Research Article

Resource Configuration Tuning for Stream Data Processing Systems via Bayesian Optimization

Shixin Huang,1,2 Chao Chen,1 Gangya Zhu,1,2 Jinhan Xin,1,2 Zheng Wang,1 Kai Hwang,3 and Zhibin Yu1

1Shenzhen Institute of Advanced Technology, CAS, China
2University of Chinese Academy of Sciences, Shenzhen, China
3The Chinese University of Hong Kong, Shenzhen, China

Correspondence should be addressed to Zhibin Yu; zb.yu@siat.ac.cn

Received 4 July 2022; Accepted 9 August 2022; Published 6 October 2022

Copyright ©2022 Shixin Huang et al. Exclusive Licensee Zhejiang Lab, China. Distributed under a Creative Commons Attribution License (CC BY 4.0).

Stream data processing systems are becoming increasingly popular in the big data era. Systems such as Apache Flink typically provide a number (e.g., 30) of configuration parameters to flexibly specify the amount of resources (e.g., CPU cores and memory) allocated for tasks. These parameters significantly affect task performance. However, it is hard to manually tune them for optimal performance for an unknown program running on a given cluster. An automatic as well as fast resource configuration tuning approach is therefore desired. To this end, we propose to leverage Bayesian optimization to automatically tune the resource configurations for stream data processing systems. We first select a machine learning model—Random Forest—to construct accurate performance models for a stream data processing program. We subsequently take the Bayesian optimization (BO) algorithm, along with the performance models, to iteratively search the optimal configurations for a stream data processing program. Experimental results show that our approach improves the 99th-percentile tail latency by a factor of 2.62× on average and up to 5.26× overall. Furthermore, our approach improves throughput by a factor of 1.05× on average and up to 1.21× overall.

1. Introduction

Stream data processing systems, such as Apache Flink [1], Apache Storm [2], Spark Streaming [3], and Apache Heron [4], are being widely deployed in an increasing number of companies (for example, Alibaba and Twitter) as these companies have to deal with continuous stream data and perform real-time data analysis. Flink, one of the latest stream data processing systems, is being widely used for real-time monitoring, real-time warehouse data analysis, and stream-data analysis.

Typically, stream data processing applications repeatedly run on a specific hardware environment for long periods of time (e.g., months or years), with each application being run using different data. Considering these applications’ large-scale deployment and long, even a small performance improvement can bring huge economic benefits (e.g., millions of dollars) for companies. Unlike other big data applications such as batch processing, the higher performance of stream data processing applications implies shorter 99th-percentile latency and higher throughput [5] simultaneously.

Stream data processing systems typically provide a large number (e.g., 30) of resource configuration parameters to specify the amount of resources such as CPU cores and memory used in tasks. These configuration parameters, including those related to I/O behavior and load balancing, significantly affect the performance of a stream data processing application; we call these key-configuration parameters. Tuning these key-configuration parameters can achieve surprisingly high performance.

However, picking key-configuration parameters, as well as finding their optimal values for a stream data processing
application, is very challenging. This is because the optimal configuration parameter values depend on various aspects including the physical cluster resources, the characteristics of the application, and its input data. Furthermore, these aspects are usually intertwined in a complex way, making the configuration tuning extremely challenging.

As such, manually tuning these parameters is extremely time-consuming. For a single unknown application, a performance engineer, who has a deep understanding on the stream data processing system, may take several days or even weeks to find its optimal resource configuration. What is worse, when the number of key-configuration parameters becomes high-dimensional (e.g., greater than 30), using traditional methods to seek the optimal optional parameter values to achieve high performance is NP-hard. Consequently, autotuning the resource configuration parameters for optimal performance is desired.

In the present study, we leverage a machine learning-based methodology [6] to adjust the resource allocation parameters automatically and efficiently for stream data processing applications. It takes the speed of input data and key-configuration parameters as input to employ a highly accurate performance model for a stream data processing workload, which outputs the tail latency or throughput of the application. Along with the performance model, we utilize a Bayesian optimization algorithm (BOA) to search the optimal resource configuration parameter values to achieve high performance.

Without loss of generality, we use Apache Flink programs as stream data processing applications for our experiments. Our methodology is not bound to specific parameters or systems. Instead, it can be applied to other stream processing systems such as Apache Storm or Spark Streaming. To the best of our knowledge, this is the first study to tune the resource configuration parameters for stream data processing frameworks by using Bayesian optimization.

Following are the contributions of this study:

(i) We propose a methodology to autotune resource configuration parameters of stream data processing frameworks

(ii) We propose a Random Forest algorithm to build a highly accurate performance model for a stream data processing program

(iii) We leverage the Bayesian optimization algorithm (BOA) to iteratively search the high-dimensional resource configuration space to achieve optimal performance

We demonstrate that our approach significantly improves the 99th-percentile latency of Flink applications compared with default configurations by a factor of $2.62 \times$ on average and up to $5.26 \times$ overall and improves the throughput by a factor of $1.05 \times$ on average and up to $1.21 \times$ overall.

The remainder of this paper is organized as follows. In Section 2, we describe materials and methods. Section 3 introduces experimental results and discussion. In Section 4, we present our conclusions.

2. Materials and Methods

2.1. Background and Motivation

2.1.1. Flink Framework. Apache Flink is an open-source real-time data processing engine that can process both batch and stream data [7]. It is based on event-driven, efficient checkpointing mechanism, and exactly once semantics. Flink allows users to write their own applications that can be deployed on multiple distributed compute nodes, which makes the speed of processing massive amounts of data much faster than a single computer node. Unlike Apache Spark, Storm, and other frameworks, Flink provides a unified processing architecture for both batch and stream data processing. As a matter of fact, while processing, it treats batch data as a type of special stream data.

Figure 1 shows the architecture of Flink. The Flink runtime consists of two types of daemons: JobManager and TaskManager. JobManager acts as the master node running on a Flink cluster and is responsible for scheduling tasks, coordinating checkpoints, and executing failover strategies, among other tasks. TaskManager is a worker process on each slave node of the cluster and executes tasks while managing the status of each slave node. In addition to the runtime, Flink uses Client to prepare and send a dataflow to the JobManager.

The entire job submission process is as follows: a user submits a job, which is then transformed into a directed acyclic graph (DAG) [8] of a task for execution. The nodes of a DAG can be classified into three categories: source nodes, sink nodes, and operators. The source nodes read data from the message queue. The operators define how the intermediate data are processed. And the sink nodes output the final processing results. The interaction between the JobManager and the TaskManager and the JobManager and the Client is based on the Akka toolkit.

The submission of a Flink job also involves the creation of the Actor System, the startup of JobManager, and the startup and registration of TaskManager. JobManager and TaskManager communicate with each other via the Actor System to obtain the status of executed tasks and then send it to a Client via the Actor System. During job execution, the JobManager may trigger checkpoint operations. After each TaskManager receives trigger-checkpoint instructions, they finish checkpoint operations. All coordination processes about checkpoint are completed by the JobManager. When a job is completed, Flink feeds the information of the job’s execution back to the Client and releases the resources occupied by the TaskManager for the next submission.

Flink applications can be deployed on mainstream resource managers (e.g., Mesos [9], YARN [10], and Kubernetes [11]). Flink provides hundreds of configuration parameters (e.g., >300) that specify different aspects of one Flink job, including JobManager, TaskManager, network communication, and shuffle behavior. Some of these configuration parameters do not affect the performance of Flink applications. For instance, jobmanager.archive.fs.dir defines the file directory where the JobManager stores the archives of jobs that have been completed, and jobmanager.rpc.port
Bayesian optimization consists of two key parts. The first part is a prior model, also called the probabilistic surrogate model; it consists of a distribution that captures our beliefs about the behavior of a black-box function. The other part is an acquisition function; examples include Thompson sampling (TS), probability of improvement (POI), expected improvement (EI), and upper-confidence-bound (UCB). The advantage of acquisition functions is that they are easy to evaluate compared with the unknown objective function $f$. In most cases, the acquisition function is much easier to optimize compared with the original black-box function. Algorithm 1 is the pseudocode of a Bayesian optimization framework, and Figure 2 illustrates the iterative process of Bayesian optimization.

The main idea of Bayesian optimization is to build a model that can be easily updated and queried to find a global maximizer in a short time. To explain Bayesian optimization, we take a model parameterized by $w$ as an example and let $D$ represent the available data. We simply consider the nonparametric case in this example. Because $w$ is unobserved, we treat it as a random variable with a prior distribution $p(w)$, which captures our beliefs about the unknown objective function. When we are given data $D$ and a likelihood $p(D|w)$, inferring a posterior distribution $p(w|D)$ is easy by using Bayes’ rule:

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)},$$

where $p(D)$ represents marginal likelihood. After observing data $D$, we update our beliefs about $w$. The Bayesian optimization algorithm uses Bayes’ rule, which makes the posterior computing analytically and quickly.

2.1.3. Motivation. Flink has more than 300 configuration parameters, and the key-configuration parameters need to be
tuned. In this section, we attempt to answer two questions: (1) whether the performance of Flink (such as throughput and latency) is affected by input data speed and (2) whether some approaches on automatic parameter tuning for big data processing systems [13–16] (such as Hadoop and Spark) are also applicable to Flink.

2.1.4. Input Data Speed Sensitivity. In theory, when the input data speed increases, the streaming processing system has to adapt itself to handle the increased input data speed and process tuples. When the input data speed is beyond the ability of the system to handle, backpressure arises in the system. To find out whether the input data speed influences throughput and latency of Flink applications, we first select three programs in HiBench [17] as follows: Flink WordCount (Flink-WC), Flink Repartition (Flink-RP), and Flink Identity (Flink-ID). Subsequently, we run these three programs with three different input data speeds—8 K records/s, 80 K records/s, and 160 K records/s—with the default resource configuration on the same cluster. Our goal is to observe the fluctuations of throughput and latency.

Figure 3 shows the fluctuations of throughput and latency. We can clearly find that the input data speed significantly influences both throughput and latency of Flink applications. In addition, throughput and latency are not closely correlated. However, we note that their values increase with the speed of the input data. Here is what explains this phenomenon.

On the one hand, when the input data speed increases, the streaming system has to scale out to process more data, which causes the system to receive data at a much higher rate than it can process it. As a result, backpressure builds in the
system. From the moment when the backpressure mechanism starts, the latency of all queue messages increases.

On the other hand, fluctuations of throughput and latency exist because of the nature of In-Memory Computing (IMC) frameworks. As an IMC framework, Flink leverages memory to keep its state. As Figure 4 shows, stateful Flink applications are optimized for local state accesses. Task state is always maintained in memory unless there is not enough memory. In that case, the task state is maintained in access-efficient on-disk data structures. Consequently, computations are performed using local states, which results in low processing latency. Besides, since it periodically checkpoints the local state to the storage, the Flink guarantees exactly once state consistency in case of failures.

2.1.5. Limitations of Prior Works. Popular big data computing frameworks such as Spark and Flink also contain numerous configuration parameters controlling memory, I/O, computing cores, and so on. Herodotou et al. [18] classify common big data parameter autotuning methods into six categories. Rule-based methods assist developers tune some parameters quickly. It relies on expert experience and manual tuning instructions, but a deep understanding of the internal mechanism of the system is required. White-box approaches are broadly used in constructing cost model and analytical model. Large amounts of history information and system logs are needed to be analyzed to build an accurate model. This type of complex analytical approach is not suitable for Flink applications. Simulation-based approaches, meanwhile, cannot tune the resource configuration parameters automatically, and thus, they are not very efficient for searching the optimal configuration for a specific streaming workload.

Though experiment-driven approaches can find optimal settings on real systems, they are time-consuming because repeated experiments on physical machines are necessary. Also, the high-dimensional configuration space makes it impossible to find an optimal configuration to achieve our desired performance. Adaptive approaches can tune resource configurations when an workload is running, but improper settings of some configuration parameters may cause certain system issues. Machine learning-based methods typically abstract a complex system into an invisible black box to tune system parameters. One advantage is they do not require an in-depth understanding of the system mechanism and hardware. The other advantage is that they have the ability to capture complex system dynamics. However, machine learning-based approaches have yet not been applied in stream data processing systems. We therefore choose machine learning approaches to build performance models for our Flink workloads.

2.2. Related Work. For batch processing systems, Herodotou et al. [19–21] first proposed using analytical models to predict the performance of the MapReduce framework. Gencer et al. [22] developed the response surface (RS) methodology to build a performance model, after which the model was used in a simulator for MapReduce jobs. Bei et al. [23] suggested using Random Forest to autotune Hadoop’s configurations. Their results showed that RFHOC’s performance increases when the size of the input data set increases.

In-memory cluster (IMC) computing frameworks such as Spark can leverage memory resource to store temporary...
data and thus perform better than traditional on-disk cluster computing [24, 25]. Yu et al. [26] proposed a DAC, a datasize-aware autotuning methodology to efficiently find the optimal configuration in high-dimensional configuration space for a given IMC program, and compared with default configurations; the authors achieved improved performance of Spark programs by a factor of 30.4× on average and up to 89× overall.

For the stream processing system Apache Storm, Li et al. [27] aimed not only to leverage support vector regression (SVR) to construct the workload performance model but also provided an efficient scheduling solution for Storm. Trotter et al. [28] suggest accelerating the exploration of the configuration space in Storm by employing searching algorithms such as genetic algorithm (GA) or Bayesian optimization (BOA).

For Spark Streaming, Lin et al. [29] leveraged an executable and reliable model named SSP (stands for Spark Streamization algorithm (BOA).) Their approach significantly improves the performance of Flink programs. Moreover, our approach treats the Flink system as a black box, which means a user is not required to have a deep understanding of Flink. In addition, we take into account the influence of input data speed, and our approach is adaptive to system changes.

2.3. Machine Learning Approach Based on Bayesian Optimization. In this section, we propose a machine learning approach based on Bayesian optimization to optimize the performance for a given Flink application on a given hard cluster by autotuning its key-configuration parameters. Figure 5 depicts an workflow of our methodology. It is composed of five modules: determining, generating, collecting, modeling, and searching.

In the determining module, we pick key-configuration parameters out of more than 300 configuration parameters in Flink using theories and experiments in the (1) step.

In the (2) step, the generating component automatically generates random configuration parameter values that are employed to learn the potential relationship between configuration parameters and the performance (e.g., tail latency or throughput) of a given Flink application. Note that in the (2) step, we classify these configuration parameters into two categories: FailConf and PassConf. FailConf represents the configuration parameters with which fail to start a Flink Cluster or submit a job. For example, on the Flink official website, it is not recommended to explicitly configure both total process memory and total Flink memory. It may lead to deployment failures due to potential memory configuration conflicts. Unfortunately, we do not know whether the configuration parameters are proper unless we use them to submit a Flink job.

On the contrary, PassConf represents the configuration parameters with which we can successfully submit a Flink job. By performing experiments with such parameters, we can easily find the PassConf. Later, in the (3) and (5) steps, the collecting component collects the performance (such as latency and throughput) of a Flink program with a number of different parameter values in PassConf.

In the (5) step, the modeling module employs an appropriate machine learning algorithm to construct a performance model and tunes the configuration parameters in PassConf.
model that inputs the key-configuration parameters and the speed of input data as inputs and outputs the performance for a given Flink program. Last but not the least, the searching module is employed to automatically search the optimal configurations in the \( \mathcal{O} \) step.

In summary, the searching component depends on the accuracy of the performance models built by the modeling component; in turn, the modeling component depends on the training data of the collecting component, and the generating component needs the key-configuration parameters from the determining component to generate proper configuration parameter values. The five components are closely connected for optimal configuration parameter tuning.

### 2.3.1. Determining Key Configurations

Flink provides more than 300 configuration parameters to tune, and they are stored in `flink-conf.yaml`, which is a collection of key-value pairs with format `key: value`. The configuration and its corresponding value is parsed from this file when an Flink process is started. Therefore, changing a configuration file requires restarting Flink processes. These configurations are mainly classified as follows: basic setup, common setup options, security, resource orchestration frameworks, state backends, metrics, history server, and so on. The basic setup includes the ones needed for a basic distributed Flink setup, which includes `hostnames/ports`, `memory sizes`, `parallelism`, `checkpointing`, and `web UI`, among others. The basic setup is necessary and significant for starting a Flink cluster.

First of all, we can exclude some configuration parameters that do not affect performance, such as `hostnames/ports`, `YARN`, `SSL`, and `web UI`. After this, we are left with approximately 60 configuration parameters. Then, we use the single variable principle to perform experiments: when we change a parameter’s value, we set the rest of the parameters as default values. At the same time, we observe the variation of performance and the corresponding resource configurations such as computing resources (the number of cores required for each task), memory resources (memory size and buffer size), network bandwidth utilization, and disk resource utilization. If the performance varies significantly when the parameter’s value changes, we call these parameters key-configuration parameters. Otherwise, we exclude this configuration parameter. Through the experiments, we end up picking 30 key-configuration parameters, as shown in Table 1. Most of these configuration parameters are related to resources such as memory, and thus, we also call them resource configuration parameters.

### 2.3.2. Generating Resource Configurations

The aim of the generating component is to randomly generate values for resource configuration parameters within their value ranges, and these configurations are written into the configuration file `flink-conf.yaml`. Here we use the following vector to represent a configuration set:

\[
\text{conf}_i = \{c_{i1}, c_{i2}, \cdots, c_{in}\},
\]

with `conf`, the \(i^{th}\) configuration set and \(c_{ij}\) the value of the \(j^{th}\) configuration parameter in the \(i^{th}\) configuration set. \(n\) is 30 which corresponds to the 30 key-configuration parameters of Flink, and \(m\) is the number of configurations. It is worth mentioning that as long as the two values of at least one parameter of two configuration sets are different, we regard them as two different configuration sets.

After generating configurations, we classify these configurations into two categories: `FailConf` and `PassConf`. Note that we only use configurations in `PassConf` to collect performance data, because the `FailConf` cannot successfully submit a Flink job or start a Flink cluster. Unfortunately, the
Table 1: Description of the 30 Flink configuration parameters, extracted from [39].

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th>Description</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallelism.default</td>
<td>the default parallelism used when no parallelism is specified anywhere</td>
<td>1-200</td>
<td>1</td>
</tr>
<tr>
<td>taskmanager.numberOfTaskSlots</td>
<td>the number of slots that a TaskManager offers. It is denoted as “#slot” by some parameters as the default value, such as taskmanager.network.netty.arenas.</td>
<td>1-40</td>
<td>1</td>
</tr>
<tr>
<td>execution.buffer-timeout</td>
<td>the maximum time frequency (milliseconds) for the flushing of the output buffers, in ms. A positive value means triggering flushing periodically by that interval, 0 ms means triggering flushing after every record, and -1 ms triggers flushing only when the output buffer is full.</td>
<td>-1, 0, 1-500</td>
<td>100</td>
</tr>
<tr>
<td>execution.checkpointing.interval</td>
<td>the interval in which checkpoints are periodically scheduled, in ms.</td>
<td>10-600000</td>
<td>(none)</td>
</tr>
<tr>
<td>taskmanager.network.numberOfBuffers</td>
<td>the number of buffers available to the network stack</td>
<td>28-44800</td>
<td>2048</td>
</tr>
<tr>
<td>jobmanager.memory.flink.size</td>
<td>total Flink Memory size for the JobManager, in MB</td>
<td>1024-40960</td>
<td>(none)</td>
</tr>
<tr>
<td>jobmanager.memory.jvm-metaspaces.size</td>
<td>JVM Metaspace size for the JobManager, in MB</td>
<td>64-4096</td>
<td>256</td>
</tr>
<tr>
<td>jobmanager.memory.jvm-overhead.max</td>
<td>Max JVM Overhead size for the JobManager, in MB</td>
<td>512-4096</td>
<td>1024</td>
</tr>
<tr>
<td>jobmanager.memory.jvm-overhead.min</td>
<td>Min JVM Overhead size for the JobManager, in MB</td>
<td>64-512</td>
<td>192</td>
</tr>
<tr>
<td>taskmanager.memory.flink.size</td>
<td>total Flink Memory size for the TaskExecutors, in MB</td>
<td>1024-40960</td>
<td>(none)</td>
</tr>
<tr>
<td>taskmanager.memory.jvm-metaspaces.size</td>
<td>JVM Metaspace size for the TaskExecutors, in MB</td>
<td>64-4096</td>
<td>256</td>
</tr>
<tr>
<td>taskmanager.memory.jvm-overhead.max</td>
<td>Max JVM Overhead size for the TaskExecutors, in MB</td>
<td>512-4096</td>
<td>1024</td>
</tr>
<tr>
<td>taskmanager.memory.jvm-overhead.min</td>
<td>Min JVM Overhead size for the TaskExecutors, in MB</td>
<td>64-512</td>
<td>192</td>
</tr>
<tr>
<td>blob.fetch.backlog</td>
<td>the config parameter defining the backlog of BLOB fetches on the JobManager.</td>
<td>200-5000</td>
<td>1000</td>
</tr>
<tr>
<td>blob.fetch.num-concurrent</td>
<td>the config parameter defining the maximum number of concurrent BLOB fetches that the JobManager serves</td>
<td>10-200</td>
<td>50</td>
</tr>
<tr>
<td>taskmanager.memory.segment-size</td>
<td>size of memory buffers used by the network stack and the memory manager, in KB</td>
<td>4-1024</td>
<td>32</td>
</tr>
<tr>
<td>taskmanager.network.blocking-shuffle.compression.enabled</td>
<td>Boolean flag indicating whether the shuffle data will be compressed for blocking shuffle mode</td>
<td>True, false</td>
<td>False</td>
</tr>
<tr>
<td>taskmanager.network.blocking-shuffle.type</td>
<td>the blocking shuffle type, either &quot;mmap&quot; or &quot;file&quot;</td>
<td>File, mmap</td>
<td>File</td>
</tr>
<tr>
<td>taskmanager.network.netty.client.numThreads</td>
<td>the number of Netty client threads</td>
<td>1-40</td>
<td>#slot</td>
</tr>
<tr>
<td>taskmanager.network.netty.num-arenas</td>
<td>the number of Netty arenas</td>
<td>1-40</td>
<td>#slot</td>
</tr>
<tr>
<td>taskmanager.network.netty.sendReceiveBufferSize</td>
<td>the Netty send and receive buffer size, in MB</td>
<td>4-40960</td>
<td>4</td>
</tr>
<tr>
<td>taskmanager.network.netty.server.numThreads</td>
<td>the number of Netty server threads</td>
<td>1-40</td>
<td>#slot</td>
</tr>
<tr>
<td>taskmanager.network.netty.transport</td>
<td>the Netty transport type, either &quot;nio&quot; or &quot;epoll.&quot; The &quot;auto&quot; means selecting the property mode automatically based on the platform.</td>
<td>nio, epoll</td>
<td>Auto</td>
</tr>
<tr>
<td>taskmanager.network.request-backoff.initial</td>
<td>minimum backoff in milliseconds for partition requests of input channels</td>
<td>10-500</td>
<td>100</td>
</tr>
<tr>
<td>taskmanager.network.request-backoff.max</td>
<td>maximum backoff in milliseconds for partition requests of input channels</td>
<td>1000-200000</td>
<td>10000</td>
</tr>
<tr>
<td>akka.framesize</td>
<td>maximum size of messages which are sent between the JobManager and the TaskManagers, in MB</td>
<td>1-2048</td>
<td>10</td>
</tr>
<tr>
<td>akka.throughput</td>
<td>number of messages that are processed in a batch before returning the thread to the pool</td>
<td>1-500</td>
<td>15</td>
</tr>
<tr>
<td>akka.transport.throttle</td>
<td>threshold for the transport failure detector</td>
<td>100-1000</td>
<td>300</td>
</tr>
<tr>
<td>fs.output.always-create-directory</td>
<td>file writers running with a parallelism larger than one create a directory for the output file path and put the different result files (one per parallel writer task) into that directory.</td>
<td>False, true</td>
<td>False</td>
</tr>
<tr>
<td>fs.overwrite-files</td>
<td>specifies whether file output writers should overwrite existing files by default</td>
<td>False, true</td>
<td>False</td>
</tr>
</tbody>
</table>

emergence of *FailConf* is inevitable because there may be conflicts between different configuration parameters. By submitting a job using generated configurations, we can easily find out whether they belong to *PassConf*.

2.3.3. Collecting Performance Data. For each set of configurations in *PassConf*, we run a Flink application in a given cluster and then use HiBench to collect its throughput and 99th-percentile tail latency. Finally, we define the performance of a Flink application as follows:

$$\text{perf}_i = \lambda_{i=1,2,\cdots,m},$$

where $\text{perf}_i$ is the performance of the $i$th trial of the application, throughput, the throughput (msg/s) of the $i$th execution, and latency$_i$ the 99th-percentile tail latency (ms) of the $i$th execution. $m$ denotes the number of executions. Higher $\text{perf}_i$ is better, and this definition can ensure high
throughput and low latency at the same time, which is good for both service providers and end users.

Subsequently, we use these data as training data to build proper performance models. \( m \) must be large enough to ensure the accuracy of a performance model (e.g., model error rate < 10\%), and different Flink programs may require different values of \( m \) for obtaining good accuracy of the performance model. \( m \) needs a proper value because a larger \( m \) causes a longer runtime of programs and collect correspond performance, and a smaller \( m \) may decrease the accuracy of a performance model. Consequently, we leverage experiments to maintain a favourable balance between the cost of time and the accuracy of the performance model in our environment.

### 2.3.4. Building Performance Models

We consider establishing the following simplified performance model for a given Flink program on a given cluster:

\[
\text{perf} = f(c_1, c_2, \ldots, c_i, \ldots, c_{30}, \text{speed}),
\]

where \( \text{perf} \) the performance, \( c_i \) the value of the \( i \)-th configuration parameter, and speed the input data speed (byte/s). In order to effectively tune configuration parameters for a given Flink application, we use machine learning (ML) approaches to construct an accurate performance model with high-dimensional parameter space. ML models provide two benefits: (1) they regard the intricate system as a black box; in other words, there is no need to comprehend the internals of the system; (2) they can improve accuracy with more training data. However, we face one challenge using ML: the ML model selection issue.

There exists many ML algorithms for us to choose from. Some well-known ones are (1) Random Forest (RF), which uses the average of the multiple decision trees’ prediction results as the final prediction result; (2) Gradient Boosted Regression Tree (GBRT), which is an iterative decision tree algorithm composed of multiple decision trees; (3) Artificial Neural Network (ANN), which forms a collection of connected nodes that can transmit signals to each other, leading to some prediction; and (4) Support Vector Machine (SVM), which is aimed at finding the separated hyperplane that can correctly partition the training data set and has the maximum geometric spacing.

Which one is the most suitable for our performance models? In theory, it is particularly challenging to obtain a correct answer to this question. Our goal is to build a sufficiently accurate model that uses as little training data as possible. We implement four different ML algorithms—SVM, ANN, RL, and GBRT—on top of the sklearn Python library [33] and adopt default hyperparameter values. We employ the four machine learning models on the same training data set and evaluate their fitting effect using the mean absolute percentage error (MAPE). MAPE is a mathematical method that widely used to evaluate the fitting quality of regression models. It is defined as follows:

\[
\text{MAPE} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{\text{pre}_i - \text{act}_i}{\text{act}_i} \right| \times 100\%,
\]

where \( \text{pre}_i \) is the performance predicted by the models for a given Flink program, \( \text{act}_i \) is the actual performance of that program, and \( n \) is the total number of the testing data. In our experiments, one-fifth of the dataset was used as a testing data and the other four-fifths as training data. Note that there exists no overlap between the testing data and the training data.

Figure 6 shows the MAPE of the models for the program `WordCount` built by the four algorithms. The horizontal axis is the total number of dataset (where the training data accounts for 80\% and the testing data accounts for 20\%). As can be seen, the MAPE decreases when the amount of training data increases for all four algorithms. However, the MAPE of RF is the lowest among the experimented algorithms for all the data sets, as shown in Figure 6. We thus choose RF as the performance model.

### 2.3.5. Searching Optimal Configurations

To search the optimal configuration parameters for a given Flink program, a well-performing search algorithm is urgently needed. We require a robust and reliable searching algorithm. Many searching algorithms exists that can efficiently search a particular parameter space, such as grid search [34], random search [35], tabu search [36], genetic algorithm [37], and...
Bayesian optimization. Grid search is highly time-consuming, especially when we search in a high-dimensional space. Random search tends to easily get stuck in local optimality. Tabu search is typically affected by local convergence rate and its initial solution space.

The genetic algorithm is an evolutionary and heuristic algorithm that seeks the optimal solution by simulating the selection and heredity mechanism in nature. However, there are some parameters in a genetic algorithm that dramatically affect the effectiveness of solutions, such as crossover rate and mutation rate. At present, it is very challenging to pick appropriate values for these parameters that heavily rely on parameter tuning experience. However, from results in Section 3.2.2, we find that the genetic algorithm needs more training time to obtain more accurate solutions, and thus, its iteration speed is relatively slow. The iteration efficiency of Bayesian optimization, however, is higher.

As a result, we choose Bayesian optimization for our experiment and compare it against the genetic algorithm on the time used to find the optimal configuration and the final speedups. Note that we utilize Matern 5/2 kernel for Bayesian optimization because it is favored on practical functions [12]. Detailed results are presented in Section 3.2.2.

3. Results and Discussion

3.1. Experimental Setup

3.1.1. Cluster Platform. Our experimental hardware environment is made up of 8 Linux compute nodes, and one of them acts as the master node and the others as the slave nodes. Each compute node is mounted with an Intel(R) Xeon(R) CPU Silver-4114 2.20 GHz 10-core processor and 64 GB DRAM. There are in total 80 cores and 512 GB of memory in our hardware cluster. The operating system used in each node is Ubuntu Server 18.04. We choose Flink 1.11 for this experiment. We use a Kafka cluster to generate data streams. Figure 7 shows the topology of our Kafka and Flink cluster. The cluster consists of 8 servers, where one is used as the Flink master and the others as Flink slaves. Each server runs Kafka and Flink daemons for data transmission and processing. Once we start Kafka, one server will be selected as the leader while the others as followers. The Kafka daemon reads data from the data generator continuously. The Flink daemon processes the stream from Kafka and then outputs the results to Kafka. The latency of each record is computed as the difference between out timestamp and in timestamp.

3.1.2. Representative Programs. We choose all Flink programs from HiBench, as listed in Table 2. Streaming benchmarks are widely used to evaluate streaming processing systems, such as Spark Streaming, Flink, Storm, and Gearpump. The four programs are as follows: WordCount, FixWindow, Repartition, and Identity. WordCount mainly evaluates the performance of the stateful operator and the cost of Checkpoint/Acker during stream processing. FixWindow involves a window operation and thus evaluates the efficiency of window operation. Repartition tests the
3.1.3. Configuration Parameters. Based on expert experience and additional experiments, we select 30 configuration parameters that affect the performance of Flink system critically, including network stack, I/O, parallelism, and execution behavior. Table 1 lists 30 key-configuration parameters in detail. The third column of Table 1 demonstrates default configurations that Apache Flink provides for developers [39]. The second column depicts the value range of each configuration.

3.2. Results and Analysis. In this section, we evaluate the impact of execution number and iteration number for Bayesian optimization. Then we present the performance comparison results and analysis among various methods.

3.2.1. Impact of Execution Number. As described in Section 2.3.3, it is extremely important to determine a suitable execution number \( m \) that affects both model accuracy and time cost. A larger \( m \) increases the model accuracy at the cost of increasing time for collecting training data and vice versa. To minimize the time cost and maximize model accuracy simultaneously, we carefully determine the trade-off \( m \) using experiments.

We first build a performance model for a Flink program using 100 different configurations and increase the training dataset by 100 configurations each time. Figure 9 illustrates the relationship between model accuracy and the number of resource configurations in the dataset for the experimented Flink programs. As mentioned earlier, we use mean absolute percentage error (MAPE) to represent the model accuracy, which is a common metric to evaluate the quality of regression models. A MAPE value of 0% indicates a perfect model (i.e., no mean errors), and as MAPE value increases, the modeling quality deteriorates. Note that when the actual value is 0, there is a problem of dividing the denominator by 0, and thus, this formula is no longer available.

For convenience, we only show the maximum (Max), mean (Mean), and minimum (Min) MAPE of the performance models for all the Flink programs in Figure 9. The overall trend is that the MAPE decreases when the number of resource configurations in the training dataset increases. When \( m \) reaches 1000, the whole curve becomes flat and shows signs of convergence, and thus, we set \( m \) to 1000 to build the model of each Flink program.

3.2.2. Comparison between Bayesian Optimization and Genetic Algorithm. To choose a suitable search algorithm, we compare the Bayesian optimization algorithm (BOA) with the genetic algorithm (GA). Figure 10 shows that Bayesian optimization has a faster convergence speed, which means it requires lesser iteration time. Besides, we observe that the average iteration time of GA is 1.4× that of BOA. Nonetheless, the final optimal search solution of Bayesian optimization is almost the same as that from GA.

Consequently, we choose Bayesian optimization as our search algorithm by iteratively searching the extremely complex resource configuration space to achieve the optimal performance for given Flink programs.

3.2.3. The Impact of Different Acquisition Functions on Bayesian Optimization. During Bayesian optimization, we use acquisition functions to select a sequence of query points. Besides, acquisition functions are helpful for satisfying our requirement to trade off between exploration and exploitation. There are three widely used acquisition functions: probability of improvement (POI), expected improvement (EI), and upper confidence bound (UCB). We divide these three acquisition functions into two categories [12]:

1. **Improvement-Based Policies.** POI favors points that are most likely to improve the current target, while EI prefers to choose these points that can maximize...
Figure 9: The MAPE of performance model as a function of the number of dataset (m executions).

Figure 10: Comparison between BOA and GA for Flink programs. The X-axis represents the time BOA and GA iteration cost in seconds, and the Y-axis represents the 99th-percentile latency. Note that the red vertical line denotes the time taken for BOA to converge, while the blue vertical line is the time taken for GA to converge.
Figure 11: The impact of different acquisition functions for Bayesian optimization. Note that the X-axis represents the iteration number, while the Y-axis represents the 99th-percentile latency. Note that the black vertical line represents the time taken for acquisition function EI to converge.

Figure 12: Performance comparisons for all programs—WordCount (WC), Identity (ID), Repartition (RP), and FixWindow (FW)—with default configurations (def), Bayesian optimization algorithm (BOA), and modified Hill Climbing (mHC) methods. The T1, T2, T3, and T4 of each program correspond to the input data speed from left to right in the third column of Table 3. (a) displays the speedup ratios of throughput to 99th-percentile tail latency using BOA and mHC with respect to def method. (b, c) show 99th-percentile tail latency and throughput of each method, respectively.
the expected improvement which includes the amount of improvement over the current best

(2) Optimistic Policies. When we aim to find a minimum over a black-box function, UCB chooses these points whose certainty region has the smallest lower bound. The key idea of this strategy is to always stay optimistic when facing uncertainty. And UCB has been shown to be an efficient approach to negotiate exploration and exploitation.

In our experiment, we first consider three widely used acquisition functions: POI, EI, and UCB. We also perform an experiment to choose an acquisition function that can achieve better iteration results in Bayesian optimization. Figure 11 shows how different acquisition functions impact the iteration of Bayesian optimization. The iteration result of EI is better compared with POI and UCB. Therefore, we choose EI as the acquisition function for our experiment.

3.2.4. Time Overhead. The total time overhead of our method mainly consists three parts: (1) data collection, (2) model building, and (3) searching the best configuration. It should be noted that the time used for data collection dominates the total time overhead. Another problem is that there always exists noise in the real hardware cluster. To reduce the influence of measurement noise, we repeat the experiment for three times to obtain the average value.

For the stream application, we tested in HiBench, we set a run time of 5 min for one trial. And for one given workload, the time cost for data collection is approximately between 3 and 4 days. In contrast, it usually takes less than 3 s to build an accurate model and approximately a few minutes (e.g., 1 or 2 min) to finish searching the best configuration.

3.2.5. Speedup. To clearly show the optimization results, we use the speedup metric that can be defined as follows:

\[
\text{speedup} = \frac{\text{perf}}{\text{perf}_0}
\]

where \(\text{perf}\) is the performance of an Flink application with configurations produced using our proposed approach Bayesian optimization algorithm (BOA), and \(\text{perf}_0\) is the performance obtained using the default configuration. Besides, we also implement modified Hill Climbing (mHC) [5], the state-of-the-art approach to tuning configuration of streaming processing systems, for performance comparison.

For stream processing systems, both throughput and tail latency are crucial performance metrics. However, it is difficult to obtain a clear relationship between them. In many cases, high throughput does not necessarily produce low latency and vice versa. As a consequence, we consider 99th-percentile tail latency, throughput, and the ratio of the throughput to 99th-percentile tail latency defined in Equation (4) (Section 2.3.3) as the optimization metrics in our study.

Figure 12(a) depicts the speedups of 16 program-input pairs with BOA and mHC compared with the default configurations. Regarding the ratio of throughput and latency (the higher the better) as the optimization metric, compared to the default configurations, BOA improves the 16 programs-input pairs by a factor of 2.73× on average and up to 5.46× overall, whereas mHC improves by a factor of 1.95× on average and up to 3.58× overall. In Figure 12(b), we consider 99th-percentile tail latency as the optimization metric (the lower the better). As can be seen, compared with the default configurations, BOA improves the 99th-percentile tail latency by a factor of 2.62× on average and up to 5.26× overall, while mHC improves by a factor of 1.93× on average and up to 3.51× overall.

In Figure 12(c), we take throughput as the optimization metric (the higher the better). We observe that when it comes to throughput, all methods exhibit similar performance: BOA improves the throughput by a factor of 1.05× on average and up to 1.21× overall, whereas mHC improves by a factor of 1.03× on average and up to 1.09× overall.

3.2.6. Future Work. The proposed methodology can adopt to different workloads (for example, time-varying workload). Furthermore, as it is based on a black-box model, specific characteristics of the application do not affect the universality of the method.

In fact, Flink is just one example of our methodology optimized for a stream processing engine. Flink essentially performs stateful stream processing, storing state (intermediate results) either in the memory or in a stable storage file system (e.g., HDFS). It also supports three different time semantics and provides an incremental checkpoint mechanism. Due to the above internal mechanism, Flink can simultaneously ensure high throughput and low tail latency when processing data streams.

However, the proposed method is a parameter-tuning tool independent of the Flink system. It interacts with the Flink system through several APIs to start Flink, run a given workload, and collect performance data. And later, it constructs an accurate machine learning model and leverages Bayesian optimization to achieve automatic parameter tuning. Because of this, it can also be integrated into other stream processing systems (e.g., Spark Streaming and Storm). In future studies, we plan to add more evaluations using various applications and benchmark suites (e.g., TPC-DS and TPC-H).

4. Conclusion

We proposed a machine learning approach based on Bayesian optimization to autotune the key-configuration parameters of stream data processing systems. The speed of input data and 30 key-configuration parameters were needed to build the performance model. We evaluated the proposed method on a cluster with eight servers using four benchmarks, with each benchmark having four different input data speeds. Experiment results revealed that our approach significantly improved the 99th-percentile tail latency of the four Flink programs by a factor of 2.62× on average and up to 5.26× overall and improved throughput by a factor of 1.05× on average and up to 1.21× overall.
Data Availability

Data are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Authors’ Contributions

S. Huang and C. Chen designed the experiments and wrote the original draft. G. Zhu, J. Xin, and Z. Wang conducted the experiments. K. Hwang reviewed the manuscript. Z. Yu conceived the idea and edited the manuscript.

Acknowledgments

This work was supported by the Key-Area Research and Development Program of Guangdong Province (grant number 2019B010155003), the Guangdong Basic and Applied Basic Research Foundation (grant number 2020B1515120044), and the National Natural Science Foundation of China (grant number 2020B1515120044), and 2019B010155003), and the Guangdong Basic and Applied Basic Research Foundation (grant number NSFC 61902355).

References


