with a current utilization of 0.7, the reinforcement learning decides to take no action (NA), because this action has the highest expected reward. If the next measured utilization for this host is 0.9, it indicates that the scaling decision increased the utilization error from $|(0.6 - 0.7)/0.6| = 0.16$ to $|(0.6 - 0.9)/0.6| = 0.5$. As result the calculated immediate reward equals to $r = 1.0 + (0.16 - 0.5) = 0.66$. The Sarsa learning algorithm updates the entry to $(1.0 - 0.2) \cdot 0.85 + 0.2 \cdot ((0.66 + 0.1 \cdot 0.5) = 0.822$ (see Table Ib), if it is the first visit of this entry and $\alpha(s_{0.7}) = 0.2$ as well as $\beta = 0.1$. As result, the difference between the reward for no action and scale out shrinks and after the next learning steps scale out might become the action with the highest reward for a utilization of 0.7.

Util	**ScaleIn**	**NA**	**ScaleOut**
0.7	0.4	0.85	0.8
0.8	0.38	0.7	0.85
0.9	0.35	0.5	0.9
1.0	0.32	0.4	1.0

(a) Before Learning

Util	**ScaleIn**	**NA**	**ScaleOut**
0.7	0.4	0.822	0.8
0.8	0.38	0.7	0.85
0.9	0.35	0.5	0.9
1.0	0.32	0.4	1.0

(b) After Learning

TABLE I: Example for reinforcement learning

V. Evaluation

The goal of the evaluation is to show to which extent the different approaches are able to fulfill the requirements presented in Section III:

- *Configurability*: we compare the performance of all three auto-scaling techniques using different configurations.
- *Workload independence*: Using the best configuration we test the robustness of each approach for three different workloads.
- *Adaptability*: We show the influence of the learning rate on the reinforcement learning approach.

In our evaluation we use three real-world data sets from the Frankfurt stock exchange. The event rates for each data set change dynamically like presented in Figure 3. Each data set represents the tick load of one working day from the Frankfurt stock exchange. The data rate are changing very fast, which requires the auto-scaling technique to also derive decision very fast. All used auto-scaling techniques have a small enough complexity to achieve this. We run each experiment with a fixed number (35) of continuous queries. For evaluating our system we use the following query template:

```
SELECT FIRST(price),MIN(price),AVG(price),
MAX(price),LAST(price) FROM tickStream
WITHIN x SEC GROUP BY comp WHERE sector=y;
```

The above query calculates the average, minimal, maximal, first and last price for each company within a certain sector. The query workload is made variable by choosing the window size (x) and the sector (y) randomly.

FUGU is deployed on top of a private, shared cloud using up to 10 processing nodes. The system automatically allocates or deallocates hosts as instructed by the selected auto-scaling technique.

For each run we collect the data for about 700 measurement points. Each experiment lasts for at least an hour. We measure

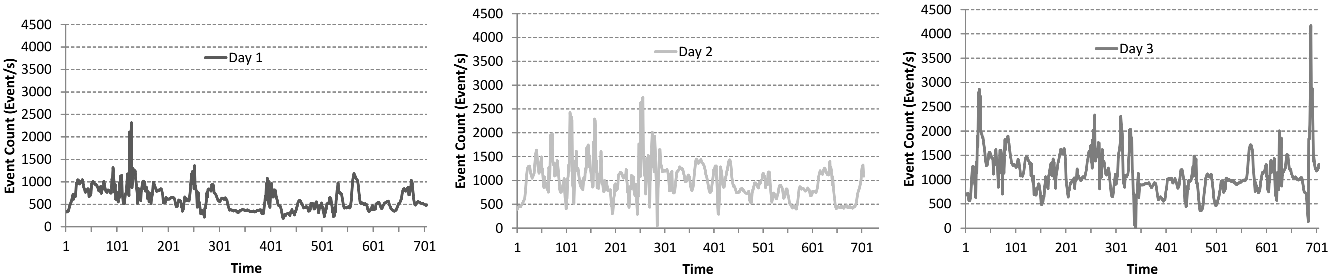

Fig. 3: Financial workload from Frankfurt stock exchange

both the utilization and the end to end latency averaged over all queries running in the system. The minimal host load, the maximal host load as well as the average host load are computed as average over all 700 measurement points. For the end to end latency we present the median, 10th percentile as well as the 90th percentile of all measurement. A measurement point is calculated as the average query latency of all running queries. The 10th percentile and median are indicators for the normal processing latency, the 90th percentile shows the impact of overload as well as scaling decisions.

A. *Configurability*

As first experiment we run all three auto-scaling techniques with the same configuration. We vary the parameters to study their influence on the utilization and latency. We use workload $Day1$ and compare for each auto-scaling technique five configurations with different lower threshold $t_{\downarrow}$ and upper threshold $t^{\uparrow}$:

- Default: $t_{\downarrow} = 0.3$ and $t^{\uparrow} = 0.8$
- Decreased Lower Threshold: $t_{\downarrow} = 0.2$ and $t^{\uparrow} = 0.8$
- Increased Lower Threshold: $t_{\downarrow} = 0.4$ and $t^{\uparrow} = 0.8$
- Decreased Upper Threshold: $t_{\downarrow} = 0.3$ and $t^{\uparrow} = 0.7$
- Increased Upper Threshold: $t_{\downarrow} = 0.3$ and $t^{\uparrow} = 0.9$

As target utilization we use 0.6. The results for using local thresholds are presented in Figure 4. For the default configuration the system achieves an average utilization of 0.52 and a 90th percentile latency of 1850 ms. If the lower threshold is decreased, also average utilization and average latency decrease. An increased lower threshold shows the opposite behavior due to more frequent operator migrations. For both varying upper threshold the latency is increasing significantly. The upper threshold 0.7 is too close to the target utilization and thereby results in many operator migrations. In contrast, the upper threshold of 0.9 result in many overload situations, which also increases the end to end latency.

The global threshold shows a comparable performance for the default configuration with an average utilization of 0.49 and a 90th percentile for latency of 2340 ms (see Figure 5). However, for all remaining configurations it has a significantly worse latency compared to the local threshold approach. Due to the inhomogeneous load of different hosts overloads of individual hosts can not be detected by a global threshold and result in an increased latency. For all configurations the average maximal host utilization exceeds 0.7, where else for local thresholds it is always below this value.

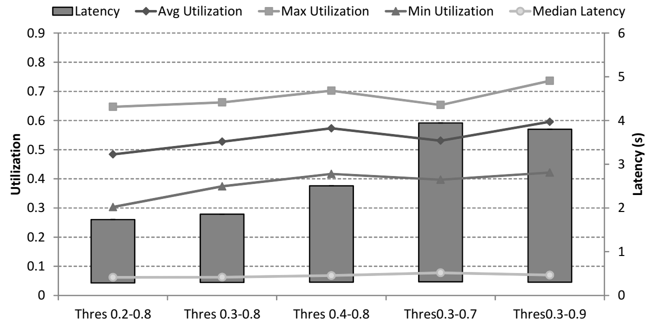

Fig. 4: Results for different configurations for Local Thresholds

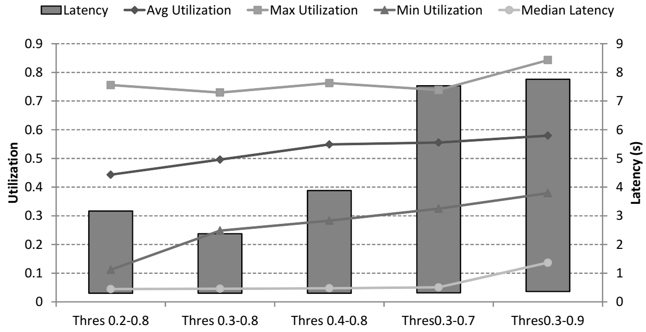

Fig. 5: Results for different configurations for Global Thresholds

The results for reinforcement learning are shown in Figure 6. The reinforcement learning is able to achieve for all configurations similar utilization and latency values. The utilization varies between 0.56 and 0.62, the average latency between 1300 and 2400 ms. The achieved average utilization improves by at least 0.05 compared to the local threshold approach with a same or even smaller 90th percentile latency. This improvement is substantial, given the small number of hosts used and the discrete operator loads. Assuming we rented our cloud infrastructure from Amazon EC2, a 5%

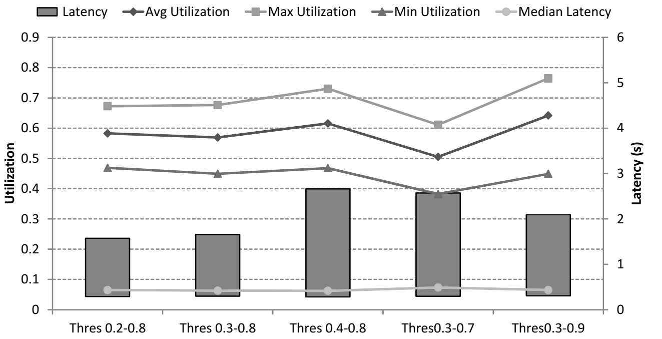

Fig. 6: Results for different configurations for Reinforcement Learning

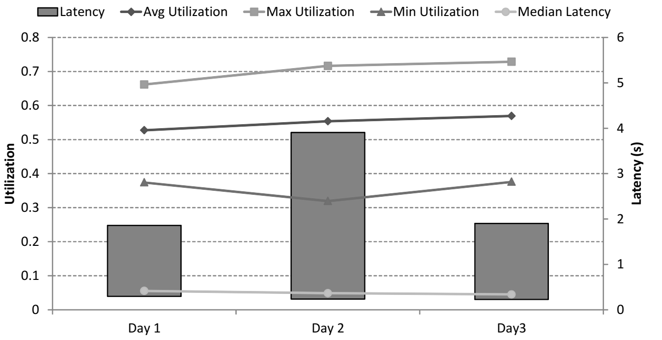

Fig. 7: Results for different workload for Local Thresholds

improvement in utilization would result in \$500 saving per 10 host cluster of m1.medium instances within a period of one year. This is caused by the decrease in the average number of used hosts for our scenario by 0.5 hosts.

In addition, the achieved utilization improvement stems solely from the changes in the scaling strategy and not from the changes to the placement algorithm. Specifically, an improved (as compared to the currently used bin packing) placement strategy could further increase the achievable system utilization.

There are two reasons for this good performance of the reinforcement learning: (1) the thresholds are adapted based on the online learning and (2) negative penalty values prevent too frequent scale in decisions. In addition, in order to maximize the learning effect, we use a coarse granularity reinforcement table. We use a step size of 0.05 which speeds up the learning and due to the resulting rounding disambiguates the scaling decisions.

B. Workload independence

We evaluate the robustness of the local thresholds and the reinforcement learning by comparing with the default configuration used in the previous section. Therefore, we run the local thresholds with a lower threshold of 0.3 and an upper threshold of 0.8 for all three workloads $Day1$, $Day2$ and $Day3$ (see Figure 7).

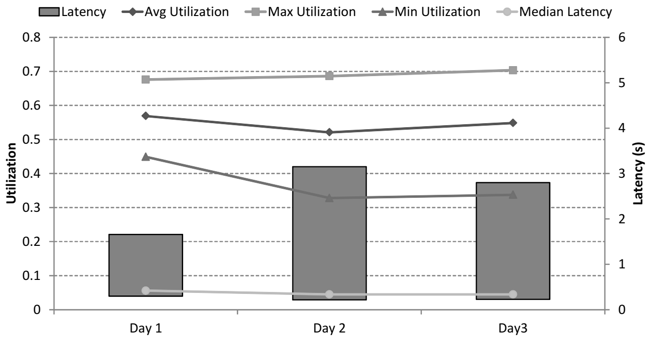

Fig. 8: Results for different workload for Reinforcement Learning

The achieved utilization as well as latency varies significantly for the local threshold based policy. Especially, for $Day2$ the achieved latency is for the same configuration nearly twice as high as for $Day1$. In contrast, the reinforcement learning shows more stable results especially in the achieved latency (see Figure 8). The reinforcement learning adapts to the changed workload characteristics and thereby shows better performance values.

C. Adaptivity

Finally, we tested the adaptivity of the reinforcement learning. We influence the learning of the system by varying the impact of the learning imp as well as the duration dur (see Section IV-C). In total we tested four different configurations:

- No Learning: $imp = 0.0$
- Default Learning: $imp = 0.2$ and $dur = 80$
- More intensive Learning: $imp = 0.5$ and $dur = 80$
- Longer Learning: $imp = 0.2$ and $dur = 160$

As configuration we use a lower threshold of 0.3 and an upper threshold of 0.8. The results are presented in Figure 9.

Our default learning configuration improves the utilization of the system compare to an disabled learning from 0.55 to 0.58. The latency decreases from 2100 to 1700 ms. Both increasing the learning duration as well as the learning impact results in a worse utilization as well as latency. The loss for an increased impact is larger than the loss for an increased learning duration. Depending on the workload, the learning will either arrive sooner or later in a stable state and show better or worse performance. However, it has to be noted that all different learning configurations still show better performance than a local threshold approach.

D. Discussion

Based on the results of our evaluation, we conclude that a global threshold-based solution is not really applicable for data stream processing system. A correctly chosen local thresholds shows a good performance, but for a wrongly configured the system performance decreases rapidly. The reinforcement learning is a good alternative, which provides a robust and adaptive solution for the auto-scaling problem. However, the

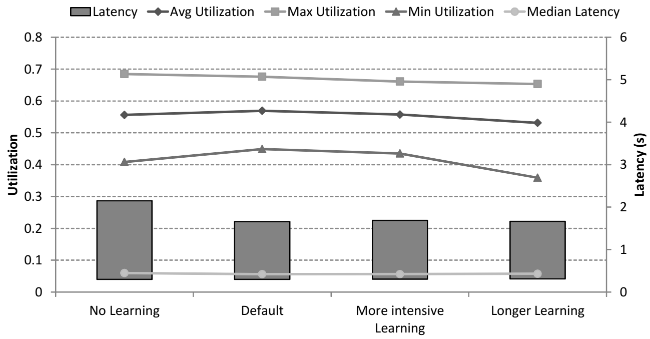

Fig. 9: Results for different learning configurations for Reinforcement Learning

parameters influencing the learning need to be carefully chosen to avoid too fast learning.

VI. Summary

Efficient scaling in and scaling out is a major challenge of modern elastic data stream processing systems. Only using intelligent scaling strategies these systems can ensure stable latency as well as good system utilization at all time, despite constantly changing workload. In this paper we have formulated four requirements for using auto-scaling techniques within data stream processing systems. We have also compared different approaches based on a scenario using real-world data. From our evaluation we conclude that global thresholds are not usable for data stream processing systems and that an approach based on reinforcement learning shows the most robust and adaptive performance.

For the future work we would like to investigate the influence of additional parameters on the auto-scaling technique. Especially, we would like to study how the end to end latency can be reflected more explicitly within our system. Two different aspect could be touched (1) how the latency could be incorporated e.g. into the reinforcement learning and (2) if the expected latency peak of a scaling decision can be estimated and used to derive better scaling decisions. Other important factors could be the queue length of an operator or the current event rate.

In addition, an investigation of controller-based techniques would be of interest. Especially adaptive, online learning controllers like, e.g., neuro fuzzy controller [17] seem to be well suited for this task.

Finally, we like to study the influence of the used placement algorithm on the elastic scaling of our system. Therefore, we want to compare adapted versions of existing load balancing algorithms [18], [19] for data stream processing systems with the used bin packing algorithm.

References

[1] R. Gençay, M. Dacorogna, U. A. Muller, O. Pictet, and R. Olsen, *An introduction to high-frequency finance*, 2001.

[2] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic, "The DEBS 2012 Grand Challenge," in *DEBS 2012: 6th ACM International Conference on Distributed Event-Based Systems*. Berlin, Germany: ACM, July 2012.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] A. Alexandrov, D. Battré, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "Massively parallel data analysis with pacts on nephele," *PVLDB*, vol. 3, no. 2, pp. 1625–1628, 2010.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *CIDR'05*, 2005, pp. 277–289.

[6] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 407–418.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, April 2010.

[8] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 725–736.

[9] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[10] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer, "Elastic complex event processing under varying query load," in *First International Workshop on Big Dynamic Distributed Data (BD3)*, 2013, pp. 25–31.

[11] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[12] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[13] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: A survey," in *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996, pp. 46–93.

[14] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 25–36.

[15] R. Das, G. Tesauro, and W. E. Walsh, "Model-based and model-free approaches to autonomic resource allocation," *IBM Ressearch Report, RC*, vol. 23802, 2005.

[16] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch, "Adaptive provisioning of stream processing systems in the cloud," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. IEEE, 2012, pp. 295–301.

[17] P. Lama and X. Zhou, "Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 151–160.

[18] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 791–802.

[19] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer, "Soda: An optimizing scheduler for large-scale stream-based distributed computer systems," *Middleware 2008*, pp. 306–325, 2008.