Online Parameter Optimization for Elastic Data Stream Processing

Thomas Heinze[1], Lars Roediger[1], Andreas Meister[2], Yuanzhen Ji[1], 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

Elastic scaling allows data stream processing systems to dynamically scale in and out to react to workload changes. As a consequence, unexpected load peaks can be handled and the extent of the overprovisioning can be reduced. However, the strategies used for elastic scaling of such systems need to be tuned manually by the user. This is an error prone and cumbersome task, because it requires a detailed knowledge of the underlying system and workload characteristics. In addition, the resulting quality of service for a specific scaling strategy is unknown a priori and can be measured only during runtime.

In this paper we present an elastic scaling data stream processing prototype, which allows to trade off monetary cost against the offered quality of service. To that end, we use an online parameter optimization, which minimizes the monetary cost for the user. Using our prototype a user is able to specify the expected quality of service as an input to the optimization, which automatically detects significant changes of the workload pattern and adjusts the elastic scaling strategy based on the current workload characteristics. Our prototype is able to reduce the costs for three real-world use cases by 19% compared to a naive parameter setting and by 10% compared to a manually tuned system. In contrast to state of the art solutions, our system provides a stable and good trade-off between monetary cost and quality of service.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications

Keywords Distributed Data Stream Processing, Load Balancing, Elasticity, Parameter Optimization

1. Introduction

A major problem of today's cloud infrastructure is the low utilization of the overall system, which results in both high cost for the user and low energy efficiency of the cloud itself. Two independent studies [6, 21] showed that the average utilization of today's cloud infrastructure is only about 30%, although more than 80% of the resources are reserved by a user. This is caused by the fact, that a user typically needs to define a scaling strategy by configuring the number of used machines, a set of thresholds, or a controller to decide when the system should scale in or out. However, often a user has a limited understanding of the used system and the workload. Therefore, he chooses the scaling scheme conservatively to be able to handle peak loads and as a result achieves a very low system utilization.

This observation also holds true for data stream processing systems [1, 25], which continuously produce output for a set of standing queries and a potentially infinite input stream. Many real-world workloads for data stream processing systems have a high variability, which means the data rate and selectivities of the streams are frequently changing in an unpredictable way. Several authors [8, 11, 12] proposed data stream processing prototypes, which automatically scale in or out based on workload characteristics, to handle these dynamic workloads. Such systems are called elastic [14] and allow to increase the system utilization by only using the minimal required number of hosts. However, in all of these prototypes the user needs to manually specify a scaling strategy.

The challenge of correctly configuring the scaling strategy has been studied for many cloud-based systems [7, 15, 18, 23]. A large number of solutions exist including auto-scaling techniques [18, 23] and task classification-based approaches [7, 15]. These systems can be classified into three major algorithmic categories: prediction-based, sampling-based, or adaptive learning-based solutions. The workload of a data stream processing system is due to its high variability hard to predict or sample. In our previous work [13], we illustrated that an adaptive auto-scaling technique is able to improve the utilization of the system, but degrades the quality of service. Each reconfiguration decision in a data stream processing system interferes with the data processing and as a result has a high impact on major quality of service metrics like the end to end latency [12]. Therefore, this characteristic needs to be reflected in the scaling strategy to achieve a good trade-off between the monetary cost spent and the achieved quality of service.

In this paper we address the problem of choosing the scaling strategy for a data stream processing system by introducing a novel approach based on online parameter optimization. We make the following contributions:

1. We present an online parameter optimization approach, which automatically (1) chooses the scaling strategy to minimize the number of hosts used for the current workload characteristics, (2) detects changes of the workload pattern, and (3) adapts the scaling policy accordingly. Our system removes the burden from the user to manually configure the scaling policy.
2. We show how this parameter optimization problem can be enhanced with a given quality of service constraint for the maxi-

SoCC'15, August 27-29, 2015, Kohala Coast, HI, USA.
 ISBN 978-1-4503-3651-2/15/08…$15.00.
DOI: http://dx.doi.org/10.1145/2806777.2806847

mum end to end latency. This constraint is used during the optimization to improve the trade-off between monetary cost and quality of service.

3. We evaluate our prototype based on three real-world use cases. We show that we reduce the average cost by 19% compared to a naive scaling scheme and by 10% compared to a manually tuned scaling strategy with a comparable or even better quality of service. We also show that our solution outperforms a state of the art adaptive auto-scaling technique based on Reinforcement Learning due to the more precise modeling of the scaling behavior of a data stream processing system.

1.1 Solution Overview

The solution presented in this paper consists of three major components (see Figure 1): an elastic scaling data stream processing engine, an online profiler, and a parameter optimization component.

We extend an existing elastic data stream processing engine [12, 13], which scales automatically in and out based on a user-defined scaling policy and the current workload characteristics. The scaling policy uses threshold-based rules for the maximum and minimum utilization of a host or the system respectively. These thresholds and some additional parameters typically have to be chosen manually by the user to fit the specific use case characteristics.

We also introduce an online profiler, which 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 [4]. If a change of the workload pattern is detected, the optimization component is triggered using an up to date short-term history of current load characteristics.

1.2 Outline

In the following, first we present some background information about the used state of the art elastic data stream processing engine in Section 2.1 and illustrate in Section 2.2 how a user-defined scaling strategy is applied to allow an elastic scaling of the system. We express the search of the scaling policy as an optimization problem and introduce our online parameter optimization to solve this problem in Section 3. In addition, we describe how a user can control the costs by configuring his expected quality of service and how our prototype uses this constraint to foster the optimization run. Then, we show how the system detects the right point in time to trigger the parameter optimization in Section 4. The evaluation of our solution is presented in Section 5. We conclude the paper with an overview of related work in Section 6.

2. System Overview

This section presents background information about the underlying elastic data stream processing system used in this paper. We illustrate its core properties and describe the existing threshold-based scaling policy.

2.1 Elastic Data Stream Processing Engine

The presented parameter optimization is implemented as an extension of the elastic data stream processing prototype FUGU [12, 13]. The existing system consists of a centralized management component dynamically allocating a varying number of hosts. The manager is executed 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 consisting of 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. Therefore, a query can be partitioned over multiple hosts. The number of hosts is variable and dynamically adapted by the management component to the changing resource requirements.

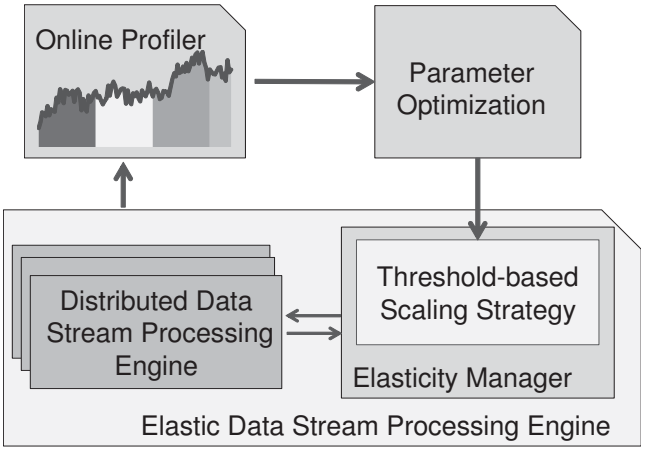


Figure 1: Architecture of our Prototype

The centralized management component serves two major purposes: (1) it derives scaling decisions, including decisions to allocate new hosts or release 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 scaling strategy 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 the operators. These thresholds and parameters are typically manually specified by the user.

In our system the movement of both stateful (join and aggregation) as well as stateless operators (selection, sink, and source) is supported. A state of the art movement protocol is used to ensure an operator movement to the new host without information loss. A detailed description of the protocol is presented in [12].

2.2 Threshold-based Elastic Scaling

In general, the problem of elastic scaling a data stream processing engine can be solved by deciding (1) *when* to scale and (2) *where* to move the operators in case a scaling decision has been made.

In the following, we first present our scaling strategy to solve the question *when* to scale. Afterwards, we describe the redistribution strategy used in our manager to answer the question, *where* to place the moved operators.

Runtime Operator Placement

The task of the runtime placement is to check if the system needs to scale in or scale out. Our system considers three major resources: CPU, network, and memory consumption. In a data stream processing system all these metrics vary over time due to the changing input rate and data distribution in a data stream. Therefore, we continuously measure these metrics for each operator and host individually. The execution of a runtime placement is started as soon as a complete set of utilization measurements is received by the centralized manager. Our solution uses threshold-based rules like established elastic scaling strategies, e.g., Amazon Auto-Scaling [2].

If the runtime placement is executed, each host utilization is checked against an upper threshold. In case the upper threshold $thres^{\uparrow}$ is exceeded for $d^{\uparrow}$ successive measurements, the host is marked as overloaded. For overloaded hosts the system has to decide which operators have to be left on these hosts and which

operators have to be moved to other hosts. For this purpose, we use a subset sum algorithm[19], which chooses a subset of operators to keep on the hosts such that the summed load is below the specified threshold. In addition, no other subset exists, which has a larger summed load and satisfies this condition. All not selected operators are reassigned to not overloaded hosts.

The system is marked as underloaded, if the average host utilization drops below the lower threshold $thres_{\downarrow}$ for $d_{\downarrow}$ consecutive measurements. In this situation our manager selects the least loaded host and tries to redistribute all operators of this host to the remaining hosts. In case a placement of all operators on other hosts is possible, the host is released. In case not enough free capacity on the remaining hosts exists, the release is canceled and no operator is moved.

Additional rules are used to monitor if a host exceeds the available memory or network bandwidth. If the upper threshold is violated for a set of measurements, the manager triggers the movement of operators based on the subset sum algorithm using the memory or network consumption as parameters.

Certain actions are taken to prevent too frequent scaling decisions: (1) after each scaling decision the affected host is not touched for a certain period of time, called the grace period g, (2) scaling decisions are only made after $d^{\uparrow}$ or $d_{\downarrow}$ consecutive violations of the threshold, and (3) the load to move for a scaling out decision is chosen to reduce the host load at least 10% below the upper threshold $thres^{\uparrow}$.

Redistribution Algorithm

We model the operator redistribution problem as an incremental bin packing problem [5]. A host is modeled as a bin with its available CPU resources as the capacity. An operator is modeled as an item with its CPU load as the weight. The items need to be assigned to bins in such a way that no bin is overloaded and each item is placed on exactly one bin. Additionally, the bin packing enforces an upper bound for the memory and network utilization of a host. For each operator its memory and network consumption is modeled and the bin packing assigns operators only to hosts with sufficient network bandwidth and memory capacity.

As a heuristic for the bin packing we implemented a FirstFit and a BestFit heuristic [5]. Both heuristics can be combined with two extensions: (1) the system sorts the operators in a decreasing order of their CPU load before they are placed and (2) a host with predecessor or successor operators is a preferred target for the placement of moved operators. The first extension is known to improve the packing of operators on hosts in certain scenarios. The second extension reduces the network load. In total, our system supports eight different bin packing variants.

During the assignment phase unnecessary operator movements are avoided by using only the operators chosen by the scaling algorithms as the input and pinning the remaining operators to their current hosts.

3. Online Parameter Optimization

In this section we introduce the optimization algorithm proposed in this work. In general, the parameter optimization is used to find the parameter configuration for the scaling strategy, which (1) minimizes the monetary cost the user needs to pay and (2) ensures a good quality of service.

The approach of the parameter optimization is outlined in Figure 2. A run of the optimization component is triggered by the online profiler after a change in the workload pattern was detected. As the first step, the system determines the spent monetary cost $cost(p)$ for the current configuration p and the short-term utilization history $utils$. Therefore, it uses a cost function, which simulates

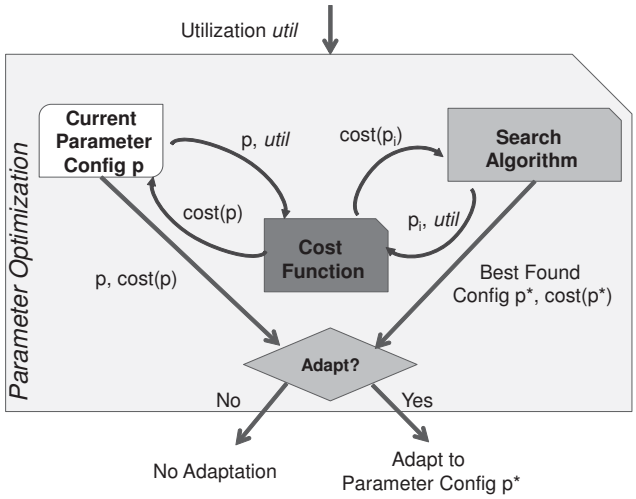


Figure 2: Overview of the Parameter Optimization

the scaling behavior of the system based on a given parameter configuration and these utilization values. As the second step, a search algorithm finds new promising configurations p_i by identifying parameter configurations with a low cost value $cost(p_i)$. Finally, the system compares the monetary cost for the best parameter configuration p^* and the current configuration p. If the new parameter configuration p^* has a smaller monetary cost, the used scaling strategy is adapted.

In the following, we illustrate the three major components: (1) the parameter search space, (2) the cost function, and (3) the search algorithm.

3.1 Parameter Search Space

A parameter configuration p_i describes a setting for six parameters of our scaling strategy (see Table 1). The current settings of these parameters influence the scaling behaviour of our prototype, e.g., the system scales in earlier if the lower threshold is increased or scales out later in case the upper threshold is increased.

The parameter domain for each individual parameter is described in Table 1. For each parameter we define a lower and an upper bound to reduce the size of the search space. We ensure that no dependency between different parameters exists. Especially, the domain for the lower and the upper threshold are disjunct to avoid that a host can be overloaded and underloaded at the same time. The small range for the upper threshold duration as well as a maximal upper threshold of 0.9 are chosen to minimize the probability of overloading a host, which creates typically an unstable system behavior and many latency peaks [13]. In total, our search space contains $720,000$ different parameter configurations.

Parameter	Domain	Granularity
Lower Threshold $thres_{\downarrow}$	[0.0,0.5]	0.01
Lower Threshold Duration $d_{\downarrow}$	[3,10]	1
Upper Threshold $thres^{\uparrow}$	[0.75,0.9]	0.01
Upper Threshold Duration $d^{\uparrow}$	[2,4]	1
Grace Period g	[1,5]	1
Bin Packing Variant bin	8 heuristics	-

Table 1: Domains and granularities for the used parameters

The thresholds for memory and network overload are kept fixed to avoid any malfunction of the system.

3.2 Cost Function

The goal of our cost function is to calculate the monetary costs for a parameter configuration, where the best configuration is marked by a minimal cost value. For this purpose, we utilize a short-term history of utilization snapshots. We simulate the scaling behaviour of our prototype using this time series. Depending on the parameter configuration p_i the system scales in or out to a varying number of host. From the total number of used hosts, we derive the result of the cost function as the monetary cost $cost(p_i)$.

We use as an input an utilization snapshot $utils(t)$ for the point in time t, which contains the current CPU usage, input rate, output rate, memory, and network usage for all operators ops. As an additional input the current assignment of operators to hosts $assign(t)$ is used.

The cost function derives from the utilization snapshot for each active host a CPU, memory, and network usage. Similar to state of the art operator placement approaches [16, 26], we simplify this calculation by assuming no interference between operators on a host and using the summed load of all operators assigned to a host. The solution presented in Section 2.2 is configured with parameter configuration p_i and executed with the simulated host loads as an input. The runtime placement evaluates if any host is overloaded or underloaded and calculates a new assignment $assign'(t, p_i)$.

Certain parameters, e.g., the grace period and the threshold duration, do not have any influence on the scaling behaviour if the cost for a single utilization snapshot is calculated. Therefore, we store a series of the last k utilization snapshots, where k is determined by the online profiler (see Section 4). As a first step of the calculation, the cost function calculates for the utilizations $utils(t_0)$ and the stored operator assignment $assign(t_0)$ a new assignment $assign'(t_1, p_i)$. Afterwards, the scaling strategy is triggered again with the operator utilizations $utils(t_1)$ and $assign(t_1) = assign'(t_0, p_i)$ as a new input. This procedure is repeated until the snapshot $util(t_k)$ has been processed.

An example for the used cost function is presented in Figure 3. The input to the cost function consists of a parameter configuration p_i and a series of utilization snapshots $utils$ with length 3. Each snapshot contains operator utilizations for three operators $S1$, $A1$, and $D1$. The initial assignment $assign(t_0)$ places these operators on two hosts $H1$ and $H2$. At t_1 the load of all operators increases, and host $H1$ exceeds the defined upper threshold $thres^{\uparrow} = 0.8$. Therefore, the system needs to scale out to three hosts. At the point in time t_3 the load decreases again and the system is able to scale in.

For calculating the cost value $cost(p_i)$, we assume a pay per use pricing scheme for the used cloud environment, because it is the most frequently used pricing scheme in public clouds [17]. In this scheme, a user is charged a fixed price per virtual machine $price_{VM}$ for a time interval t_{charge}. The used virtual machines are derived from the k calculated assignment $assign'(t_j, p_i)$. The total cost value $cost(p)$ is determined by summing up the number of used intervals per virtual machine. For the presented example, the hosts $H1$ and $H2$ are used for all four points in time and host $H3$ is used for two points. If we use time interval $t_{charge} = 1$ the monetary cost is calculated as follows $cost(p_i) = (4+4+2) \cdot price_{VM} = 10 \cdot price_{VM}$.

3.3 Latency-Aware Cost Function

Our prototype intends to minimize the monetary cost and at the same time to fulfill certain quality of service constraints. As a measure for the quality of service, we use an upper bound for the query end to end latency. In this section we introduce how we define

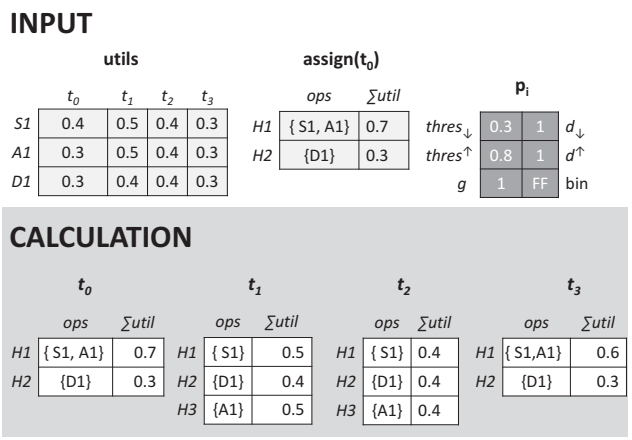


Figure 3: Example for our Cost Function

and estimate a violation of this upper bound. In addition, we present how the parameter optimization is extended to reflect the number of latency violations.

We measure the end to end latency $lat(q_l, t_j)$ for each individual query q_l for a specific point in time t_j. The system latency $lat(t_j)$ is measured as the average latency of all currently running queries. A latency violation is detected, if the current system latency is larger than the user-defined threshold lat_{thres}.

In our system, scaling decisions are optimized to minimize the reconfiguration overhead of the system. Therefore, operators to move and hosts to release are selected in a way to minimize the latency peak created by the operator movement [12]. The system tries to avoid optional scaling decisions, e.g., scale in decisions, which would exceed the latency threshold lat_{thres}. In addition, the system prefers the movement of stateless operators or operators with small state sizes over operators with a large state.

However, the elastic scaling of our data stream processing engine still creates due to unexpected load peaks a set of latency violations caused by (1) an overloaded host or (2) operator movements due to scaling decisions. The online parameter optimization of the scaling strategy has to choose configurations, which minimize monetary cost without creating additional latency violations. For this purpose, our prototype estimates for the given parameter configuration p_i and a set of utilization snapshots how many latency violations $lat_v(p_i)$ would be created.

We adjust our cost function as follows:

$$cost(p_i) = \begin{cases} cost(p_i) & \text{if } lat_v(p_i) = 0 \\ MAX + lat_v(p_i) & \text{if } lat_v(p_i) > 0 \end{cases}$$

The MAX is a predefined constant, which is larger than the maximum expected cost of any configuration. The implemented approach ensures that at any time a configuration without additional latency violations is favored over a configuration with a lower cost but with latency violations. In addition, in case no valid configuration exists, the configuration with the smallest number of violations is chosen.

During the cost function calculation for a parameter configuration p_i we estimate for each query q_l and each point in time t_j the latency $lat_e(q_l, t_j)$. If for a given point in time t_j the average of all estimated query latencies exceeds the latency threshold lat_{thres}, we count for this point in time a latency violation:

$$lat_v(p_i) = \sum_{j=0}^{k} lat_v(p_i, t_j),$$

where

$$lat_v(p_i,t_j) = \begin{cases} 1, & \text{if } \text{avg}_{q_l \in Q}(lat_e(q_l,t_j)) \geq lat_{thres} \\ 0, & \text{otherwise} \end{cases}$$

We assume that the scaling strategy has no influence on the processing latency, if no host is overloaded and no operator is moved. Therefore, we initialize the latency $lat_e(q_l,t_j)$ to the measured latency value $lat(q_l,t_j)$, which is contained in the utilization snapshot $utils(t_j)$.

We check for each point of the used time series $utils(t_j)$, if any host is overloaded before the runtime placement starts. In our prototype we detect an overloaded host by a CPU utilization larger than 95%. Each query q_l with at least one operator running on an overloaded host, is assumed to violate the latency threshold and we define $lat_e(q_l,t_j) = lat_{thres}$.

Our system moves a set of operators ops_{moved} as a result of a scaling decision. During an operator movement the processing of the incoming events needs to be paused to avoid the loss of information during the time between the old instance is stopped and the new instance is ready to run [12]. In our prototype this is realized by pausing the predecessor operator of a moved operator. This interruption of the processing can be noticed by the user in a temporary latency increase after the movement is finished. The latency peak $latPeak_{move}(o)$ created for the moved operator o is estimated using an existing estimation model [12], which computes the duration of an operator movement and the resulting latency peak. As a result, we can estimate the latency for a query q_l, where one (or more) operator $o \in q_l$ is moved:

$$lat_e(q_l,t_j) = lat(q_l,t_j) + \max_{o \in ops_{moved}} (latPeak_{move}(o))$$

3.4 Search Algorithm

The task of the search algorithm is to find the best parameter configuration p^* with the minimal cost, given the current time series of operator utilizations.

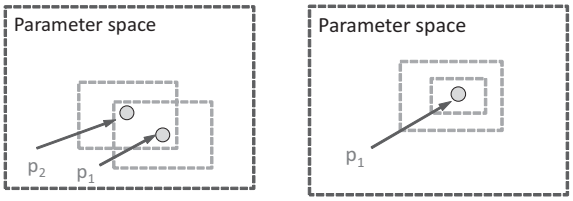
Our search problem is not differentiable and, therefore, cannot be treated by existing gradient search algorithms. In addition, the search should be able to keep an upper bound on the runtime to stay responsive to the varying workload. Therefore, we use an established search algorithm based on random search, called Recursive Random Search (RRS) [27]. It has been used for searching the correct parameter configuration for network models [27] as well as for the search of optimal parameter settings for MapReduce dataflows [15].

The basic assumption of RRS is that the efficiency of a classical random sampling decreases with an increasing sample size. RRS tries to improve this characteristic by restarting the search for promising areas. The algorithm consists of two major phases: exploration and exploitation. During the exploration the complete search space is studied for a solution with the minimum cost. During the exploitation only a promising area is considered. This area is chosen based on the sample with the minimum cost found during the exploration phase.

A promising area used in the exploitation phase is defined by its size and a center. The algorithm studies additional parameter configurations inside this promising area to potentially find a better solution. The exploitation phase distinguishes two scenarios depending on the result for these parameter configurations(see Figure 4):

Realign sub-phase If a new better solution has been found during the exploration phase the sample space is moved towards this new best solution.

Shrink sub-phase If no better solution has been found the size of the sample space is reduced.



(a) Realign (b) Shrink

Figure 4: Possible Actions in the Exploration Phase of Recursive Random Search [27]

The exploitation phase terminates, when the size of the area is below a certain threshold. Afterwards, the search continues with a new exploration phase until a termination condition is fulfilled. In our system, as a termination criteria an upper bound for the number of evaluated configurations is used.

4. Online Profiler

A key decision for our online parameter optimization is *when* to trigger the optimization as well as the size of the utilization history to be used as an input. The history needs to cover both recent trends as well as a short-term history. A fixed window size to capture the characteristics can hardly be defined. It can be either too small with a very high variability or too large with too much smoothing. Therefore, we apply an adaptive window, which automatically chooses a window size between these two extremes by detecting changes of the workload pattern.

As a metric we use the summed CPU load of all hosts in the system. We choose this metric, because it allows to capture a changing input rate and changing stream characteristics like selectivities using a single value. Please note, that we use the individual CPU utilization per host for deriving scaling decisions like presented in Section 2.2.

In this paper we use an established approach for adaptive windowing [4] to detect changes of the workload pattern. The main idea of the adaptive window is that the size of the window increases as long as the observed metrics does not change significantly. If a significant change is detected, elements from the end of the window are removed until the variance of the window is small again.

The algorithm checks on adding a new element, if the overall window has a small variance. Therefore, it calculates all splittings of the window into two non-empty sub-windows and compares the average of these two sub-windows. If the difference of these averages is larger than a threshold ε, the oldest element of the overall window is dropped. This is repeated until for all possible pairs of sub-windows the following condition holds: The difference between both averages is smaller than ε [4]. The threshold ε depends on the current size of the window and a sensitivity value $\delta \in [0,1]$. Assume n be the current size of the adaptive window and n_1 and n_2 the sizes of the two sub-windows. We define m to be the harmonic mean of n_1 and n_2 and $\delta' = \frac{\delta}{n}$. The value of ε can be calculated as:

$$\varepsilon = \sqrt{\frac{1}{2m} \ln \frac{4}{\delta'}}.$$

The length of the adaptive window defines the current size of the utilization history. If the workload pattern does not change significantly, the size of this history increases. If the adaptive window detects a larger change of the average CPU load, the parameter optimization is triggered. In this situation, the length of the history is reduced by dropping the oldest elements, analogously to the adaptive window.

The δ value influences how often the parameter optimization is triggered. A smaller delta value triggers less frequently the parameter optimization. We evaluated different possible δ values (a detailed evaluation is omitted due to space constraints). We observed only a measurable decrease of the performance for extreme values like $\delta \leq 0.1$ and $\delta \geq 0.9$. For both cases, our system exhibits a certain variability of the results. The system shows stable results in the remaining range. Therefore, in the following we use a delta value of 0.2 for all experiments.

5. Evaluation

We implemented the proposed parameter optimization on top of an existing elastic data stream processing engine [12, 13]. In the following, we evaluate the presented solution with real-world data. We study three major questions:

1. Does the online parameter optimization improve the trade-off between monetary cost and latency violations compared to state of the art solutions?
2. Can we influence the results of our parameter optimization by the latency threshold lat_{thres}?
3. Does the online parameter optimization solution scale with an increasing operator count and history size?

To answer these questions, we compare our parameter optimization component with a scaling strategy based on manually tuned thresholds as well as with an adaptive auto-scaling technique based on Reinforcement Learning.

We use three types of input data for our experiments (see Figure 5): three daily traces of tick data recorded at the Frankfurt stock exchange, three different traces of Twitter feeds recorded using the public Twitter API, and three time series of energy data taken from the DEBS challenge 2014 [28]. Each scenario illustrates different workload characteristics: the financial workload changes very frequently, the Twitter presents a workload with a seasonal pattern and random load peaks, and the energy workload has a low average load and unexpected load increases.

For our evaluation we run 10 queries for the energy workload, 20 queries for the Twitter workload, and 35 queries for the financial workload. We choose different query counts for the mentioned workloads to further increase the diversity of the tested scenarios. Each query consists of 3 operators (1 filter, 1 aggregation, and 1 sink) and is generated using a query template (see also Listing 1 for an SQL like description): For the financial workload we use queries to calculate candle sticks (maximal, minimal, average, first, and last value for a time window) per tick symbol. For the Twitter stream we analyze the hashtags of incoming tweets and calculate the currently most frequently used hashtags with a specific prefix. The query for the energy use case calculates the median load of all smart meters for a specific household in the current time interval. Each query instance uses different window sizes $\$ts$ and different sectors $\$sector$ for tick symbols, prefixes $\$prefix$, or households $\$houseID$ respectively. However, we use for all experiments with one workload the same query instances to ensure comparability of the results.

We run our prototype in a private shared cloud environment with up to 12 client hosts and one manager host. Each host has 2 cores and 4 GB RAM.

For each experiment we measure the monetary cost as well as the number of latency violations. For the monetary cost we distinguish between the cost used for the allocated hosts and the cost paid for network traffic and event load. Due to the fact, that network and event load costs are identical for all experiments with the same workload, we denote them as fixed costs.

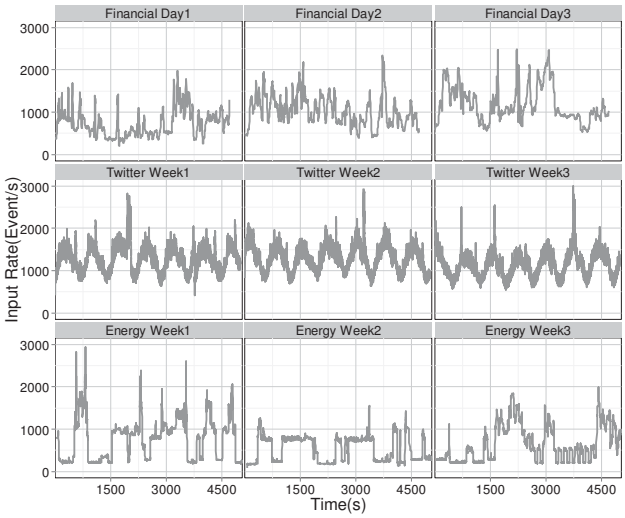


Figure 5: Workload Patterns for Different Scenarios

```
%Financial Template%
INSERT INTO financialOutStream
SELECT compName, AVG(tick),MAX(tick),
MIN(tick), FIRST(tick), LAST(tick)
FROM tickStream WITHIN $ts seconds
GROUP BY symbol WHERE sector = $sector;

%Twitter Template%
INSERT INTO twitterOutStream
SELECT hashTag,COUNT(hashTag) as freq
FROM twitterStream WITHIN $ts seconds
GROUP BY compName ORDER BY freq DESC
Where hashTag LIKE $prefix;

%Energy Template%
INSERT INTO energyOutStream
SELECT houseID,meterID,MEDIAN(value)
FROM energyStream WITHIN $ts seconds
GROUP BY meterID WHERE houseID=$houseID;
```

Listing 1: Used Query Templates

We calculate the monetary cost value according to the current Amazon Kinesis pricing scheme [3]. Amazon Kinesis offers data stream processing capabilities on top of a public cloud environment and charges two main components: The cost for the infrastructure and the cost for using the data management platform. An on demand instance of medium size is sold at a price of $0.120 per hour. Each used instance of the data stream processing engine is charged with $0.015 per hour. In addition, outgoing network traffic is charged with $0.120 per GB and a user needs to pay $0.028 for 1,000,000 *write* commands. For our system each input event is denoted as a *write* command. We adapt the Kinesis cost model slightly for the usage in this paper. We adjusted the billing interval from one hour to one minute, the prices are scaled proportionally. The reason for this adaptation is that (1) a streaming system needs to scale out or scale in within seconds to accommodate peak loads and (2) it allows to capture the system behaviour for our experiments better.

Each experiment runs for about 90 minutes with roughly 2500 measurement points. All performance metrics like CPU load, network consumption, and memory usage are periodically measured

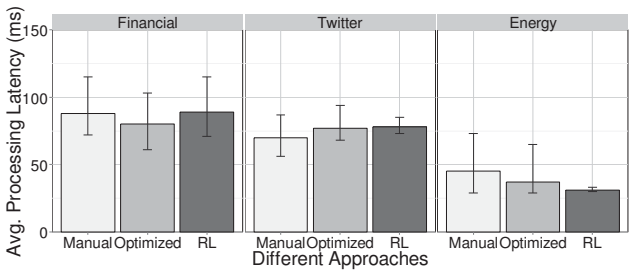


Figure 6: Average Processing Latency of Different Approaches

and averaged over a ten seconds interval. The maximal host count for all measurements within the same minute is used for the cost calculation. We present the total cost, that the user needs to pay for the 90 minutes of the experiment. For a single point we measure the average end to end latency of all running queries. The end to end latency is measured as time between an event enters the query and it leaves the query at the sink. A latency violation is counted, if for a measurement point the average query latency exceeds the latency threshold. We use a value of 3 seconds as an upper threshold for the latency, because a latency larger than 3 seconds only occurs in our prototype, if a large set of operators is moved or a host is overloaded [12].

The average processing latency is not influenced by the used scaling strategy. This is shown in Figure 6, which illustrates the average processing latency per experiment run for the different methods. For all scenarios and strategies this average latency is below 100 ms.

5.1 Definition of the Used Baseline

As a baseline of our experiments, we use a system manually tuned by the user. Therefore, we choose 16 different parameter configurations and measure the results for each configuration and workload individually (see Figure 7). The range of achieved results illustrate the potential performance of a manually tuned system.

The used configurations (see Table 2) are chosen to cover the parameter space of the two most important parameters $thres_{\downarrow}$,$thres^{\uparrow}$ (an entry *X* in the table indicates that the parameter combination is used for the baseline), all remaining parameters are fixed. Parameter configurations with very close lower and upper thresholds (e.g. $(thres^{\uparrow},thres_{\downarrow}) = (0.75,0.5)$) typically create very frequent operator movements, which results in an instable system and many latency violations. Therefore, these configurations are excluded from the baseline.

$thres^{\uparrow} \setminus thres_{\downarrow}$	**0.2**	**0.25**	**0.3**	**0.35**	**0.4**	**0.45**	**0.5**
0.75	X	X	X	X			
0.8		X	X	X	X		
0.85			X	X	X	X	
0.9				X	X	X	X

Table 2: Baseline Parameter Configurations ($thres_{\downarrow}$, $thres^{\uparrow}$)

Based on the results presented in Figure 7 the achievable monetary cost and the number of latency violations vary significantly for different configurations, e.g., for the scenario *Twitter Week 1* the achieved cost ranges between \$0.65 and \$1.20, for the same scenario the number of violations ranges between 4 and 72. While this scenario contains many load peaks, the variability of the achieved cost and latency violations decreases significantly for other scenarios. For the scenario *Twitter Week 2* even 6 configurations create no latency violation at all, because only very few scaling decision

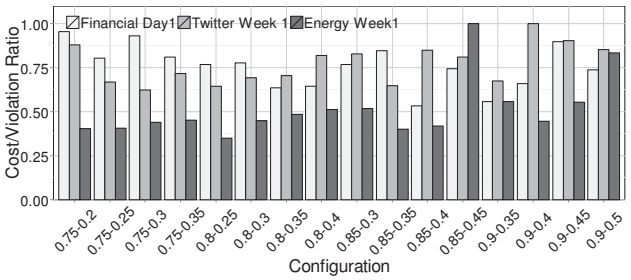


Figure 8: Comparison of Different Configurations for Different Use Cases

(< 10) are done. As a drawback of this result, the achieved cost is up to 100% higher for these configurations compared to the configuration with the minimal costs.

In average over all experiments the maximal measured cost is nearly double as high as the minimal costs (+97%) and the maximal number of latency violations ranges between 3x up to 16x of the lowest count. This high variability outlines how much the measured performance of an elastic data stream processing engine depends on the user configuration.

To judge the achieved trade-off between the monetary cost and latency violations, we normalize the measured cost and latency values based on the maximum observed value of any configuration for each scenario. The average of both normalized values is used as a single metric to compare the achieved trade-off for the different configurations.

In Figure 8 we show the achieved trade-off value (a lower value is better) for all configurations and three scenarios *Financial Day 1*, *Twitter Week 1*, *Energy Week 1*. We are not able to clearly identify rules for tuning the parameters based on these evaluation results. Most configurations perform good only for two or three out of nine scenarios without clear pattern. The two extreme configurations with $(thres^{\uparrow},thres_{\downarrow}) = (0.75,0.2)$ and $(0.9,0.5)$ perform very bad in most scenarios. Two configurations $(0.8,0.35)$ and $(0.9,0.35)$ perform quite well in average, but also show bad results for *Energy Week 3* or *Energy Week 1* respectively. These results illustrate (1) the divergence of the presented evaluation scenarios and (2) the difficulty of manually identifying a good parameter configuration.

Based on these trade-off values, we select different configurations to compare the manually tuned system with our parameter optimization:

Naive Strategy We use the median result of all measured configurations for each workload to illustrate the expected results for an unexperienced user without knowledge on how to set the parameters.

Top 3 Per Workload As a second baseline, we determine the best three configurations per workload for all nine scenarios and average their results. This illustrates the best achievable performance for a very carefully tuned system.

The performance of the best single configuration $(0.85,0.35)$ is between the two strategies *Naive Strategy* and *Top 3 Per Workload* based on our experimental results and therefore not shown in the following.

5.2 Comparison to the Baseline

We illustrate the results for the two baselines and for our parameter optimization component in Figure 9. For the parameter optimization we present the average results of three different runs.

The parameter optimization outperforms in all tested scenarios the naive selection. The parameter optimization reduces the cost at least by 4% for *Twitter Week 2* to 30% for *Financial Day 1*. In aver-

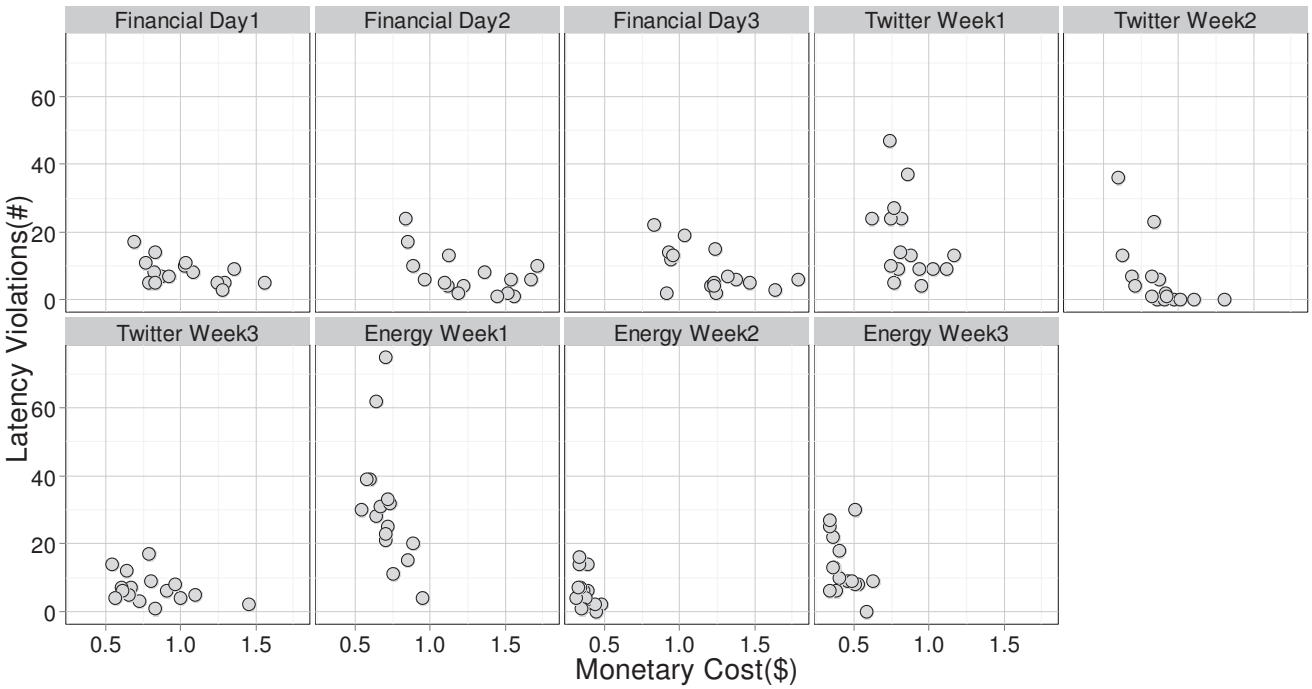


Figure 7: Evaluation Results for Manually Tuned Thresholds

age, for the financial scenario a cost saving of 24%, for the Twitter scenario a saving of 18% and for the energy scenario a saving of 14% is measured compared to the naive strategy. For the financial use case the number of latency violations slightly increases (in average 3 violations more), where else a slight decrease of the number of violations is measured for the Twitter and energy workload (in average 3 or 5 violations less). In average over all scenarios, 19% costs can be saved with the same number of latency violations.

The cost with the parameter optimization is reduced by 10% in average compared to the best configuration per workload. The number of latency violations increases only by in average 3 violations compared to this baseline.

While the improvement of the utilization or violation values depends on the scenario, our parameter optimization significantly reduces the variability of the results. The measured cost varies only by 10% and the maximal latency violation count is only two times the minimal violation count. In contrast to the manually tuned system, our parameter optimization shows stable performance independent of the scenario and the assistance of the user.

We conclude from the presented results, that our parameter optimization performs at least as good as a carefully tuned threshold-based scaling strategy. Our system finds good parameter configuration online without additional effort by the end user.

5.3 Comparison to Reinforcement Learning

We compared our parameter optimization component also with an auto-scaling approach based on Reinforcement Learning. The deployed implementation [13] uses as an input only utilization values to learn the right point to scale. This allows us to contrast our parameter optimization approach, which uses a white-box optimization approach, to an adaptive black-box approach without detailed knowledge about the internal scaling behavior.

The performance of the used Reinforcement Learning depends on three parameters: the used utilization target and two parameters influencing the learning of the controller (learning rate and importance) [13]. We illustrate the importance of these learning parameters by evaluating a fixed utilization target of 0.8 with nine different learning configurations. The results are presented in Figure 10 as average of three runs per configuration. The achieved monetary cost for all configurations shows a small variation, which illustrates that the Reinforcement Learning adapts towards a certain scaling behaviour. The number of latency variations shows a measurable variation, for the scenarios *Financial Day 2*, *Twitter Week 2*, and *Energy Week 1* the maximal number of violations is two times higher than the minimal count. Similar to the manually tuned thresholds, the best learning configuration differs between the scenarios. Therefore, we determine analogously a best configuration and the average result of all configurations for comparing the Reinforcement Learning with the Parameter Optimization.

The results of these two baselines for the Reinforcement Learning are illustrated also in Figure 9. Overall, the Reinforcement Learning shows low cost values: the achieved cost values average over all configurations are between 24% smaller for *Energy Week 1* to 6% larger for *Financial Day 3* than the values of our presented parameter optimization component. The monetary cost for the Reinforcement Learning can be improved by 6% in average over all scenarios. However, at the same time the number of latency violations significantly increases. In average, Reinforcement Learning shows 14 additional latency violations, which demonstrates the aggressive scaling realized by it. In some scenarios the number of violations more than doubles compared to the parameter optimization, e.g., for *Energy Week 1* the number of violations increases from 16 to 42. If we choose the best learning configuration for the Reinforcement Learning per scenario, the number of latency violations still increases by in average 6 violations compared to our Parameter Optimization.

The results show, that our parameter optimization component achieves a better trade-off between monetary cost and quality of service than existing black-box optimization approaches. The in-

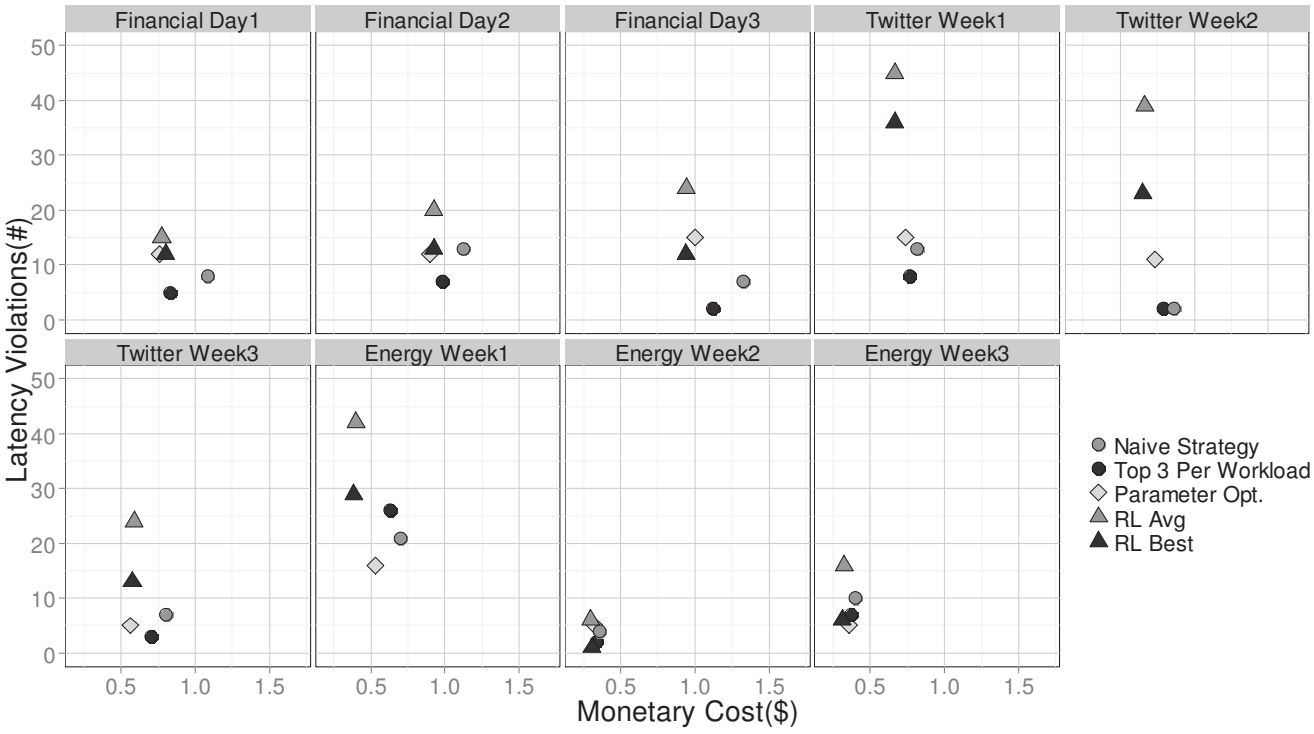


Figure 9: Comparison of Manual Tuned Thresholds, Reinforcement Learning, and Parameter Optimization

creased level of detail allows our white-box model to outperform existing black-box optimization approaches.

5.4 Different Latency Thresholds

Our presented elastic scaling as well as the parameter optimization can be configured with an upper threshold lat_{thres} for a latency peak created by a scaling decision like described in Section 3.3 or our previous paper [12], respectively.

For the previously presented evaluation results we used for both approaches a latency threshold of 3 seconds, because it has been shown to be a good discriminator for large scaling decisions in our system [12].

In the following, we like to study if we can vary the trade-off between the number of violations and the spent monetary cost based on the latency threshold. Therefore, we configure our parameter optimization component with a latency threshold of 2, 3, 4, 5, or 6 seconds. In Figure 11 the monetary cost and the number of latency violations are shown as the average of three runs (the error bars indicate the variance of the measurements). The number of violations in the charts describes measurements larger than 3 seconds to ensure comparability of the presented results.

For all scenarios, with an increasing latency threshold the monetary cost can be further reduced. In the scenario *Energy Week 1* the additional cost saving is up to 16% for a latency threshold of 6 sec.. However, at the same time the number of latency violations increases by 27. The saving potential depends on the scenario, for some workloads the values for the different latency thresholds are nearly unchanged. These scenarios like *Energy Week 1* and *Twitter Week 2* showed also in the other measurements the smallest variability in both the monetary cost and latency violations.

Overall, we are able to create a scaling strategy using higher latency thresholds, which optimize the monetary cost similarly to the presented Reinforcement Learning. For a latency threshold of 6 sec. our parameter optimization shows in average the same cost improvement like the Reinforcement Learning, where the number of latency violations of the parameter optimization is still smaller (7 latency violations less).

The presented results indicate that our parameter optimization is able to reduce the monetary cost with an increasing latency threshold, because parameter configurations which scale more frequently can be chosen. At the same time the number of latency violations increases. This illustrates that a user is able to tune the trade-off between monetary cost and latency violations depending on the requirements of his use case by configuring a single parameter.

5.5 Scalability

Finally, we study the scalability of our approach. The presented parameter optimization runs with a fixed upper bound on the number of samples. Therefore, the scalability of the approach mostly depends on the time to evaluate the cost function for a parameter configuration p_i.

The evaluation time of the cost function is influenced mostly by two metrics: the number of placed operators and the length of the utilization history. In Figure 12 we illustrate the average evaluation time for the different use cases. The different use cases also use a different number of queries, which allows us also to illustrate the scalability with the number of operators.

From the presented measurements we conclude that our cost function scales linearly with the window size for all three scenarios. If the window size is increased by a factor of 6, from 30 seconds to 180 seconds, the average runtime only increases by a factor of 2.5, 1.4, or 3.0 for the energy, Twitter or financial workloads respectively.

Similar results can be observed if we compare for a fixed window size the runtimes for different query counts: The runtime for increasing the number of queries from 10 for the energy use case

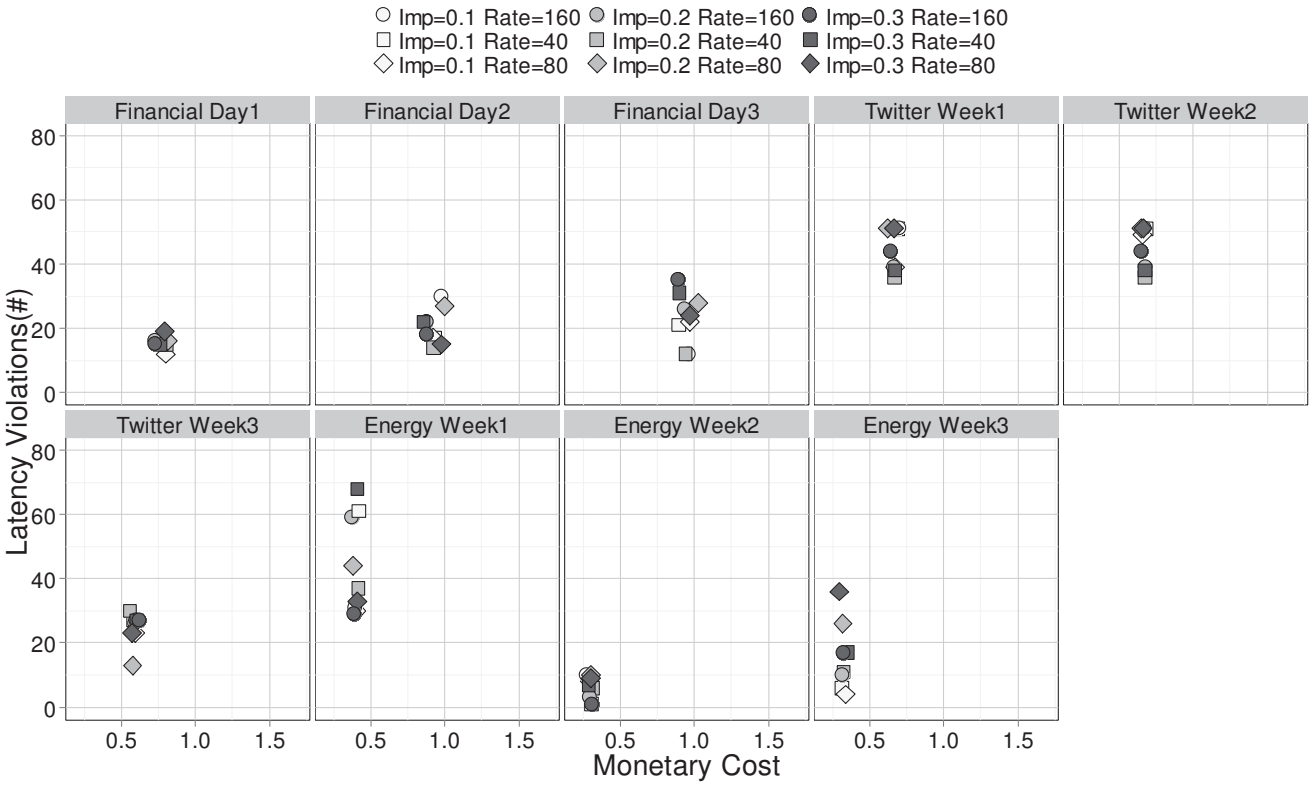


Figure 10: Evaluation Results for Reinforcement Learning

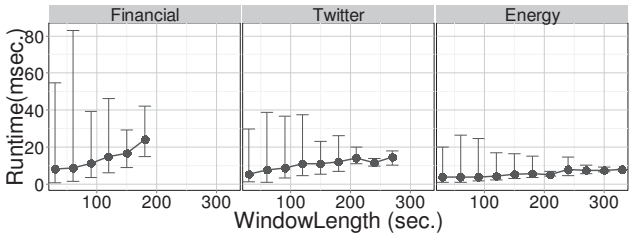


Figure 12: Scalability of the Online Parameter Optimization

to 35 queries for the financial use case, results only in a runtime increase by in a factor of 1.8 in average over all window sizes.

Overall, we conclude that our prototype scales nearly linearly with both the number of queries and the window size.

5.6 Discussion

The presented evaluation results show that our parameter optimization component achieves a good trade-off between monetary cost and the number of latency violations. The results of our component are as good as or even better than a manually tuned configuration without the time consuming and error-prone task of manual tuning the system. While black box optimization approaches like a Reinforcement Learning are able to reduce the monetary cost compared to fixed thresholds, only our parameter optimization component achieves a good trade-off between the monetary cost and the quality of service.

Our prototype proved its scalability and in addition can be configured by an experienced user to control the trade-off between monetary cost and number of latency violations. However, this step is optional, in contrast to the mandatory configuration of the user-defined threshold-based approach.

The present parameter optimization can be used as an extension for any elastic scaling data stream processing system [8, 11], which uses a threshold-based scaling strategy. The implemented parameter optimization and the online profiler have been realized in less than 10000 lines of code and can be reused without any modification. Only the cost function may require small changes to reflect system specifics.

6. Related Work

Related concepts to our work can be found in the area of resource scaling for cloud-based data management systems. Various authors also studied the problem of elastic scaling for data stream processing systems. In the following, we outline similarities and differences between these systems and our approach.

Elastic Data Stream Processing Systems Different authors [8, 9, 11, 22] studied the problem of elastic scaling a data stream processing system. Gulisano et al. [11] presented an architecture for scaling out and in a data stream processing system with a varying load. The focus of their work was to introduce the required extensions of state of the art streaming systems including operator movement and a basic elastic scaling manager. Fernandez et al. [8] presented a new notion of operator state management, which is used in their system to scale dynamically to a varying number of hosts and provide fault tolerance.

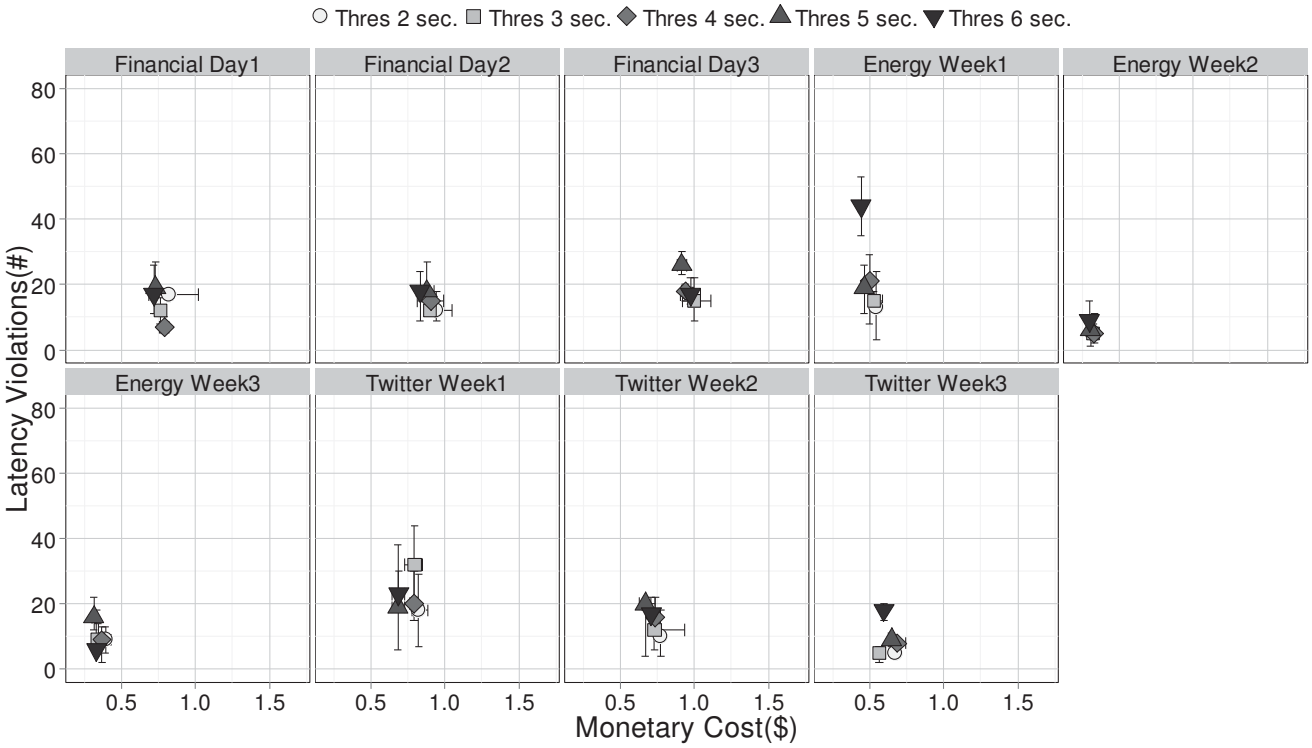


Figure 11: Influence of Used Latency Threshold

Schneider et al. [22] and Gedik et al. [9] studied, how to determine the optimal level of parallelism for a stateless or a stateful operator respectively. They presented an approach to define automatically the optimal parallelism level and adapt it during runtime.

None of the presented approaches studied how to choose the optimal resource allocation scheme automatically like presented in this paper. In contrast, these systems are based on fixed threshold parameters or manually tuned controllers like the solution used as a baseline in our evaluation.

Resource Provisioning for Elastic Data Management Platforms The resource scaling for data management systems can be differentiated based on the type of provisioning: auto-scaling techniques for instance-based systems [13, 18, 23] or provisioning schemes for task-based systems [7, 15].

Auto-scaling techniques [18, 23] use different algorithmic approaches like thresholds, control theory, Reinforcement Learning, queuing theory, or workload prediction for deriving the right point in time to scale. In our previous work [13] we showed, that only adaptive approaches without any assumption on workload characteristics show good performance for data stream processing systems, because the workloads have a high variability and change in an unpredictable way. In addition, we showed that black-box approaches improve the utilization and, as a consequence, monetary cost of the system. However, they often need to be tuned towards a specific workload, have difficulties to handle non-stationary workloads, and can not model the resulting quality of service easily [13].

For MapReduce and similar systems, for each new task certain parameter settings like the number of mappers, reducers, and used hosts can be tuned before the task execution. State of the art approaches use either techniques based on white-box optimization [15] or the history of previous configurations to derive the right parameter set [7]. Such approaches typically rely on traces and input samples for new tasks. Our approach is an extension of these approaches for data stream processing techniques. Specifically, it provides (1) an online parameter optimization approach including an optimization trigger and a novel cost function, and (2) a quality of service aware parameter optimization.

Adaptive Resource Provisioning Although, the used bin packing is a well-established approach for placing a set of instances to a minimal number of hosts, different approaches for placing virtual machines with adaptive resource demands have been studied in recent time [10, 20, 24]. The types of algorithm applied in this context include online bin packing [24], vector packing [10], and min-cut graph partitioning [20]. An integration of one of these approaches into the presented solution is an interesting future work.

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 used thresholds, which is an error-prone task and requires detailed knowledge about the workload.

In this paper we introduce an online parameter optimization approach, which automatically adjusts the scaling strategy of an elastic scaling data stream processing system. Our system minimizes the monetary cost for the user and at the same time keeps the number of latency violations low. For the presented real-world scenario our system is able to reduce the cost by 19% compared to a naive parameter selection by a non-experienced user and 10% compared to a manually tuned scaling strategy. At the same time, we are able

to influence the trade-off between monetary cost and latency violations using a single parameter.

For the future, we plan to investigate the proposed model for other data stream processing use cases and larger setup sizes. Our current evaluation lacks workloads with load bursts (increase of the event rate by several orders of magnitude). While such loads are already a major challenge for the overload management of the underlying data stream processing system, an evaluation of the different scaling approaches would be interesting. In addition, we like to improve our latency-aware cost function. Especially, the current modeling of overload situations could be enhanced, which would allow to improve the performance of the parameter optimization even further.

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 *Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, CIDR 2005, pages 277–289, 2005.

[2] Amazon. Amazon Auto Scaling. http://aws.amazon.com/autoscaling/.

[3] Amazon. Amazon Kinesis. http://aws.amazon.com/de/kinesis/.

[4] A. Bifet and R. Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the Seventh SIAM International Conference on Data Mining*, SDM 2007, pages 443–448, 2007.

[5] 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*, pages 46–93. PWS Publishing Co., 1996.

[6] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ASPLOS 2014, pages 127–144. ACM, 2014.

[7] M. Ead, H. Herodotou, A. Aboulnaga, and S. Babu. PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs. In *Proceedings of the 17th International Conference on Extending Database Technology*, EDBT 2014, pages 1–12, 2014.

[8] R. C. 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 SIGMOD International Conference on Management of Data*, SIGMOD 2013, pages 725–736. ACM, 2013.

[9] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1447–1463, 2014.

[10] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 455–466. ACM, 2014.

[11] 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)*, 23(12):2351–2365, 2012.

[12] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS 2014, pages 13–22. ACM, 2014.

[13] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops*, ICDEW 2014, pages 296–302. IEEE, 2014.

[14] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC 2013, pages 23–27, 2013.

[15] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[16] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. SQPR: Stream query planning with reuse. In *Proceedings of the 27th International Conference on Data Engineering*, ICDE 2011, pages 840–851. IEEE, 2011.

[17] U. Lampe, R. Hans, M. Seliger, and M. Pauly. Pricing in infrastructure clouds - an analytical and empirical examination. In *Proceedings of the 20th Americas Conference on Information Systems*, AMCIS 2014, 2014.

[18] 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.

[19] S. Martello and P. Toth. Algorithms for knapsack problems. *Surveys in Combinatorial Optimization*, 31:213–258, 1987.

[20] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of 2010 IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[21] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC 2012, page 7. ACM, 2012.

[22] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *IPDPS 2009: Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing*, IPDPS 2009, pages 1–12. IEEE, 2009.

[23] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the second ACM Annual Symposium on Cloud Computing*, SoCC 2011, pages 1–14. ACM, 2011.

[24] W. Song, Z. Xiao, Q. Chen, and H. Luo. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2014.

[25] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the SIGMOD International Conference on Management of Data*, SIGMOD 2006, pages 407–418. ACM, 2006.

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

[27] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS 2003, pages 196–205. ACM, 2003.

[28] H. Ziekow and Z. Jerzak. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS 2014, pages 266–269. ACM, 2014.