2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00100

AuTraScale: An Automated and Transfer Learning Solution for Streaming System Auto-Scaling

Liang Zhang, Wenli Zheng, Chao Li, Yao Shen, Minyi Guo
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
zhangliang@sjtu.edu.cn, {zheng-wl, lichao, yshen, guo-my}@cs.sjtu.edu.cn

***Abstract*—The complexity and variability of streaming data have brought a great challenge to the elasticity of the data processing systems. Streaming systems, such as Flink and Storm, need to adapt to the changes of workload with auto-scaling to meet the QoS requirements while saving resources. However, the accuracy of classical models (such as a queueing model) for QoS prediction decreases with the increase of the complexity and variability of streaming data and the resource interference. On the other hand, the indirect metrics used to optimize QoS may not accurately guide resource adjustment. Those problems can easily lead to waste of resources or QoS violation in practice.**

To solve the above problems, we propose AuTraScale, an automated and transfer learning auto-scaling solution, to determine the appropriate parallelism and resource allocation that meet the latency and throughput targets. AuTraScale uses Bayesian optimization to adapt to the complex relationship between resources and QoS, minimizing the impact of resource interference on the prediction accuracy, and a new metric that measures the performance of operators for accurate optimization. Even when the input data rate changes, it can quickly adjust the parallelism of each operator in response, with a transfer learning algorithm. We have implemented and evaluated AuTraScale on a Flink platform. The experimental results show that, compared with the state-of-the-art method like DRS and DS2, AuTraScale can reduce 66.6% and 36.7% resource consumption respectively in the scale-down and scale-up scenarios while ensuring QoS requirements, and save 13.5% resource on average when the input data rate changes.

***Index Terms*—Streaming system, Auto-scaling, Bayesian optimization**

I. Introduction

Streaming data is playing an increasingly important role. To deal with the streaming data that arrives at the processing system at a fast, and time-varying rate, some stream processing frameworks have emerged, such as Flink [1], Spark [2], and Storm [3]. As the input data rate changes frequently, the elasticity becomes a critical attribute in a streaming system, which needs to adjust the amount of allocated resources in a timely manner to adapt to workload changes.

Usually, the resource allocation in a streaming system is highly correlated with the system-level or application-level parallelism parameters. Traditionally, those parameters are set by manual tuning, which takes too much time and easily results in sub-optimal configurations. Therefore, automatic on-demand scaling technologies have become necessary to find optimized configurations avoiding both resource waste and QoS violation.

In the past few years, many auto-scaling solutions have been proposed for streaming system elasticity. Among them, the solutions to meet the latency demand mainly rely on the prediction ability of the queueing models [4]–[6]. But their prediction accuracies and convergence rates can be significantly affected when stream processing jobs co-run on the same machine and interfere with each other. The auto-scaling solutions to meet the throughput demand are usually based on some indirect metrics such as backpressure (or congestion) [7] [8], queue size [7] [9], and observed rate [8], [10]–[13]. These metrics can not represent the true processing performance of operator instances, which may lead to inaccurate or incorrect decision-making. The recently proposed auto-scaling solutions based on the dataflow model, notably DS2 [14], can optimize throughput efficiently by analyzing the true processing rates of operator instances. However, their approach of determining the parallelism assumes the processing performance of an operator linearly increases with the number of its instances, which may become inaccurate when the increased instances lead to more inference. In addition, most of the existing methods do not adjust their models online, and thus the models might be inaccurate at some moments during the long-running of stream processing, e.g., when the data rate significantly changes.

Therefore, we propose AuTraScale, an automated and transfer learning solution for streaming system auto-scaling. It finds the optimal configuration of parallelism parameters quickly and accurately to guarantee meeting multiple performance requirements, and can continuously optimize the accuracy online. AuTraScale provides four key features. First, AuTraScale uses the Gaussian process to model the relationship between the QoS and operator parallelism. In contrast to a queueing model, the Gaussian process model can cover the impact of interference and avoid accuracy degradation. Second, to optimize the model accuracy continuously, AuTraScale uses the Bayesian optimization method to iteratively update the model, and recommends the optimized configuration in time. Third, AuTraScale uses the transfer learning method to accelerate the convergence of optimal configuration at a new rate of data. Fourth, it uses the true processing rate metric to optimize throughput more accurately and reduce the pending time of data before being processed.

We implement and evaluate AuTraScale based on the Flink framework. Compared with the state-of-the-art method, AuTraScale reduces 66.6% and 36.7% resource usage respectively in the scale-down and scale-up scenarios while ensuring QoS requirements and save 13.5% resource on average when the data rate changes. Our main contributions in this work are as follows:

- We abstract the relationship between the parallelism and QoS in streaming systems to a Gaussian process model, which is trained by the data containing the interference due to resource contention, so the prediction results can be more accurate.
- We propose an automated learning auto-scaling solution based on Bayesian optimization. It iteratively updates the Gaussian process model with a scoring function, to dynamically evaluate the impacts of different parallelism parameters on the overall performance. It also uses an acquisition function to trade off between exploitation and exploration.
- We propose a transfer learning algorithm to make full use of the trained model to find the optimal configuration quickly at a new rate of data. It greatly reduces the costs of training a new model.

The rest of the paper is organized as follows. In Section II, we discuss the challenges and opportunities of resource allocation in the streaming system. Then, we present the details of AuTraScale in Section III and an overview of the system architecture in Section IV. Experimental results are presented and analyzed in Section V. We discuss related work in Section VI and conclude the paper in Section VII.

II. Motivation

In this section, we identify the challenges of auto-scaling in current streaming systems via case studies, and introduce the opportunities brought by the Bayesian optimization algorithm to address these challenges.

A. *Case Studies and Challenges*

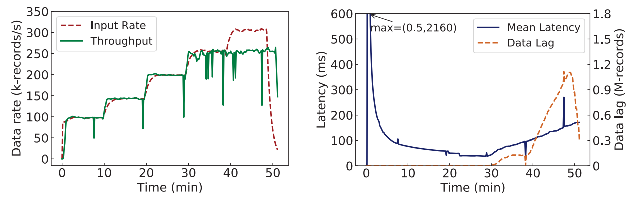
Different operator tasks in the streaming system cannot be completely isolated, so resource contention is inevitable. Therefore, the performance of a job (e.g., the throughput) may not be linearly related to the amount of resources allocated to it. We conduct a few experiments on a Flink cluster to explore the complexity of this relationship and discuss its challenges to auto-scaling.

CASE 1: Fixed parallelism, increasing data rate.

We run a WordCount Streaming job on the Flink cluster, whose data come from Kafka. The parallelism of each operator is set to 2 and remains the same. The input data rate starts from 100k records/s and increases every 10 minutes by 50k records/s. We use Kafka metrics and Flink metrics to obtain relevant runtime information. Fig. 1(a) shows the changes in both the input data rate and throughput, and Fig. 1(b) shows the changes in the end-to-end latency in Flink and the data lag in Kafka during the experiment.

Observation 1: When the input data rate increases and exceeds the upper bound of the throughput with the fixed parallelism configuration, data are accumulated in Kafka, and the end-to-end latency in Flink increases.

As shown in Fig. 1(a), when the input data rate is 100k records/s, 150k records/s, or 200k records/s, the throughput of the job can meet the input data rate and there is no data accumulation in the Kafka. In Fig. 1(b), the average end-to-end latency temporarily peaks when the job just starts and then it gradually flattens out. When the input data rate reaches 250k records/s, the job throughput begins to lag behind for a short time. The data begin to accumulate in Kafka, and the end-to-end latency begins to increase. Then the input data rate increase to 300k records/s, which is significantly greater than the processing capacity of the fixed parallelism configuration, while the throughput of the job still maintains at 250k records/s. The growth rate of the accumulated data in Kafka and the average end-to-end latency of data in Flink further increase until the test ends at the 50th minute.



(a) Throughput (b) Latency and data lag

Fig. 1. **The running results of the WordCount Streaming job with the fixed parallelism and increasing data rate.** Parallelism under-provisioning leads to non-trivial data lag and end-to-end latency.

This case shows that it is necessary to adjust the parallelism configuration as well as the resource allocation when the data rate changes. Lack of resources will result in data processing lag and increased latency, while redundant resources can be wasted if the data rate decreases. However, dynamically allocating resources for jobs is challenging, as shown by another set of our experiments as follows.

CASE 2: Fixed data rate, increasing parallelism.

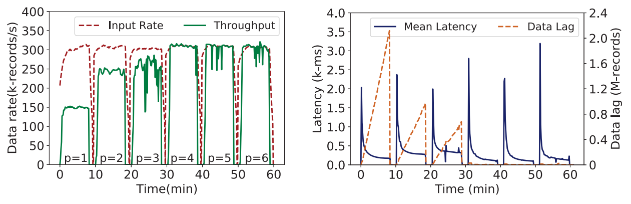
The experiment consists of six independent small tests. The experimental environment is the same as CASE 1, except for that the input data rate in each test maintains at about 300k records/s, and the operator parallelism in the 6 small tests is (1, 2, 3, 4, 5, 6) respectively. The results are shown in Fig. 2.

Observation 2.1: The relationship between operator parallelism and job throughput is not linear.

In Fig. 2(a), in the first three sets of tests, the parallelism is 1, 2, and 3, but the corresponding throughput is about 150k records/s, 250k records/s, and 275k records/s respectively. The multiplying growth of parallelism does not provide the proportional increase in throughput. The reasons for this phenomenon can be synchronization and resource competition between different operator instances.

Observation 2.2: The appropriate parallelism brings latency benefits, and a higher parallelism may not be better.

Comparing the latency between the first three tests and the last three tests in Fig. 2(b), we find that improving the parallelism is helpful to reduce the latency and data lag on the whole. However, the latter two tests (The latencies are 100 ms and 125 ms respectively) also indicate that the increase of parallelism may increase communication cost [15] and thus increase the latency.



(a) Throughput (b) Latency and data lag

Fig. 2. **The running results of the WordCount Streaming job with the fixed data rate and increasing parallelism.** p represents the operator parallelism. 1) There is a nonlinear relationship between the parallelism and the throughput. 2) The appropriate parallelism brings latency benefit, but a higher parallelism may not gain lower latency.

The relationship between the parallelism (resource allocation) of operators, latency, and throughput is greatly complicated by synchronization, resource contention, communication, and system scheduling policies. The main challenge to identify the appropriate parallelism configuration when the workload changes is how to deal with this unknown relationship.

B. *Opportunities*

To deal with the above challenges, we need a method that can capture the unknown relationship between resource allocation and QoS metrics (e.g., throughput and latency), and recommend the most appropriate parallelism configuration for the application. Bayesian optimization (hereinafter referred to as "BO") is what we need. First, it uses the surrogate models, such as Gaussian process regression and random forests, to approximate the real relationship, and does not require the specific mathematical formula about the objective function and the input variables. Second, the goal of BO is to find the input variables that optimize the objective function with as few rounds of executions as possible. This is also consistent with the goal of minimizing reconfiguration overheads caused by scaling decisions in streaming systems. Therefore, this black-box method to solve optimizing objective functions is very suitable for auto-scaling scenario in the streaming system.

Mathematically, Bayesian optimization can be expressed by the following formula:

$$x^* = \arg \max_{x \in A \subset \mathbb{K}^d} f(x) \tag{1}$$

AuTraScale takes the parallelism of each operator as the input variable x, and takes a scoring function that quantifies the comprehensive benefits of service quality and resource usage with the given parallelism as the objective function f. The goal of AuTraScale is to find the most profitable parallelism configuration scheme to maximize the scoring function in the minimum number of iterations. In this paper, Bayesian optimization also involves the selection of the surrogate model, acquisition function and the initial training data, as elaborated in Section III.

III. Model and Algorithm Design

AuTraScale optimizes the parallelism configuration with Bayesian optimization and transfer learning, based on the data flows across the directed acyclic graph (DAG) of the stream processing job. The Bayesian optimization algorithm can iteratively update benefit model, which quantifies the comprehensive benefit of QoS and resource usage given the complicated relationship between them, and recommend an appropriate operator parallelism configuration when the input data rate is steady. Since the benefit model is binded to the specific input data rate, when the rate changes, the transfer learning algorithm quickly adjusts the parallelism using the existing model at a new rate. This section elaborates the details of AuTraScale. For concise presentation, we summarize the frequently used notations throughout the paper in Table I.

TABLE I
Symbols used in this paper.

Symbol	Meaning
v_e	The input data rate of a streaming job
$\bar{v}_i$	The average true processing rate of operator i
λ_i^*, v_i^*	The total true input, processing rate of operator i
$\boldsymbol{k'}$	A vector $(k'_1, ..., k'_N)$ consisting of optimal parallelisms of all operators when the throughput is maximized
$\boldsymbol{k}$	A vector $(k_1, ..., k_N)$ consisting of current parallelisms of all operators in a job
N	The number of operator in a DAG
P_{max}	Upper limit of parallelism under the current resources
$\mu(x)$	The expected value of x
$\sigma(x)$	The variance of x
l_t	Target latency of records in a job
l_r	Average latency of records in a job
s_l	The resource score threshold of a job
$\mathcal{M}$	Gaussian Process model
Ω	The search space of Bayesian optimization algorithm

A. *Assumptions*

The streaming systems manage resources like CPU and memory. Specifically in Flink, these resources are encapsulated in the form of slots, which are fixed subsets each worker's resources are divided into. Managed memory isolation is implemented between slots, but there is no CPU isolation. Data arriving at an operator is assigned to its instances each in a different slot by specific rules. We assume each instance of the same operator has the same amount of data, like [5]. We also assume that the network does not bottleneck the system, which benefits from the advanced network hardware, such as 10G Ethernet and InfiniBand. We also assume that the cluster resources are enough to support the QoS requirements of the workload. It means there will not be a situation in which all resources are exhausted, but the QoS is still unable to meet.

B. *Operator performance model*

In the auto-scaling scenario, to optimize the parallelism and resource allocation, the actual processing performance of the operator is usually defined as the ratio of the number of processing records over the processing time. However, the observed data processing time of an operator often contains a large amount of waiting time due to blocking, and hence cannot reflect the actual processing ability of the operator accurately. Therefore, we refer to the concept of the true rate in DS2 [14]. The true processing rate of an instance of an operator is:

$$v = \frac{R}{T_u} \tag{2}$$

where R is the total number of records processed by an operator instance in a period of time T, and T_u represents the time used to process data in a period of time T. T_u includes three parts: the deserialization time, processing time, and serialization time. In the rest of the paper, we use the true processing rate as defined above to indicate the performance of the operator.

C. Throughput optimization

For a stream processing job, the throughput that can not meet the input data rate results in data lag and latency increase. Therefore, a successful auto-scaling strategy needs to adjust the resource configuration quickly to make the throughput as close to the input data rate as possible.

AuTraScale refers to the idea of DS2 [14] and uses the true processing rate of the operators mentioned above to optimize the throughput. Suppose that operator $i-1$ and operator i are two connected operators in the DAG. The operator $i-1$ only contains one instance and its total true processing rate v^*_{i-1} is equal to the external input data rate v_e and total output rate o^*_{i-1} at time t. The operator i is the successor of the operator $i-1$ in a DAG and its total data input rate is λ^*_i. There is a simple equivalence, $\lambda^*_i = o^*_{i-1} = v^*_{i-1}$. To make the total true processing rate of the operator i catch up with the input data rate, the number of instances of operator i can be set to $\left\lceil \frac{v^*_{i-1}}{\bar{v}_i} \right\rceil$ ($\bar{v}_i$ is the average true processing rate of all the instances of operator i).

However, for an operator, its parallelism is usually greater than one and the total true processing rate may not meet the external input data rate in practice. For this general case, AuTraScale uses the following formula to calculate the optimal parallelism of each operator in each step of iteration:

$$k'_i = \begin{cases} \left\lceil \frac{v_e}{\bar{v}_1} \right\rceil & i = 1 \\ \left\lceil \frac{v^*_{i-1} \cdot \times \frac{k'_{i-1}}{k_{i-1}}}{\bar{v}_i} \right\rceil & i > 1 \end{cases} \tag{3}$$

where k'_i and k'_{i-1} is the optimal parallelism of operator i and operator $i-1$ at the current step of iteration.

The advantage of this approach is to quickly and accurately find the minimum resource configuration that enables the throughput to meet the requirements. But the throughput is often constrained by other factors and can not meet the input data rate in practice (see the results of the Yahoo Streaming job in Section V for details). The DS2 method does not address this issue and thus can fall into an infinite loop due to that the throughput does not reach the target value. Hence AuTraScale adds a new termination condition that two consecutive identical recommended configurations occur during the throughput optimization process.

It is worth noting that the potential benefit of throughput optimization is to minimize event-time latency. Unlike the processing latency we observe in the case studies in Section II, event-time latency is defined as the interval between a tuple's event-time and its output time from the stream processing system [16]. Therefore, event-time latency includes the pending time of data in Kafka and the processing delay in streaming systems. The above algorithm can find the configurations that maximize throughput and reduce data lag, so it is also the optimal solution for reducing pending time. To minimize the event-time latency, AuTraScale uses the output of the above throughput optimization algorithm as the minimum operator parallelism in the following algorithms.

D. Preconditions for model training

Bootstrapping samples selection. As mentioned above, to optimize the event-time latency, AuTraScale takes the parallelism that maximizes throughput as the basic condition to optimize the processing latency and resource usage using Bayesian optimization. It means that the search space of the BO algorithm is limited between the optimal configuration of throughput and the maximum allowable parallelism of the system. AuTraScale also selects bootstrapping samples for the surrogate model of the BO algorithm within this scope.

There are two types of samples in the initial training set of AuTraScale. 1) *All operators in a sample have the same parallelism, and different samples have different parallelisms.* First, the parallelism of all operators is set to the maximum value k'_{max} of the optimal parallelism for throughput. Then we divide the remaining parallelism (the difference between the current parallelism k'_{max} and the maximum allowable parallelism P_{max} of the system) into $M-1$ parts, each of which is called an *interval*. The parallelism of all operators in the i-th sample is set to $k'_{max}+i \times interval$. Those M samples can help the BO algorithm perceive the QoS results of different configurations, and also help us to determine whether the current resources can meet the QoS requirements. 2) *The parallelism of only one operator is set to P_{max}, and the parallelism of other operators is kept in the basic configuration.* There are N such samples (where N is the number of operators in a DAG) that can make the BO algorithm capture the different impact of each operator on QoS as far as possible and have a more precise decision. AuTraScale uses these samples to help the BO algorithm find a near-optimal solution faster.

Scoring function. In addition to throughput optimization, AuTraScale is designed to also keep the latency below the target threshold while ensuring that resources are not over-allocated. Traditional latency-sensitive resource allocation solutions often regard the latency as the sole optimization goal of the model and rely on a greedy way to allocate resources from low to high to avoid over-allocation of resources. AuTraScale jointly optimizes these two objectives with the help of a scoring function.

AuTraScale's scoring function comprehensively quantifies the benefits of latency and resource usage of each parallelism allocation scheme and uses the output value to train the Gaussian process model. The score function needs to satisfy two basic rules: (a) the lower the latency, the higher the score; (b) the closer the parallelism is to the basic configuration (the parallelism for maximizing throughput), the higher the score. So AuTraScale defines the scoring function as:

$$F = \alpha \times \min\left(1.0, \frac{l_t}{l_r}\right) + (1-\alpha) \times \frac{1}{N} \times \sum_{i=1}^{N} \frac{k_i'}{k_i} \tag{4}$$

where k_i' represents the minimum parallelism of operator i that can maximize throughput. k_i represents the current parallelism of operator i. l_r is the average processing latency of data with the current configuration. l_t is the target latency. The first half of the formula is used to judge whether the current latency meets the requirements, and the second half is used to prevent over-provisioning of parallelism. α is an elastic parameter indicating the relative importance of the two targets.

E. Bayesian optimization method at a steady rate

Here we introduce in detail how AuTraScale uses Bayesian optimization to update the model and recommend the appropriate parallelism when the input data rate is steady. The overall workflow is shown in Algorithm 1. This algorithm aims to make streaming jobs meet the latency constraint and minimize resource usage in the scale-up and scale-down situations.

The BO method of AuTraScale needs an appropriate scoring function to evaluate the comprehensive performance benefits of different configurations and a surrogate model to fit the relationship between the parallelism of operators and the performance benefits. When the current resource is over-provisioned or the QoS violation occurs, the acquisition function of AuTraScale will recommend new parallelism samples for the next run of the job. Then, the model is updated using the current metric information. If the termination condition is not met, AuTraScale will iteratively perform the *recommend-run-judge* process. We will introduce the key parts involved in the above process as follows.

Surrogate Model. AuTraScale uses the Gaussian process (GP) model with the Matern covariance kernel as the surrogate model. Compared with others like the random forest, its extrapolation quality is better. The Gaussian process does not need to make assumptions about the relationship between parameters and objective function in advance. This attribute makes the training of the underlying model more flexible.

Acquisition function. The acquisition function aims to find the next sample that is closer to the optimal solution. It also balances the proportion of exploration and exploitation during the sampling period. In the stream computing scenario, an suitable acquisition function satisfies the following two conditions: (a) Try to find the global optimal value; (b) The evaluation cost should not be too large. To find the global

Algorithm 1 BO-based Resource Allocation Algorithm

```
Input:
    $l_t, s_l$                 // Target latency and resource score threshold
    $bootstrap\_set$           // Train sample set
    $\boldsymbol{k}' = (k_1', \ldots, k_N')$   // Throughput optimal parallelism
    $\boldsymbol{k} = (k_1, \ldots, k_N)$      // Initial parallelism
Output:
    $\boldsymbol{k}_{best}$                    // Best parallelism configuration
 1: Model pre-training using samples in $bootstrap\_set$
 2: Obtain the BO's search space $\Omega$.
       $\boldsymbol{x} = (x_1, \ldots, x_N) \in \Omega, x_i \in [k_i', P_{max}]$
 3: while $true$ do
 4:    $l_r \leftarrow Run\_Job(\boldsymbol{k}, T_d)$
 5:    $score \leftarrow Score\_Function(\boldsymbol{k}, \boldsymbol{k}', l_r, l_t)$
 6:    Add $(\boldsymbol{k}, score)$ to the existing set
 7:    Update the surrogate model $\mathcal{M}$
 8:    if $l_r <= l_t$ and $score >= s_l$ then
 9:       $\boldsymbol{k}_{best} \leftarrow \boldsymbol{k}$
10:       break
11:    else
12:       $\boldsymbol{k}_{new} \leftarrow \arg\max_{\boldsymbol{x} \in \Omega \subset \mathbb{R}^N} EI(\boldsymbol{x}, \mathcal{M})$
13:       $\boldsymbol{k} \leftarrow \boldsymbol{k}_{new}$
14:    end if
15: end while
```

optimal value, AuTraScale wants to mimimize the expected deviation from the true maximum. However, its expense is very high when we consider multiple steps ahead [17]. So AuTraScale chooses an alternative, which is to maximize the expected improvement with respect to the best value known. Besides, to adjust the proportion of global search and local optimization, AuTraScale introduces parameter ξ in the expectation of the improvement function [18]. The formula of acquisition function is as follows:

$$EI(x) = \begin{cases} K\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{5}$$

$$K = \mu(x) - f\left(x^+\right) - \xi \tag{6}$$

$$Z = \begin{cases} \frac{K}{\sigma(x)} & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{7}$$

where $\mu(x)$ and $\sigma(x)$ is the GP mean and standard deviation at the sample x. $\Phi(Z)$ and $\phi(Z)$ is the standard normal CDF and PDF of Z respectively. $f\left(x^+\right)$ is the best value known.

Termination condition. The termination condition of AuTraScale's BO algorithm is that the latency requirement is met and the benefit score is greater than the threshold. AuTraScale calculates the benefit score threshold using the over-allocation ratio w of the resource specified by the user. w is defined as follows:

$$\frac{C_{now} - C_{opt}}{C_{opt}} < w \iff \frac{C_{opt}}{C_{now}} > \frac{1}{1+w} \tag{8}$$

where C_{now} is the current parallelism and C_{opt} is the optimal parallelism. $\frac{C_{opt}}{C_{now}}$ is the resource allocation ratio. For a job with N operators, AuTraScale defines the resource allocation ratio as the mean of the corresponding values of all operators, which can be expressed as $\frac{C_{opt}}{C_{now}} = \frac{1}{N} \times \sum_{i=1}^{N} \frac{k_i'}{k_i}$. Therefore, combining with the definition of the scoring function, the condition that comprehensive benefit should meet when Bayesian optimization terminates is as follows:

$$F \geqslant \alpha + (1-\alpha) \times \frac{1}{1+w} \tag{9}$$

F. Transfer learning method at the changed rate

For the same job, AuTraScale needs a new model to describe the relationship between resource allocation and comprehensive benefits when the input data rate changes. However, the cost of training a model from scratch is unacceptable. AuTraScale refers to the idea of transfer learning and makes full use of the existing benefit model to recommend new configurations. This approach can replace an inaccurate new model by providing an allocation closer to the optimal solution when the amount of initial training samples is insufficient.

Algorithm 2 Transfer Learning Method

```
Input:
    ${\mathcal{M}_i}_{i=1}^{c-1}$, $D_c = \{(\boldsymbol{k}_t^c, s_t^c)\}_{t=1}^{T}$, $\boldsymbol{k}'$, $Num, l_t, s_l$
Output:
    $\boldsymbol{k}_{best}$                // Best parallelism configuration
1:  while $true$ do
2:      $(\boldsymbol{k}_t, s_t) \in D_c$
3:      $\mu_{c-1}(\boldsymbol{k}_t) \leftarrow \mathcal{M}_{c-1}.predict(\boldsymbol{k}_t)$
4:      $D_c' \leftarrow \{(\boldsymbol{k}_t, s_t - \mu_{c-1}(\boldsymbol{k}_t))\}_{t=1}^{T}$
5:      $\mathcal{M}_c' \leftarrow Gaussian\_Process\_Regress(D_c')$
6:      $X_c \leftarrow bootstrap\_set(P_{max}, \boldsymbol{k}')$
7:      $D_{c\_predict} \leftarrow D_c$
8:      for all $\boldsymbol{x}_{test} \in X_c$ do
9:          $\mu_{c-1}(\boldsymbol{x}_{test}) \leftarrow \mathcal{M}_{c-1}(\boldsymbol{x}_{test})$
10:         $\mu_c'(\boldsymbol{x}_{test}) \leftarrow \mathcal{M}_c'(\boldsymbol{x}_{test})$
11:         $\mu_c(\boldsymbol{x}_{test}) = \mu_{c-1}(\boldsymbol{x}_{test}) + \mu_c'(\boldsymbol{x}_{test})$
12:         $D_{c\_predict}$. add $(\boldsymbol{x}_{test}, \mu_c(\boldsymbol{x}_{test}))$
13:     end for
14:     $(\boldsymbol{k}_{new}, score) \leftarrow Algorithm1(l_t, s_l, D_{c\_predict})$
15:     $D_c.add(\boldsymbol{k}_{new}, score)$
16:     $num++$
17:     if $num >= NUM$ then
18:         $Algorithm1(l_t, s_l, D_c)$
19:         break
20:     end if
21: end while
```

When the input data rate changes, AuTraScale first optimizes throughput to obtain the base configuration $\boldsymbol{k}'$ and then calls the transfer learning method. Suppose that there are $c-1$ benefit models $\{\mathcal{M}_i\}_{i=1}^{c-1}$, where the corresponding rate of the model $\mathcal{M}_{c-1}$ is the closest to the new rate. We chose the model $\mathcal{M}_{c-1}$ and the real samples from an available set D_c at the current rate to train a residual model $\mathcal{M}_c'$. Then we use the residual model and the model $\mathcal{M}_{c-1}$ to calculate the Gaussian process mean of data x_{test} in the initial sample set, which is called μ_c' and μ_{c-1}. The sum of μ_c' and μ_{c-1} is the objective function estimation of the current data x_{test}. This method saves the consumption of running the initial set samples one by one. The specific steps of the above process can be seen in Algorithm 2.

It is important to note that when there are enough real samples at the new rate, AuTraScale can automatically switch from Algorithm 2 to Algorithm 1. Because when the accuracy of the model trained by the real sample size is high enough, the estimated sample will lose its function and even reduce the accuracy of the model. We recommend that the algorithm is switched when the number of real samples is at least larger than the initial set size. The switching time can be determined by setting parameter Num according to the specific situation.

IV. System Design

The overall design of the AuTraScale system follows the control cycle of the monitor, analyze, plan, and execute (MAPE) [19]. An overview of the system architecture is presented in Fig. 3. Rectangles filled with grids, diagonals, brick lines, and transverse lines represent the above four control modules respectively.

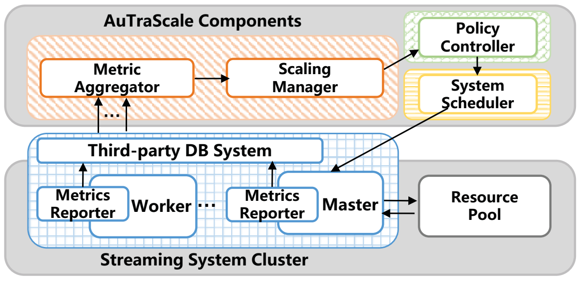


Fig. 3. The architecture overview.

Monitor. The streaming systems usually provide users a metric interface to gather job running information or expose metrics to external systems. To more conveniently analyze the monitored metrics, AuTraScale chooses a third-party database system, such as Prometheus and InfluxDB, to store metrics information uniformly. Besides, to obtain the true processing rate of the operator mentioned in the Section III, it is necessary to modify the source code of the system. The new metrics are also stored in the third-party DB system.

Analyze. The Metric Aggregator selects and integrates metrics information monitored, such as calculating the total processing rate of all instances of each operator, and sends them to the Scaling Manager. The Scaling Manager will judge whether the resource configuration needs to be adjusted and whether there is a model suitable for the current rate in the model library.

Plan. The Policy Controller decides to execute Algorithm 1 or Algorithm 2 according to the feedback from the Scaling Manager. The controller creates or updates the benefit model in the model library, and outputs the next resource allocation scheme. It is worth mentioning that the accuracy of the model will gradually increase as the training data increases during the job runs.

Execute. After receiving the resource configuration recommended by the controller, the System Scheduler stops the current job and store status information as a *savepoint* firstly. Then, it restarts the job that was just interrupted using the new configuration.

In addition, to improve AuTraScale's flexibility and stability, we introduce the following parameters during the AuTraScale component operation.

- *Policy interval.* This parameter determines how often AuTraScale components are called.
- *Policy running time.* When the Policy Controller makes a decision, the job needs a certain amount of time to restart and the QoS is extremely unstable at this time. Therefore, AuTraScale needs the policy running time to obtain a stable metric in the current configuration. In the policy running phase, the decisions of the Metric Aggregator and Scaling Manager will be ignored. It is better to set the *policy running time* to an integer multiple of the *policy interval*.

V. Evaluation

A. Experiment Setup

We evaluate AuTraScale on a physical cluster composed of 3 Dell PowerEdge R730xd (20-core CPUs and 256 GB RAM) and 1 Dell PowerEdge R740xd (32-core CPUs and 256 GB RAM). AuTraScale is implemented in Flink 1.10.0 and Hadoop Yarn 2.7.5. We run Flink applications in the yarn-per-job mode, which launch Flink within YARN only for executing a single job. Hadoop and Flink are located at three R730xd machines. To simulate the real production environment, we also deploy Zookeeper and Kafka on the other R740xd machine in a pseudo-distributed mode to store data. Flink metrics and Kafka metrics are stored in influxDB databases that running on the Hadoop master machine.

For comparison, we also evaluate recently proposed auto-scaling methods DRS [5] and DS2 [14] in the Flink environment. DS2 is a simple and fast method for optimizing throughput with some latency benefits, but its optimization effect is affected by the linear assumption. Comparing with DS2, we want to verify the effectiveness of AuTraScale and whether it can obtain some better solutions. DRS can guarantee end-to-end latency and minimize resource usage based on the queuing theory and greedy algorithm during the auto-scaling process. However, its accuracy is easily affected by interference between tasks. We mainly compare the performance of AuTraScale and DRS in terms of latency guarantee. To avoid the influence of different rate metrics, we use the observed processing rate and true processing rate to run DRS respectively. In this way, the advantages of AuTraScale itself in minimizing resource usage and convergence rate can be highlighted without the effect of new metric.

We use three representative workloads to verify the performance of AuTraScale. WordCount Streaming Job has a simple linear DAG structure, which contains four operator: Source, FlatMap, Count and Sink. Yahoo Streaming Benchmarks [20] is an advertisement event processing case and we use an extended version [21] (The DAG structure is shown in Fig. 4). Nexmark [22] is a benchmarking suite of Apache Beam that contains multiple continuous data stream queries. We use Query5 (sliding window) and Query11 (session window) to evaluate the performance of AuTraScale on the special operators.

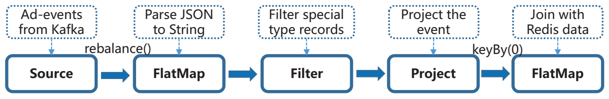


Fig. 4. The topology of Yahoo Streaming Benchmarks.

B. Throughput optimization

To verify the effect of throughput optimization in Algorithm 1, we run four workloads: WordCount, Yahoo, Nexmark-Query5, and Nexmark-Query11. Their input data rates are 350k records/s, 60k records/s, 30k records/s, and 100k records/s, respectively. The initial parallelism of all operators for each workload is 1 and the policy running time is 5 minutes.

When the throughput is lower than the input data rate and running time is greater than 5 minutes, the Scaling Manager will inform the Policy Controller to call Algorithm 1 and return new parallelism. The System Scheduler receives the new configuration and restarts the job. After the policy running time, if the Scaling Manager detects that the current throughput is greater than the input data rate or the current parallelism is the same with the last iteration, the algorithm will terminate.

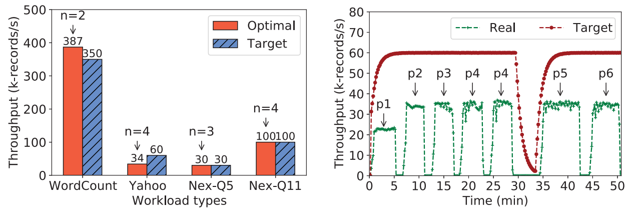


Fig. 5. (a) **Throughput optimization results for different workloads.** *n* represents the number of iterations. Because of the record backlog in Kafka, WordCount's throughput is temporarily higher than the targe. The throughput of Yahoo jobs is limited by Redis's read/write rate and cannot reach the target rate. (b) **Throughput optimization process for Yahoo Streaming job.** In the fourth iteration, the parallelism is p4 (40, 1, 1, 1, 40) again, and the algorithm is terminated. When the parallelism is larger, such as p5 (40, 40, 40, 40, 40) and p6 (50, 50, 50, 50, 50), the throughput does not continue to increase.

When the throughput optimization algorithm is terminated, the operator parallelisms of the four workloads are (3, 4, 12, 10), (40, 1, 1, 1, 40), (1, 18), and (1, 11), respectively. Fig.

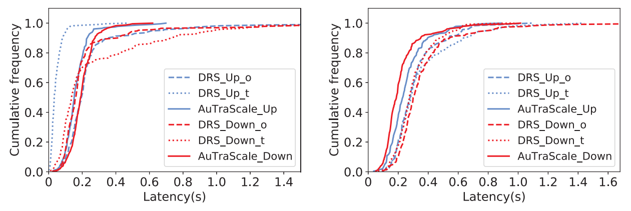
5(a) shows the optimal throughput and the number of iterations for different workloads. From the results, AuTraScale can achieve the optimal throughput with four iterations at most. For WordCount job, the data accumulated in Kafka are also processed along with the newly arrived data under the current optimized configuration, so its throughput will be higher than the input data rate. Due to the limitation of the read/write rate of Redis, the optimal throughput of the Yahoo streaming job cannot reach the input data rate. In this situation, the parallelism at the end of iteration is not always optimal. Therefore, AuTraScale reviews the iterative process and selects the solution with maximum throughput and less resource utilization as the final result. In Fig. 5(b), the parallelism p2 (4, 2, 1, 1, 34) is selected as the final optimal configuration. The experiment after 35th minute in Fig. 5(b) is to verify that higher parallelism does not lead to further throughput optimization due to the external limit.

C. *Elasticity tests at a steady rate*

After determining the optimal configuration of throughput, AuTraScale trains the initial model using the bootstrapping samples and iteratively updates the model as the job runs. When QoS and resource utilization do not meet requirements, AuTraScale starts the scale-up or scale-down operation to adjust the resource configuration. We design the following experiments to show the efficiency of AuTraScale in this process.

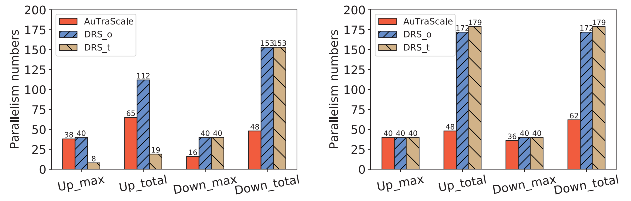
The experiment consists of two jobs: WordCount job (target throughput is 350k records/s and target latency is 180 ms) and Yahoo job (target throughput is 34k records/s and target latency is 300 ms). The initial training set of two jobs contains 10 and 40 samples respectively. AuTraScale algorithm terminates when the latency, throughput and benefit scores meet the requirements concurrently. The benefit score threshold is 0.9 and the policy running time is 10 minutes. The DRS method with true and observed processing rate is used for comparison. It runs until the latency meets the requirements or the total number of new parallelism scheme is over the upper limit of resource.

The results are shown in Table II and Table III. From the results, we can draw the following conclusions. First, as long as the resources are sufficient, AuTraScale can find a parallelism scheme that meets QoS requirements in fewer steps. The more train samples, the fewer iterations. Second, Fig. 7 shows that the optimal parallelism that meets QoS requirements of AuTraScale takes fewer resources comparing with the DRS method in most tasks. AuTraScale can reduce 66.6% and 36.7% resource consumption respectively in the scale-down and scale-up scenarios. Although the DRS method with true processing rate can find solutions that use fewer resources than AuTraScale in the WordCount scale-up experiment, it can not meet the throughput requirements. Third, Fig. 6 shows that less parallelism may bring better latency benefit, and this phenomenon is consistent with Observation 2.2 in Section II. It is worth noting that the configurations obtained by the DRS method sometimes do not meet QoS requirements. It shows that the error of the queueing model is larger in complex resource mapping schemes. In contrast, the Gaussian process model used by AuTraScale has better performance.



(a) WordCount Streaming job (b) Yahoo Streaming job

Fig. 6. **Latency comparison of optimal configuration for different methods in elasticity tests.** *DRS_Up_o* represents the final optimized configuration found by running the DRS method with the observed rate in the scale-up test. *DRS_Down_t* represents the final optimized configuration found by running the DRS method with the true rate in the scale-down test, and so on.



(a) WordCount Streaming job (b) Yahoo Streaming job

Fig. 7. **Parallelism comparison of optimal configuration for different methods in elasticity tests.** *Up_max* represents the max parallelism of optimal configuration in the scale-up test. *Down_total* represents the sum of all operators' parallelism in optimal configuration in the scale-down test.

D. *Transfer efficiency when data rate changes*

In this set of experiments, we compare the efficiency of AuTraScale's transfer learning algorithm and DS2 method when data rate changes by running Query5 and Query11 in Nexmark benchmarking suite. The input data rates of the two queries are set to 30k records/s and 100k records/s respectively. We train the benefit models corresponding to 20k records/s and 80k records/s in advance as the input of the transfer learning algorithm. The target latency of the two queries is set to 500 ms and 150 ms respectively. We run DS2 with offline mode to get the optimal parallelism. Fig. 8(a) shows the parallelism and the number of iterations when these two methods terminate the iteration. Fig. 8(b) plots per-record latency for the terminal configuration of each query with different methods.

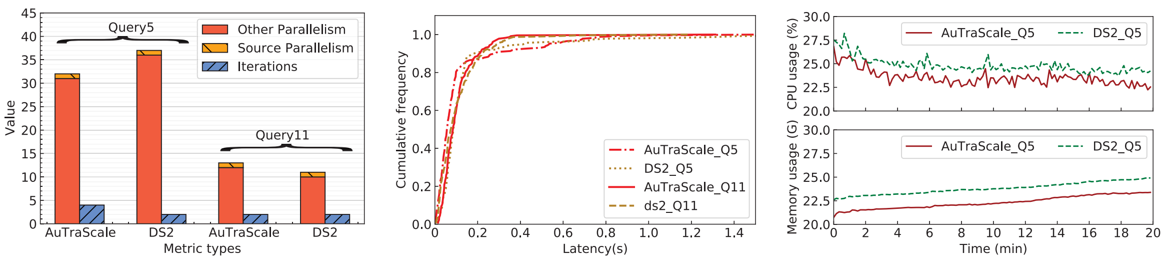
As shown in Fig. 8(a) and Fig. 8(b), for Query11, AuTraScale uses the same number of iterations as DS2 to find a configuration with similar parallelism and similar latency distribution. It indicates that our transfer learning algorithm can achieve similar results with existing methods. For Query5, although AuTraScale needs two iterations more than DS2 to find the configuration to meet QoS requirements, its recommended configuration saves five parallelisms (resource units) than DS2. For both Query11 and Query5, AuTraScale saves an

TABLE II
COMPARISON OF OPTIMAL CONFIGURATION IN THE SCALING TEST FOR WORDCOUNT JOB.

Type	Initial value	Method	Rate type	Optimal value	Throughput (records/s)	Latency (ms)	Score	Iteration	meet QoS
Up	1,3,12,2	DRS	observed	23,33,40,16	381k	246	–	9	no
			true	2,1,8,8	338k	146	–	2	no
		AuTraScale	true	7,8,12,38	405k	172	0.9096	3	yes
Down	36,32,24,20	DRS	observed	40,40,40,33	405k	120	–	1	yes
			true	40,33,40,40	406k	242	–	1	no
		AuTraScale	true	10,16,12,10	355k	160	0.9275	4	yes

TABLE III
COMPARISON OF OPTIMAL CONFIGURATION IN THE SCALING TEST FOR YAHOO STREAMING JOB.

Type	Initial value	Method	Rate type	Optimal value	Throughput (records/s)	Latency (ms)	Score	Iteration	meet QoS
Up	4,2,1,1,4	DRS	observed	40,33,33,33,33	35.3k	329	–	4	no
			true	40,33,33,33,40	35.4k	242	–	2	yes
		AuTraScale	true	4,2,1,1,40	34.7k	223.7	0.994	1	yes
Down	40,40,40,40,40	DRS	observed	40,33,33,33,33	35.1k	276	–	2	yes
			true	40,33,33,33,40	34.5k	247	–	2	yes
		AuTraScale	true	22,2,1,1,36	36.2k	246	0.965	1	yes



(a) Parallelism and iterations.

(b) Latency distribution.

(c) CPU and memory usage.

Fig. 8. The optimal configuration comparison of the AuTraScale transfer learning algorithm and DS2 at a new rate for Nexmark job.

average of 13.5% in parallelism. Specifically, as shown in Fig. 8(c), our method saves 5.2% CPU resources and 6.2% memory resources on average comparing with DS2. In terms of latency, AuTraScale regards latency as one of the optimization goals to ensure that the latency requirement is met, but DS2 does not explicitly optimize the latency. Besides, the per-record latency benefits of AuTraScale are slightly better than that of DS2.

E. The running overhead of the AuTraScale

AuTraScale directly calls the *metric group* interface of Flink to expose the rate information to the outside, so it does not bring additional overhead to the system. Users can get the true rate information by the path *taskmanager_job_task_trueProcessingRate* like accessing other native metrics.

To evaluate the algorithm overhead of AuTraScale, we run two algorithms with a different number of operators to obtain the CPU time consumed by them. The results are listed in Table IV. From the results, the overheads of *Alg1_train* and *Alg2* are linear to the number of operators, but it is not enough to affect the QoS of job. Moreover, the algorithm overhead will be less than 1ms when Algorithm 1 directly calls the model to recommend new parallelisms for different operators, as shown in line *Alg1_use*.

TABLE IV
COMPUTATION OVERHEADS IN SECOND AT THE DIFFERENT NUMBER OF OPERATORS.

Numbers	2	4	6	8	10
Alg1_train	0.042	0.053	0.065	0.076	0.088
Alg1_use	0.0008	0.0006	0.0006	0.0006	0.0006
Alg2	0.070	0.081	0.094	0.104	0.116

VI. RELATED WORK

Streaming system need more flexible auto-scaling solutions to avoid inappropriate resource allocation because of frequently changing and unpredictable workloads. Existing auto-scaling solutions of streaming systems include the following policies or models.

Threshold-based policy. The controller uses metrics about system resources and runtime information to trigger scaling action. Some methods use traditional mertics include CPU utilization [10] [23], memory utilization [23], throughput [8], network [24], congestion status [8], etc. Some other methods define some new metrics, such as the Effective Throughput Percentage(ETP) in Stela [11]. The scaling threshold may be dynamic or contain some buffer conditions of trigger action.

Queuing theory model. Some work [4] [5] use mathematical models related to queuing theory, such as Kingman's formula and Jackson open queueing networks, to model the end-to-end latency of the system. The controller uses these models to predict the latencies of different scaling schemes before action and find the best one to execute. However, the accuracy of this policy is easily affected by interference between multiple tasks.

Rule-based and blacklisting policy. Rule-based Dhalion [7] detects system bottlenecks by analyzing metrics such as system backpressure, then generates diagnosis and select a scaling plan according to the select rule in the resolution stage. The blacklisting means to put a solution that has no actual benefits on the blacklist and not execute it again. However, the backpressure monitoring method cannot give a reduction plan in the case of over-provisioning. IBM Streams [8] that uses the congestion metric has similar drawbacks.

Other policies. Google Dataflow [12] heuristically adjusts the number of workers in a cloud environment based on several signals such as CPU utilization, backlog, and throughput. Based on the data flow model, DS2 [14] determines the appropriate parallelism according to the input data rate and the true processing rate of the operator. A topology-based scaling policy [25] for Apache Storm is proposed to make up for some drawbacks of rebalance command and improve scaling performance. This scheme performs coarse-grained elastic scaling without capturing the difference between operators and has a poor overall resource utilization.

VII. Conclusion

In this paper, we propose AuTraScale, an auto-scaling solution for streaming systems to guarantee QoS and save resource usage. AuTraScale abstracts the relationship between the parallelism and QoS in streaming systems to a Gaussian process model to minimize the impact of resource interference on the prediction accuracy. The Bayesian optimization method is used to iteratively update the model and recommend the optimal parallelism configuration. When the input data rate changes, the transfer learning method can quickly adjust the parallelism as needed with minimal samples. To evaluate the effectiveness of AuTraScale, we conduct several experiments and compare with the state-of-the-art methods. Results show AuTraScale can reduce 66.6% and 36.7% resource consumption respectively in the scale-down and scale-up scenarios while ensuring QoS, and save 13.5% resource on average when the input data rate changes. For future work, we plan to investigate efficient methods to unbind benefit models from input data rates, reduce the training costs and decrease the additional latency overhead in the reconfiguration process.

Acknowledgment

We thank anonymous reviewers for their insightful comments. This work is partially sponsored by the National Natural Science Foundation of China (NSFC) (No.61832006, No.61702329, and No.61972247) and the Research and Development Projects of Applied Technology of Inner Mongolia Autonomous Region of China under Grant No.201802005. The corresponding author is Wenli Zheng.

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. of the SOSP'13*.

[3] "Storm," https://storm.apache.org/.

[4] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. of the ICDCS'15*.

[5] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: Auto-scaling for real-time stream analytics," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3338–3352, 2017.

[6] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016.

[7] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, 2017.

[8] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2013.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska *et al.*, "The design of the borealis stream processing engine." in *Cidr*, vol. 5, no. 2005, 2005, pp. 277–289.

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

[11] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. of the IC2E'16*.

[12] M. Dvorský and E. Anderson, "Comparing cloud dataflow autoscaling to spark and hadoop," https://cloud.google.com/blog/products/gcp/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop, 2016.

[13] D. Sun, H. He, H. Yan, S. Gao, X. Liu, and X. Zheng, "Lr-stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications," *Future Generation Computer Systems*, 2020.

[14] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. of the OSDI'18*.

[15] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proc. of the ICDCS'14*.

[16] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *Proc. of the ICDE'18*.

[17] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[18] D. J. Lizotte, *Practical bayesian optimization*. University of Alberta, 2008.

[19] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[20] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. of the IPDPSW'16*.

[21] "Extended yahoo streaming job," https://github.com/dataArtisans/yahoo-streaming-benchmark.

[22] "Apache beam nexmark benchmark suite," https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[23] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. of the SoCC'15*.

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

[25] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti, "A topology-based scaling mechanism for apache storm," *International Journal of Network Management*, vol. 27, no. 3, p. e1933, 2017.