# FUGU: Elastic Data Stream Processing with Latency Constraints

Thomas Heinze[1], Yuanzhen Ji[1], Lars Roediger[1], Valerio Pappalardo[1], Andreas Meister[2], Zbigniew Jerzak[1], Christof Fetzer[3]

[1]SAP SE {firstname.lastname}@sap.com

[2]University of Magdeburg andreas.meister@iti.cs.uni-magdeburg.de

[3]TU Dresden christof.fetzer@tu-dresden.de

**Abstract**

*Elasticity describes the ability of any distributed system to scale to a varying number of hosts in response to workload changes. It has become a mandatory architectural property for state of the art cloud-based data stream processing systems, as it allows treatment of unexpected load peaks and cost-efficient execution at the same time. Although such systems scale automatically, the user still needs to set configuration parameters of a scaling policy. This configuration is cumbersome and error-prone.*

*In this paper we propose an approach that tries to remove this burden from the user. We present our data stream processing system FUGU, which optimizes the selected scaling policy automatically using an online parameter optimization approach. In addition, we demonstrate how our system considers user-defined end to end latency constraints during the scaling process.*

## 1 Introduction

Data stream processing systems [1] continuously produce output for a set of standing queries and potentially unbounded input streams. Many real-world workloads for data stream processing systems have a high variability, which means that the data rates of the input streams and the selectivities of query operators are frequently changing in unpredictable ways. Several authors [5, 7, 8] have proposed data stream processing prototypes that automatically scale in or out based on workload characteristics to handle such dynamic workloads. Such systems are called *elastic* [11] and support increasing system utilization by using only the minimum required number of hosts. However, in all these prototypes, the user needs to manually specify a scaling strategy, which controls when and how the system scales.

The challenge of correctly configuring the scaling strategy has been studied for many cloud-based systems [4, 12, 13, 16]. A large number of solutions exist, including auto-scaling techniques [13, 16] and task-classification approaches [4, 12]. These systems can be classified into three major algorithmic categories: prediction-based, sampling-based, and adaptive (learning-based) solutions. Both sampling and prediction-based approaches are hard to apply in a data stream processing system, because its workload is hard to predict or sample due to its high variability. An adaptive auto-scaling technique is able to improve the utilization of such a system, but degrades the quality of service [9]. Each reconfiguration decision in a data stream processing system interferes

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

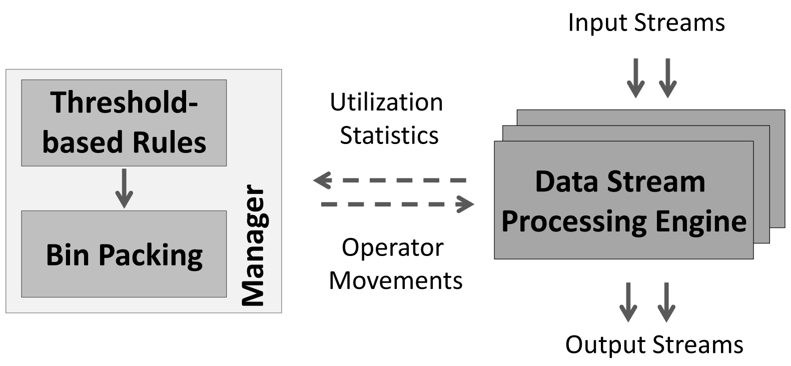

Figure 1: Architecture of FUGU

with the data processing and as a result has a high impact on major quality of service metrics such as end to end latency [8]. Therefore, this characteristic needs to be reflected in the scaling strategy to achieve a good trade-off between the spent monetary cost and the achieved quality of service.

In the context of our elastic data stream processing prototype FUGU, we study how we can relieve the user from configuring these parameters and how to support different quality of service levels. In this paper, we outline the two major concepts we use to realize this vision in context of FUGU: (1) the latency-aware scaling strategy and (2) online parameter optimization. The latency-aware scaling strategy introduces a model to estimate the latency peak created by a scaling decision. This information is used to derive scaling decisions with a minimal latency peak and avoid scaling decisions with a too high latency peak. Online parameter optimization presents a white-box model to study the influence of different parameters on scaling behaviour. This white-box model can be used to search for good parameter settings for the current workload.

In the following, we describe both techniques in the context of an existing data stream processing system. In addition, we present a real-world evaluation to demonstrate the strength of the presented techniques.

## 2 Background

The concepts presented here are implemented as an extension of the elastic data stream processing prototype FUGU [8, 9] (see Figure 1). The existing system consists of a centralized management component, which dynamically allocates a varying number of hosts. The manager executes on top of a distributed data stream processing engine, which is based on the Borealis semantic [1].

The data stream processing system processes continuous queries, which can be modeled as directed acyclic graphs of operators. Our system supports primitive relational algebra operators (selection, projection, join, and aggregation) as well as additional data stream processing specific operators (sequence, source, and sink). Each operator can be executed on an arbitrary host and a query can be partitioned over multiple hosts. The number of hosts is variable and dynamically adapted by the management component to changing resource requirements.

The centralized management component serves two major purposes: (1) it derives scaling decisions, including decisions on allocating new hosts or releasing existing hosts, and assigns operators to hosts; and (2) it coordinates the construction of the operator network in the distributed data stream processing engine.

The management component constantly receives statistics from all running operators in the system. Based on these measurements and a set of thresholds and parameters, it decides when to scale and where to move operators. Typically, these thresholds and parameters are manually specified by the user. Our system supports the movement of both stateful (join and aggregation) and stateless operators (selection, sink, and source). A state of the art movement protocol [8, 15] ensures an operator moves to the new host without information loss.

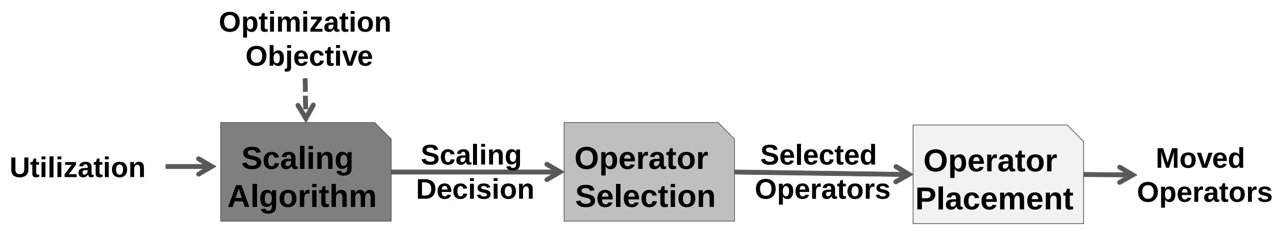

Figure 2: Scaling Strategy of FUGU

## 3 Threshold-based Elastic Scaling

The scaling approach used by the FUGU server is illustrated in Figure 2. A vector of node utilization measurements (CPU, memory, and network consumption) and a vector of operator utilizations are used as input to the *Scaling Algorithm*. The *Scaling Algorithm* derives decisions that mark a host as overloaded or the system as underloaded. The *Operator Selection* algorithm decides which operators to move and the *Operator Placement* algorithm determines where to move these operators.

The default scaling strategy of FUGU is threshold-based, namely, a set of threshold rules are used to define when the system needs to scale. These thresholds mark either the entire system or an individual host as over/underloaded. A threshold rule describes an exceptional condition for the consumption of one major system resource (CPU, network, or memory), which triggers a scaling decision in FUGU. Some examples for these rules include:

1. A host is marked as overloaded if the CPU utilization of the host is above 80% for three seconds.
2. A host is marked as underloaded if the CPU utilization of the host is below 30% for five seconds.

The threshold-based rules need to be used carefully [6]. In particular, the frequent alternating allocation and deallocation of virtual machines, called thrashing, should be prevented. Several steps are taken in FUGU to avoid thrashing. First of all, each threshold needs to be exceeded for a certain number of consecutive measurements before a violation is reported. This number is called the *threshold duration*. In addition, after a threshold violation is reported, no additional scaling actions are done for the corresponding host for a certain time interval called a *grace period* (or cool-down time). The system checks for overloaded or underloaded host each time a new batch of utilization measurements for all operators has been received. Our scaling strategy checks all hosts using the overload criteria first, afterwards it tests if the system is underloaded. This order avoids to first release a host due to an underload and afterwards allocate a new host to solve an overload.

The load in a data stream processing system is partitioned among all operator instances running in the system. Therefore, each scaling decision needs to be translated into a set of moved operators. The first problem is to identify which operators to move. This identification is done by the *Operator Selection* algorithm. If the system is marked as underloaded, it selects all operators running on the least loaded hosts. For an overloaded host, the *Operator Selection* algorithm chooses a subset of operators to move in a way, that the summed load remaining on the host is smaller than the given threshold. FUGU models this decision as a *subset sum problem* [14], where the operators on the host are the possible items and the threshold represents the maximum sum. We use a heuristic, which identifies the subset of all operator instances whose accumulated load is smaller than the threshold and no other subset with a larger accumulated load fulfilling this condition exists. All operators selected by this algorithm are kept on the host; the remaining operators are selected for movement.

The selected operators are the input of the *Operator Placement* algorithm, which decides *where* the operators should be moved. We solve this problem using different bin packing algorithms [3]. The goal of a bin-packing algorithm is to assign each item to exactly one bin in a way that (1) the number of bins is minimized and (2)

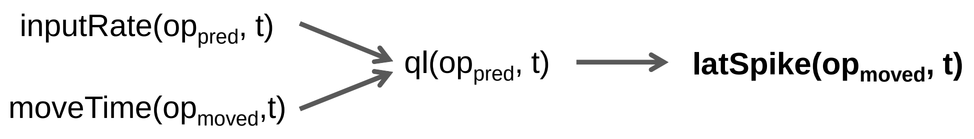

Figure 3: Latency Peak Estimation

the sum of the weights of all assigned items is smaller than the capacity of the bin. In the context of FUGU, an operator represents an item and its CPU usage is its weight. A host is modeled as a bin with its CPU resource as the capacity. In addition, we use network and memory consumption as sub-constraints. The bin-packing problem is known to be NP-complete [14], however, many efficient heuristics have been proposed to solve it. For FUGU we implemented two well-known bin-packing heuristics, FirstFit and BestFit.

## 4 Latency-aware Elastic Scaling

As illustrated in the previous section, a set of operators needs to be moved between hosts in the system in response to a scaling decision. This movement has to ensure that no information is lost. This condition requires the usage of an operator movement protocol [15], which guarantees that an operator and its state are moved together. For each operator to be moved, the protocol used first pauses the processing of the predecessor operators, which causes all newly arriving events to be enqueued. Then, a new instance of the operator is created and the operator state is moved. When the state movement is completed, the predecessor operator is restarted. As the processing of the enqueued events at the predecessor operator is delayed, a latency peak can be observed. Existing scaling strategies [5, 7] optimize the scaling decision based only on the CPU load moved or the state size moved and ignore the resulting latency peak.

In FUGU we deal with this problem by introducing a model to estimate the latency peak created by an operator movement. The model (see Figure 3) estimates the queue length $ql(op_{pred}, t)$ of the predecessor operator created during the movement, which determines the observed latency peak. As input for this estimation two major factors are considered: workload characteristics such as the current input rate *inputRate* of the predecessor operator $op_{pred}$ and the movement time $moveTime$ of the moved operator $op_{moved}$. The major challenge is that the movement time of an operator depends on multiple factors such as the state size, the operator type, and the current host load [8]. Therefore, we collect a set of samples of these characteristics together with the corresponding latency peak online. The samples are clustered based on these factors, and for a new operator movement, the cluster of samples with the highest similarity is identified. That subset of samples is used to estimate the movement time for new movements.

This estimation model is used to extend the *Operator Selection* algorithm presented in Section 3. Our system allows the user to define a latency threshold, which is considered when the scaling decisions are computed. We classify scaling decisions into two categories (1) mandatory and (2) optional movements. All scaling decisions necessary to avoid an overload of the system are mandatory scaling decisions. The release of a host due to underload is an optional scaling decision. Any optional scaling decision can be postponed or canceled in case the estimated latency peak would be too high. Thereby, unnecessary violations of the latency constraints can be avoided. The operator selection for an overloaded host is modified to identify a set of candidate solutions whose summed operator loads are above a certain CPU threshold. Among all candidates, the solution with the minimum estimated latency peak is chosen. In addition, the way in which the system handles CPU underload is changed. Normally, if the system detects a system underload, the host with the minimal CPU load is released and all operators running on this host are moved to other hosts. In our latency-aware elastic scaling the system releases the host, that minimizes the estimated latency peak for moving all operators on the host. If no host with

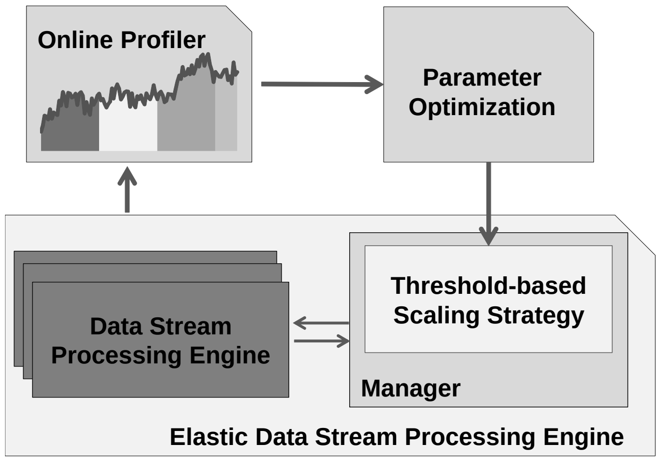
Online Profiler
Parameter Optimization
Threshold-based Scaling Strategy
Data Stream Processing Engine
Manager
Elastic Data Stream Processing Engine

Figure 4: Architecture of Online Parameter Optimization

an estimated latency peak below the user-defined threshold exists, only a subset of the operators on the host with the smallest estimated latency peak is moved.

## 5 Online Parameter Optimization

The configuration of a threshold-based scaling strategy is very difficult for an inexperienced user, as he typically has a limited understanding of the system and the influence of the possible parameter settings on system performance. Therefore, we introduce an online parameter optimization approach, which chooses these parameter settings automatically based on current workload characteristics.

Online parameter optimization adds two new components to the existing elastic scaling data stream processing engine (see Figure 4): a parameter optimization component and an online profiler. We identified a set of six major parameters for our system, such as utilization thresholds and the bin packing method used, that primarily influence the scaling behaviour of the system and describe the parameter configuration of the scaling strategy. For each parameter, we determine a reasonable domain. In total, 720,000 parameter configurations exist [10].

Our optimization component automatically discovers a good parameter configuration based on a short-term utilization history of the running system. In this approach we use a cost function [10], that models the influence of these parameters on the scaling behaviour. Threshold-based scaling deterministically derives a scaling decision for a given operator assignment of operator instances, current utilization values and a setting of the mentioned parameters. For the cost function, we input a time series of utilization values and assignments and get as a result a set of scaling decisions for the given parameter settings. From these scaling decisions, we can determine both the amount of resources used and the latency peaks created by the scaling decisions.

We determine possible parameter configurations using an improved random search algorithm [17] and identify a configuration with a good trade-off between resources used and latency based on the short term utilization history. Finally, we compare these results with the results of the current parameter configuration of the system and adapt the parameters, if a configuration with less host use and a less or equal number of moved operators was found.

The previously mentioned online profiler determines the frequency of triggering the parameter optimization. It monitors changes of the workload pattern based on the overall CPU load using an adaptive window [2]. The system periodically adds a new value to the window. If this new value is similar to the existing values, it is simply appended at the head of the window. If a significant change is detected, values from the tail are deleted until all values in the window are similar again. Parameter optimization is triggered each time a change is detected. The

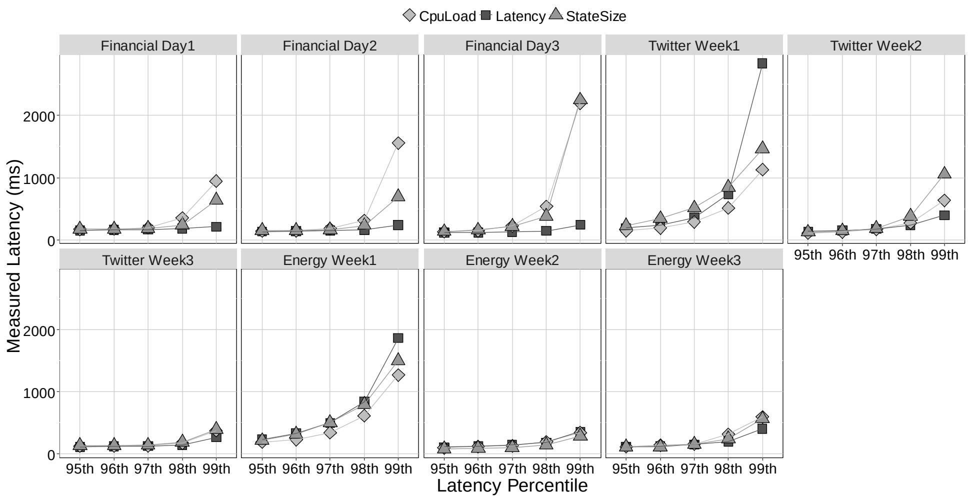

Figure 5: Latency Results for Different Operator Selection Strategies

length of the window also specifies the length of the short-term history of current load characteristics to use for the online parameter optimization. This approach allows adaptively identifying a good parameter setting for the system.

## 6 Evaluation

We implemented both latency-aware elastic scaling and online parameter optimization as extensions of FUGU. During an evaluation with three real-world scenarios, we tried to answer two major questions:

1. Does latency-aware elastic scaling improve latency compared to other operator selection strategies?
2. Does online parameter optimization provide a good trade-off between system utilization and query processing latency, thus relieving the user of the burden of manually configuring the parameters?

In the evaluation we use a private, shared cloud environment with one master node and up to twelve workers. We run three different real-world scenarios [10]: a scenario with financial data, one with Twitter messages, and a third with smart meter measurements. For each case we use three different traces, which make up in total nine workloads. Each experiment lasts for 90 minutes, where end to end latency and host utilization are measured roughly every five seconds. For a single measurement point, we use the average utilization of all hosts and average latency of all queries to quantify the utilization of the system and the quality of service, respectively.

### 6.1 Latency-aware Elastic Scaling

We compare our latency-aware operator selection strategy with two alternative operator selection strategies [8]: CPULoad and StateSize. The CPULoad strategy selects operators to move in a way that minimize the total CPU load moved. In contrast, the StateSize strategy minimizes the total state size moved, when moving operators between hosts. For each strategy we evaluated six different thresholds and average the results to avoid any influence of the chosen threshold configurations on the results. We present the resulting latency in Figure 5 and the measured utilization values in Figure 6.

For the latency results we show the 95th, 96th, 97th, 98th, and 99th percentiles of all measurements. The measured results for the 95th, 96th and 97th percentile for the three strategies differ only very marginally, which

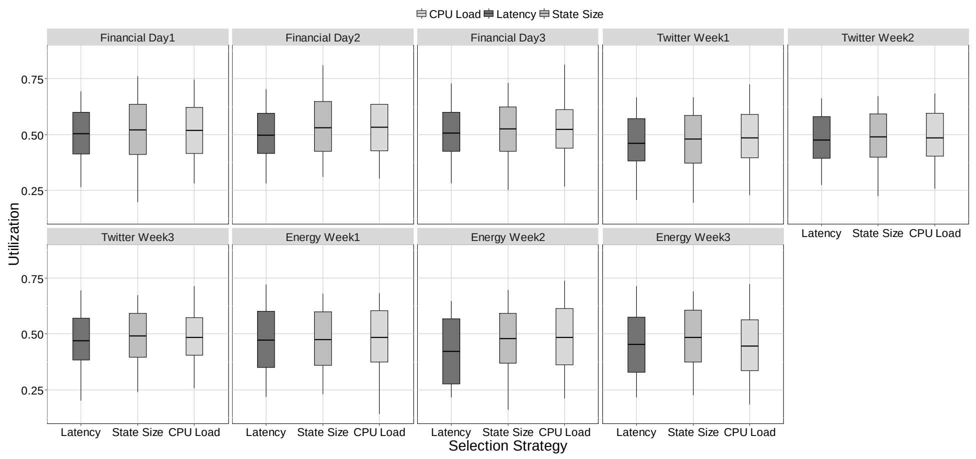

Figure 6: Utilization Results for Different Operator Selection Strategies

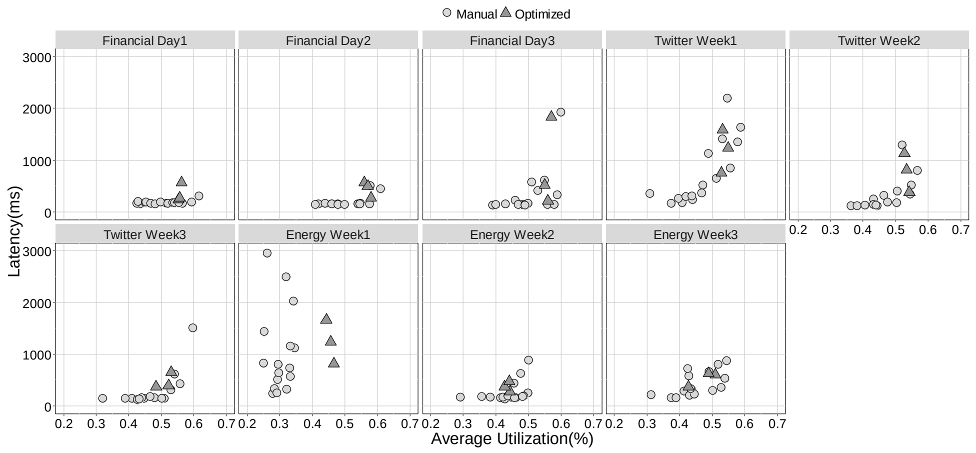

Figure 7: Comparison of Online Parameter Optimization and Manually Tuned Thresholds

demonstrates that the operator selection strategy used influences only the measured latency peaks. The latency-aware operator selection we presented outperforms the two other strategies in seven out of nine scenarios. On average, over all nine scenarios, the latency-aware selection strategy has a 18% and 19% lower 98th percentile latency than the CPULoad and StateSize strategies, respectively. For the 99th percentile our strategy's latency is 16% and 22% lower than for the CPULoad or StateSize strategies.

Figure 6 shows a comparison of the utilization results, where we present a comparison of the average utilization for the three different strategies using a boxplot. The operator selection strategy used has only a small influence on the utilization achieved. The latency-aware strategy has only a two percent point smaller utilization than the CPULoad or the StateSize strategy.

## 6.2 Online Parameter Optimization

As a baseline for online parameter optimization, we manually tuned the thresholds. We evaluated 16 different threshold configurations and compared the results achieved for our parameter optimization over three different runs. We show the average node utilization and the 98th percentile of the averaged latency in Figure 7.

The results show a significant variance in both the average utilization and the latency for different configurations: the minimal and maximal utilization differ by 20 percentage points. From the 16 measurements, we extract the average to estimate the results that an inexperienced user might achieve. Online parameter optimization shows a five percentage point better utilization with only a slight increase of the 98th percentile latency (231 ms) averaged over all scenarios.

Subsequently, we selected the three best configurations per workload and compared them to the configuration derived by online parameter optimization. Online parameter optimization shows comparable utilization results (0.02% worse) and again only a small increase of the 98th percentile latency (330 ms).

From these results we conclude that our online parameter optimization provides a good trade-off between system utilization and query processing latency. It also removes the burden of manually choosing the thresholds from the user.

## 7 Summary

Elastic scaling allows a data stream processing system to react to unexpected load spikes and reduce the amount of idling resources in the system. Although several authors proposed different approaches for elastic scaling of a data stream processing system, these systems require a manual tuning of the thresholds used, which is an error-prone task and requires detailed knowledge about the workload.

In this paper we introduce a model to estimate the latency peak created by a scaling decision and present an approach to minimize that peak accordingly. In addition, we propose an online parameter optimization approach, which automatically adjusts the scaling strategy of an elastic scaling data stream processing system. Our system minimizes the number of hosts used and at the same time keeps the number of latency peaks low. Both approaches have been evaluated in the context of several real-world use cases and have demonstrated their applicability for such use cases.

## References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, “The Design of the Borealis Stream Processing Engine,” in *CIDR ’05: Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, 2005, pp. 277–289.

[2] A. Bifet and R. Gavaldà, “Learning from Time-Changing Data with Adaptive Windowing,” in *SDM 2007: Proceedings of the Seventh SIAM International Conference on Data Mining*, 2007, pp. 443–448.

[3] 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.

[4] M. Ead, H. Herodotou, A. Aboulnaga, and S. Babu, “PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs,” in *EDBT ’14: Proceedings of the* $17^{th}$ *International Conference on Extending Database Technology*, 2014, pp. 1–12.

[5] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management,” in *SIGMOD ’13: Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 725–736.

[6] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, “Exploring Alternative Approaches to Implement an Elasticity Policy,” in *CLOUD ’11: Proceedings of the IEEE International Conference on Cloud Computing*. IEEE, 2011, pp. 716–723.

[7] 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 (TPDS)*, vol. 23, no. 12, pp. 2351–2365, 2012.

[8] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems,” in *DEBS ’14: Proceedings of the* $8^{th}$ *ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[9] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, “Auto-scaling Techniques for Elastic Data Stream Processing,” in

*ICDEW '14: Workshops Proceedings of the* 30*th* *International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 296–302.

[10] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online Parameter Optimization for Elastic Data Stream Processing," in *SoCC '15: Proceedings of the ACM Symposium on Cloud Computing 2015*. ACM, 2015, pp. 276–287.

[11] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *ICAC '13: Proceedings of the* 10*th* *International Conference on Autonomic Computing*, 2013, pp. 23–27.

[12] H. Herodotou and S. Babu, "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[13] 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*, vol. 12, 2012.

[14] S. Martello and P. Toth, "Algorithms for Knapsack Problems," *Surveys in Combinatorial Optimization*, vol. 31, pp. 213–258, 1987.

[15] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," in *ICDE '03: Proceedings of the* 19*th* *IEEE International Conference on Data Engineering*. IEEE, 2003, pp. 25–36.

[16] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *SoCC '11: Proceedings of the second ACM Annual Symposium on Cloud Computing*. ACM, 2011, pp. 1–14.

[17] T. Ye and S. Kalyanaraman, "A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration," in *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 2003, pp. 196–205.