Hdconfigor: Automatically Tuning High Dimensional Configuration Parameters for Log Search Engines

HUI DOU, PENGFEI CHEN, AND ZIBIN ZHENG, (Senior Member, IEEE)
School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China
Corresponding author: Pengfei Chen (chenpf7@mail.sysu.edu.cn)

This work was supported in part by the National Key Research and Development Plan under Grant 2019YFB1804002, in part by the Research and Development Plan in Key Areas of Guangdong Province under Grant 2020B010165002, in part by the National Natural Science Foundation of China under Grant 61902440, Grant 61802448, and Grant U1811462, and in part by the Fundamental Research Funds for the Central Universities under Grant 19lgpy224.

Abstract
Search engines are nowadays widely applied to store and analyze logs generated by large-scale distributed systems. To adapt to various workload scenarios, log search engines such as Elasticsearch usually expose a large number of performance-related configuration parameters. As manual configuring is time consuming and labor intensive, automatically tuning configuration parameters to optimize performance has been an urgent need. However, it is challenging because: 1) Due to the complex implementation, the relationship between performance and configuration parameters is difficult to model and thus the objective function is actually a black box; 2) In addition to application parameters, JVM and kernel parameters are also closely related to the performance and together they construct a high dimensional configuration space; 3) To iteratively search for the best configuration, a tool is necessary to automatically deploy the newly generated configuration and launch tests to measure the corresponding performance. To address these challenges, this paper designs and implements HDConfigor, an automatic holistic configuration parameter tuning tool for log search engines. In order to solve the high dimensional optimization problem, we propose a modified Random EMbedding Bayesian Optimization algorithm (mREMBO) in HDConfigor which is a black-box approach. Instead of directly using a black-box optimization algorithm such as Bayesian optimization (BO), mREMBO first generates a lower dimensional embedded space through introducing a random embedding matrix and then performs BO in this embedded space. Therefore, HDConfigor is able to find a competitive configuration automatically and quickly. We evaluate HDConfigor in an Elasticsearch cluster with different workload scenarios. Experimental results show that compared with the default configuration, the best relative median indexing results achieved by mREMBO can reach 2.07×. In addition, under the same number of trials, mREMBO is able to find a configuration with at least a further 10.31% improvement in throughput compared to Random search, Simulated Annealing and BO.

Index Terms
Log search engine, configuration parameter tuning, black-box optimization, Bayesian optimization, random embedding.

I. INTRODUCTION
In the era of big data and artificial intelligence, large scale distributed systems are generating tons of logs in the meantime with data processing. To efficiently operate systems, log-based applications such as anomaly detection [1]–[3] and workflow monitoring [4], [5] have become hot research topics in both academia and industry. As the underlying infrastructure of log-based applications, log search engines like Elasticsearch [6], [7] and Solr [8], [9] are widely deployed to store and analyze these “big” logs. To adapt to varying workload scenarios, log search engines usually expose a considerable large number of configuration parameters to developers that nearly can change all the runtime behaviors. Different settings of these configuration parameters can significantly affect the end-to-end performance of document indexing and...
query. Therefore, to pursue better performance for log search engines, it is necessary to delicately tune their configuration parameters.

In practice, manual configuring is time consuming and labor intensive. From our own practical experience when cooperating with one of the most famous banking companies in China, tuning parameters for their Elasticsearch cluster in production environment usually took a few to ten days. Hence, developers usually have to accept the default configurations [10]. In order to change the situation, many previous research [11]–[22] have studied how to automatically tune the configuration parameters for various applications. However, these studies usually only focus on the inherent configurations of target applications, while ignoring parameters from the runtime environment and operating systems kernels. In fact, both runtime and kernel configurations are closely related to application performance. For example, according to Elasticsearch official guides [23], in addition to Elasticsearch configuration parameters, runtime parameters such as JVM heap size (-Xms and -Xmx) and kernel parameters such as swappiness should also be carefully tuned for improving indexing speed. Besides, in our own practical experiments, we find that the median indexing throughput of Elasticsearch will sharply drop from 49261 docs/s to 7651 docs/s if we change the kernel parameter dirty_ratio to 0 while keeping all the other configurations as default (Experiments are performed on a local cluster of 3 nodes with the percolator workload, and the default values of configuration parameters are shown in Table 3. More detailed information can be found in Section V.). Therefore, the full stack configuration parameters from log search engines, runtime environment and operating system kernel should be all taken into consideration to achieve better performance.

In this paper, we focus on how to automatically tune the full stack configuration parameters for log search engines. Based on the in-depth analysis of previous studies as well as our experimental observations on a practical log search engine cluster, there are mainly three challenges to solve this problem:

- **Black-box objective function.** To solve this problem, a straightforward method is to construct a performance prediction model first and then utilize some search based algorithms to explore the optimal configuration [11]–[14], [24]. However, due to the complex implementation of log search engines, it is very difficult if not impossible to figure out the relationship between configuration parameters and performance, and building a useful prediction model usually requires a considerable number of high-quality observations. As a result, we treat the target performance as a black-box objective function of the configuration parameters and we have to solve the formulated black-box optimization problem.

- **High dimensional configuration space.** The full stack configuration parameters from log search engines, JVM and kernel actually construct a high dimensional configuration space. Although traditional black-box optimization methods such as Bayesian optimization algorithm (BO) [22], [25] have been successfully applied to find best configuration for different application scenarios, these methods cannot be directly utilized to solve this high dimensional black-box optimization problem due to the curse of dimensionality. According to [26]–[28], BO does not scale beyond 10–20 parameters, which is much less than the total number of parameters of log search engines.

- **Automatic configuration update.** Once a new configuration is selected, we should deploy it to the cluster and then observe the performance. However, different parameters usually have quite different updating methods. In Elasticsearch, parameter index.translog.durability can be online updated through REST APIs, while changing its JVM heap size need a restart. In addition, a higher configuration dimensionality usually means more necessary observations and hence, more frequent configuration updates. In order to make developers be free of these complex and frequent operations, an automatic configuration tool is necessary to solve this high dimensional black-box optimization problem.

Unfortunately, none of previous studies can efficiently address all these challenges in the same time according to our analysis. We list the detail comparison of previous studies in Table 1, and more detail introduction of these related work can be found in Section II. Therefore, we propose **HDConfigor** - an automatic configuration tuning system for log search engines. **HDConfigor** consists of three important modules: **Configuration Generator**, **Data Aggregator** and **Test Launcher**. Specifically, **Configuration Generator** is responsible for iteratively generating a new configuration according to current observation set. The core lies in it is our modified Random EMbedding Bayesian Optimization algorithm (mREMBO), which is proposed to solve the high dimensional black-box optimization problem. Once a new configuration is generated, **Test Launcher** is responsible for automatically deploying this configuration and launching tests to measure the corresponding performance metrics. After that, **Data Aggregator** will add the configuration-performance pair into the observation set and then update on its persistent storage. Finally, as shown in Table 1, **HDConfigor** is able to simultaneously address all the three challenges described above to automatically tune high dimensional configuration parameters for log search engines.

In order to evaluate the effectiveness of **HDConfigor**, we compare mREMBO with three other widely used black-box optimization algorithms [31]: Random search, Simulated Annealing and Bayesian optimization. Experimental results on a practical Elasticsearch cluster show that compared with the default configuration, the best relative median indexing throughput achieved by mREMBO are 2.07×, 1.46× and
1.07 × for three different types of workload scenarios respectively. Besides, under the same number of trials, mREMBO is able to find a configuration with at least a further 10.31% throughput improvement compared to the three baseline algorithms. In summary, our work makes the following key contributions:

- As shown in Table 1, few previous studies consider JVM or kernel parameters. With practical experiments, we point out the necessity of considering full stack configuration parameters of log search engines, which can further improve the performance by 16.86% according to our experiments.
- We conclude that the full stack configuration space of Elasticsearch have low effective dimensions. Then based on this observation, we propose mREMBO algorithm to solve the high dimensional black-box optimization problem. The evaluation based on practical experiments shows the effectiveness of mREMBO.
- We design and implement an automatic tool HDConfigor to automatically tune high dimensional configuration parameters for Elasticsearch. By simply re-configuring HDConfigor, it is easy to support other log search engines. We will open source the codes of HDConfigor in the near future.

The rest of this paper is organized as follows. Section II gives a detailed introduction of related work. Section III describes the motivation and formulates the optimization problem. Section IV introduces the design and implementation of HDConfigor. Section V describes our experimental settings. Section VI presents and discusses the experimental results. Finally, Section VII concludes the paper.

II. RELATED WORK

In this paper, we propose a high dimensional black-box optimization algorithm mREMBO and implement HDConfigor to automatically tune the full stack configuration parameters for Elasticsearch. There are already some previous studies targeting on configuration parameters optimization for various objectives. In the following we will give an introduction and discussion of them.

A. PREDICTION MODEL-BASED METHODS

A straightforward method [12]–[14], [24], [32]–[34] to solve the configuration parameter optimization problem is to construct an offline prediction model first and then apply some search algorithms to online find the optimal configuration based on this prediction model. For instance, Xiong et al. [24] utilize an ensemble learning algorithm to build the performance-prediction model and leverage genetic algorithm to search the optimal configuration parameters for HBase. Similarly, for Spark clusters, Yu et al. [12] propose a hierarchical modeling method to build the prediction model and then employ genetic algorithm to find the optimal configuration. However, these prediction-model based methods usually require a considerable number of high-quality observations, which are hard to obtain especially within a high dimensional configuration space. In addition, static prediction models are not able to adapt to workload or hardware changes.

Considering the high dimensional configuration space and changing workload patterns, Jamshidi et al. [11] study the feasibility of transfer knowledge about performance models across environments. Chen et al. [35] use the dependence between configuration parameters in one system to speed up searching the optimal configuration for another system. Besides, Mahgoub et al. [36] combine workload prediction with a cost-benefit analyzer to compute the relative cost and benefit of each reconfiguration step for a future time window. However, due to the high dimensional full stack configuration parameters of Elasticsearch, trying to directly build a useful prediction model through experiments on a practical Elasticsearch cluster is still quite challenging, especially under the specified observation times constraint.
B. RANKING-BASED METHODS
In order to address the challenge of high dimensional configuration space, comparison or rank-based methods are provided in previous studies [15]–[17], [29], [37] to select the most important parameters to shrink the original space. Cao et al. [37] show that some parameters have greater performance impact than others and focusing on a smaller number of more important parameters can speed up auto-tuning systems. Therefore, Nair et al. [29] propose a rank-based approach to reduce the number of training samples. Van Aken et al. [16] use a LASSO algorithm to select the most impactful knobs first and then recommend knob settings based on Gaussian Processes. Bao et al. [17] utilize a weighted Latin hypercube sampling approach to select the sample set and then build a comparison-based model to find the optimal configuration for Kafka. However, obtaining the correct importance of each parameter usually require a considerable number of high-quality observations and what is worse, the importance of parameters are quite different with different objectives and workloads.

C. BLACK-BOX METHODS
Instead of constructing a prediction model or ranking the importance of parameters, there exists a category of heuristic search algorithms that treat the objective function as a black box [18]–[22], [30], [31], [38], [39], mREMBO in HDConfigor can also be divided into this category. Random search is now widely utilized in hyper-parameter optimization for machine learning models [40]. For configuration parameters tuning, Ye and Kalyanaraman [18] propose a recursive random search algorithm based on random sampling for large-scale network parameter optimization. Zhu et al. [30] first propose a divide-and-diverge sampling method and then design a recursive bound-and-search algorithm to tune system configurations within a resource limit. Heinze et al. [38] build a monetary costs function for each parameter configuration and then utilize recursive random search to find the optimal configuration. However, these random-based methods are simple to understand but cannot utilize any useful information generated from historical observations.

Therefore, Cao et al. [19] develop two versions of an enhanced multi-objective simulated annealing approach to solve configuration optimization problem with multiple hard constraints, while Wang et al. [39] provide a control-theoretic approach to continuously tune a distributed application. Further, based on the reward function calculated with observations, an enhanced reinforcement learning approach is proposed in [20] to online tune configurations for web systems, while Zhang et al. [21] design an end-to-end automatic cloud database tuning system using deep reinforcement learning. The most related work to our paper is [22]. It propose an auto-tuning algorithm that leverages Bayesian Optimization to iteratively capture posterior distributions of the configuration spaces and sequentially drive the experimentation. However, BO is restricted to problems of moderate dimension typically up to about only 10-20 [26]–[28], and hence cannot be directly used to solve the high dimensional black-box optimization problem in this paper. Therefore, we proposed mREMBO algorithm based on the Bayesian optimization algorithm to solve the high dimensional black-box optimization problem in order to automatically tune configuration parameters for Elasticsearch.

D. HARDWARE RESOURCE CONFIGURATION
In addition to system performance, configuration optimization is also utilized towards other objectives such as resource management [25], [41]–[43]. For instance, BO [25] and an augmented BO [42] are utilized to find the best configurations of cloud VMs from a broad spectrum of candidates. Klimovic et al. [43] use latent factor collaborative filtering to predict how an application will perform across different configurations and then recommend configurations of cloud compute and storage resources. In this paper, we focus on how to tune the full stack parameters from Elasticsearch, runtime and kernel and regard configurations of hardware resource as given constant. These studies can provide a strong reinforcement and complement for our work.

III. MOTIVATION AND PROBLEM FORMULATION
In this paper, we study how to automatically tune the full stack configuration parameters of Elasticsearch to improve its median indexing throughput. Although we choose Elasticsearch as the target application, it is easy to support other log search engines in HDConfigor. Detailed description about how to add support for other log search engines in HDConfigor can be found in Section IV. Besides, we choose median indexing throughput as the optimization objective because it can tell how many documents Elasticsearch can index per second, which characterizes one of the most important capabilities when Elasticsearch is deployed as a log search engine. Especially, when the the volume of log data is huge, the throughput is considered specially.

A. ELASTICSEARCH ARCHITECTURE
Elasticsearch is an open source Java-based search engine built on top of Apache Lucene [44]. It is able to index and search documents with diverse formats in near real-time. Currently, Elasticsearch is widely applied for storing and analyzing system logs. Figure 1(a) shows the architecture of an Elasticsearch cluster in detail. This cluster consists of three data nodes and one dedicated master node. Each Elasticsearch index has one primary shard and each shard has one replica considering data reliability. Shards in the same color are belonging to the same Elasticsearch index. As Elasticsearch is built on top of Lucence, each shard is actually a Lucence index which stores data of indexed documents.

In order to tune the full stack configuration parameters related to indexing throughput, we illustrate the detail indexing data flow in Figure 1(b). Once a new indexing request arrives, the coordinating node will decide its target primary shard according to a certain routing policy.
FIGURE 1. The architecture and indexing data flow of Elasticsearch.

After successfully indexing the corresponding new documents, the primary shard node will send this request to all its replicas. Finally, it will send back a response to the coordinating node when it has received response from all its replicas.

Next, we focus on the data flow inside each shard. When the indexing request arrives at the shard, the indexed data will first be written into the memory buffer. Considering reliability, Elasticsearch will also write the indexed data as well as operation logs into a transaction log file. After the specified refresh interval, all the newly indexed data in the memory buffer will be refreshed into a new segment file and then become searchable. It is worth noting that these newly generated segment files are still stored in the memory, and they will be flushed into the disk for persistent storage when the total size or the existing time of the transaction log file is larger than a specified threshold. Besides, for the sake of high reliability, the transaction log file also needs to be fsynced into the disk when the \texttt{fsync} interval is exceeds the specified maximum.

From the architecture and indexing data flow of Elasticsearch we can find that, in order to improve the median indexing throughput, we should pay special attentions to configuration parameters which are related to write, refresh, flush and fsync operations.

B. FULL STACK CONFIGURATION PARAMETERS

According to the above description of indexing data flow in Elasticsearch, configuration parameters related to indexing throughput actually exist in the runtime, kernel and Elasticsearch. For example, Figure 2 shows the median indexing throughput of Elasticsearch when the parameter \texttt{threadpool.write.size} varies from 1 to 4 and the JVM heap size varies from 2 GB to 8 GB, while all the others are set as defaults. More detailed description of our experimental settings can be found in Section V. As shown in Figure 2, the median indexing throughput will change from 41940 docs/s to 51861 docs/s by only tuning these two parameters. Specifically, for the same value of \texttt{threadpool.write.size}, tuning JVM heap size is able to improve the throughput as much as 15.6%. On the other hand, with practical experiments we also find that the indexing throughput will sharply drop from 49261 docs/s to 7651 docs/s if we change the kernel parameter \texttt{dirty_ratio} to 0 while keeping the other parameters as default. Therefore, all of these full stack configuration parameters from Elasticsearch, runtime and kernel should be taken into consideration to tune for a better median indexing throughput.

It is worth noting that the JVM heap size is not 8 GB in the best configuration. In detail, from Figure 2 we can find that for the same \texttt{threadpool.write.size}, a larger JVM heap size does not lead to a better throughput. Besides, the best JVM heap size configurations quite different across different values of \texttt{threadpool.write.size}. Actually, due to the complex implementation of Elasticsearch, the relationship between performance and configurations is quite difficult to predict. Hence, instead of trying to construct an explicit prediction model, we will formulate the full stack configuration parameters optimization problem as a high dimensional black-box optimization problem.

C. PROBLEM FORMULATION

In this paper, we focus on how to automatically tune the full stack configuration parameters of Elasticsearch to improve its median indexing throughput. To achieve this target, first we formulate it as a high dimensional ($D$ dimensions in total) black-box optimization problem. Let $x_i$ indicate the $i$-th ($i = 1, \ldots, D$) configuration parameter and $x_i$ takes values from a finite domain $\text{Dom}(x_i)$. In our case, $x_i$ is either a numerical or a categorical variable. For categorical parameters, we use its index to represent the corresponding category and let $x_i$ equal
to the index. Thus, the full stack configuration parameters space can be denoted as $X = \prod_{i=1}^{D} Dom(x_i)$. We denote the median indexing throughput under configuration $x \in X$ as $f(x)$, our target is to find the optimal configuration $x^\star$ that maximizes the median indexing throughput of Elasticsearch:

$$x^\star = \arg \max_{x \in X} f(x)$$

Due to the complex implementation of Elasticsearch, we treat $f(\cdot)$ as a black-box function that only allows us to observe its value $f(x)$ under a certain configuration $x \in X$ through practical experiments. Since evaluation is expensive, the total times of observation is often restricted in practice. We define this restricted observation times as a constraint, which is denoted as $OT$. Now we can obtain the following $D$-dimensional black-box optimization problem:

$$x^\star = \arg \max_{x \in X} f(x) \quad \text{s.t.} \quad \text{observation times} \leq OT \quad (1b)$$

### IV. HDCONFIGOR ARCHITECTURE

#### A. DESIGN OVERVIEW

In order to automatically tune the full stack configuration parameters of Elasticsearch to optimize the median indexing throughput, we design and implement HDCONFIGOR, an automatic High Dimensional Configuration parameters tuning tool. The high-level architecture of HDCONFIGOR is shown in Figure 3. Once the configuration optimization process is necessary to be launched, developers can specify the configuration parameter space and observation time constraint. After that, HDCONFIGOR is able to automatically generate new configurations and update them to the Elasticsearch cluster. To obtain the median indexing throughput under newly generated configurations, HDCONFIGOR is also able to automatically launch performance tests utilizing benchmark tools. When the constrained observation times is reached, HDCONFIGOR will report the best configuration to developers.

In detail, HDCONFIGOR mainly consists of three modules, namely Configuration Generator, Data Aggregator and Test Launcher. It works in an iterative way. For each iteration we have:

**Configuration Generator** is responsible for generating a new configuration according to the current observation set stored in Data Aggregator. In order to achieve a tradeoff between exploitation and exploration, the new configuration should be either better than the currently best configuration, or help observe more information about the black-box objective function. The core lies in Configuration Generator is our modified REMBO (mREMBO) algorithm, which is proposed based on the Random EMbedding Bayesian Optimization (REMBO) algorithm [26], [27]. Instead of directly solving the high dimensional black-box optimization problem described as Problem 1, mREMBO first generates a low dimensional embedded space and then performs Bayesian optimization in this embedded space to generate new configurations.

**Data Aggregator** is responsible for collecting newly generated configurations from Configuration Generator and calculating the median indexing throughput utilizing performance metrics form Test Launcher. For the sake of system stability, all configuration changes should be able to rollback to the latest version. To this end, these configuration-performance pairs will be stored in a persistent database such as HBase or Elasticsearch.

**Test Launcher** is responsible for automatic deployment of the new configuration and tests on the Elasticsearch cluster to measure the corresponding performance metrics. It is worth noting that practical testbed may contain noises due to the performance interference generated by other co-located applications and network jitters. In order to mitigate the impact of performance variability [45], Test Launcher repeats each iteration for a certain times and send the averages of target metrics back to Data Aggregator.

#### B. MREMBO IN HDCONFIGOR

1) **BAYESIAN OPTIMIZATION**

Bayesian optimization (BO) [46]–[48] is a framework to solve optimization problem like Problem 1 where the objective function $f(\cdot)$ is unknown but can be observed through experiments. Two key parts of the BO framework are: a prior distribution which captures the beliefs about $f(\cdot)$, and an acquisition function which quantifies the expected value of
learning the value of $f(\cdot)$ under each configuration $\overrightarrow{x} \in \mathcal{X}$. As shown in Figure 4, in order to optimize a testing black-box objective function $f(\cdot)$, BO repeats the following phases: (i) use the prior distribution and the acquisition function to determine the next best configuration $\overrightarrow{x}_{nb}$; (ii) observe the result of the black-box objective function $f(\overrightarrow{x}_{nb})$; (iii) add the new configuration-performance pair $\langle \overrightarrow{x}_{nb}, f(\overrightarrow{x}_{nb}) \rangle$ to the observation set and then update the prior distribution.

BO has been demonstrated to outperform other state-of-the-art black-box optimization algorithms when observations are expensive and observation times constraint is low. However, BO is restricted to optimization problems with up to 10-20 dimensions [26]–[28] for the following two reasons. First, the number of required observations grows exponentially with the dimensionality. Second, the global optimization for the high dimensional acquisition function is difficult to obtain. Considering the full stack configuration parameters of Elasticsearch, we need to propose an effective high dimensional black-box optimization algorithm.

2) LOW EFFECTIVE DIMENSION
The full stack configuration parameters of Elasticsearch construct a high dimensional configuration space. However, through detailed analysis of these parameters and extensive experiments on a practical Elasticsearch cluster, we conclude that there are mainly three categories of configuration parameters actually do not change the median indexing throughput:

- **Irrelevant configurations.** This kind of configuration parameters are totally irrelevant with the optimization objective. For example, for a server without NUMA nodes, kernel configurations related to vCPU scheduling among NUMA nodes such as numa_balancing can not make any contribution. Besides, some Elasticsearch configurations belonging to search operation such as threadpool.search.size are also irrelevant with the indexing performance.

- **Dependent configurations.** This kind of configuration parameters only work when some other configuration parameters are set to certain values. For instance, Elasticsearch parameter index.translog.sync_interval only works when index.translog.durability is set to async. In addition, all parameters related to the CMS garbage collector will loose effect if developers choose to use G1 as the JVM garbage collector by configuration -XX:+UseG1GC.

- **Conflicting configurations.** This kind of configurations work for the same function but are mutually exclusive. For example, kernel parameters dirty_ratio and dirty_bytes are used to set the maximum amount of system memory that can be filled with dirty pages before flushing to disk. In general, dirty_bytes works as the counterpart of dirty_ratio, and only one of them can be updated at a time. When one is written, it will be immediately active to evaluate the dirty memory limits and the other one will be set as 0 automatically.

That is to say, the high dimensional black-box objective function $f(\cdot)$ in Problem 1 has a feature of “low effective dimension” and we denote the effective dimension of $f(\cdot)$ as $d_e$. Note that due to the complex relationship between performance and configurations, it is impossible to directly find out these most effective configuration parameters for Elasticsearch.

3) MREMBO
Based on the REMBO algorithm recently proposed in [26], [27], we propose a modified REMBO (mREMBO) algorithm to automatically tune high dimensional configuration parameters for Elasticsearch to improve its median indexing throughput. The key idea lying behind mREMBO is the following theorem:

**Theorem 1:** For the given objective function $f(\cdot)$ with effective dimension $d_e$ and a random matrix $A \in \mathbb{R}^{D \times d}$ with independent entries sampled according to $N(0, 1)$ and $d \geq d_e$, with probability 1, for any $\overrightarrow{x} \in \mathcal{X}$, there exists a $\overrightarrow{y} \in \mathcal{Y}$ ($\mathcal{Y} = \prod_{j=1}^{d} \text{Dom}(y_j)$) such that $f(\overrightarrow{X}) = f(A \overrightarrow{Y})$.

**Proof:** The proof of Theorem 1 can be found in [26], [27] and is omitted here for brevity.

Theorem 1 implies that for any optimal configuration $\overrightarrow{X}^* \in \mathcal{X}$, there exists a corresponding point $\overrightarrow{Y}^* \in \mathcal{Y}$ that $f(\overrightarrow{X}^*) = f(A \overrightarrow{Y}^*)$. Therefore, for the high dimensional black-box optimization problem described as Problem 1, instead of directly optimizing $f(\cdot)$ in the high dimensional space $\mathcal{X}$, mREMBO first introduces a random embedding matrix $A \in \mathbb{R}^{D \times d}$ ($d \geq d_e$) and then performs BO to optimize function $g(\overrightarrow{Y}) = f(A \overrightarrow{Y})$ in the low dimensional embedded space $\mathcal{Y}$.

The detailed process of mREMBO is described in Algorithm 1. Line 1-3 is responsible for generating the embedded space $\mathcal{Y}$ and specifying its bounded region set. Then in this low dimensional space $\mathcal{Y}$, mREMBO performs BO algorithm as well as data transformation to iteratively generate the next best configuration for Elasticsearch (shown in line 4-11). When the total observation times achieve the specified constraint $OT$, mREMBO will terminate and
report the best configuration to developers. Next, we give a detailed discussion about the setting of hyper-parameters in mREMBO:

**Algorithm 1 mREMBO**

**Input:** Configuration space $X$, Observation times constraint $OT$, Embedded dimensionality $d$

**Output:** Next best configuration $X_{nb}$.

1. Transform all configurations in $X$ to be numerical and rescale $X$ to space $X_{res} = [-1, 1]^D$.
2. Generate a random matrix $A \in \mathbb{R}^{D \times d}$ with independent standard Gaussian entries.
3. Set the bounded region set as $Y = [-\sqrt{d}, \sqrt{d}]^d$.
4. Initialize with random points in $Y$ and get the initial GP model as the prior function of $g(\cdot)$.
5. for $t = 1, 2, \ldots, OT$ do
   6. Find the next best point $\overrightarrow{y}_{nb} \in Y$ according to the current GP model and the EI acquisition function.
   7. Project the product of $A\overrightarrow{y}_{nb}$ into $X_{res}$ and then rescale to obtain the next best configuration $X_{nb} \in X$.
   8. Send out $X_{nb}$ to deploy and Wait for the median indexing throughput $f(X_{nb})$, use it as the observation result of $g(\overrightarrow{y}_{nb})$.
   9. Augment the observation set $OB_{t+1} = \{OB_t, (\overrightarrow{y}_{nb}, g(\overrightarrow{y}_{nb}))\}$.
10. Update the GP model with $OB_{t+1}$.
11. end
12. Report the best configuration.

**a: PRIOR FUNCTION**

As most BO studies do [22], [25] [42], [49], we also choose Gaussian Process (GP) as the prior function $g(\overrightarrow{y})$. Specifically, a GP is a distribution over functions and is defined by its mean function $\mu$ and its covariance function (also called kernel) $\kappa$. When the observation set is augmented from $OB_t$ to $OB_{t+1}$, we can easily update the GP model and then obtain the posterior distribution of function $g(\overrightarrow{y})$ according to Bayes’ theorem.

**b: ACQUISITION FUNCTION**

Acquisition function is utilized to search the next best point $\overrightarrow{y}_{nb} \in Y$. Typically, acquisition functions are optimized by choosing points where the predictive mean is high (exploitation) and where the variance is large (exploration). Therefore, acquisition functions need to be carefully designed to tradeoff between exploration and exploitation. There are three main strategies to design an acquisition function [50]: Probability of Improvement (PI), Expected Improvement (EI) and Gaussian Process Upper Confidence Bound (GP-UCB).

In mREMOB we also choose to use EI as the acquisition function as most BO applications do:

$$acq(\overrightarrow{y} | OB_t) = \mathbb{E}(\max\{0, g_{t+1}(\overrightarrow{y}) - g(\overrightarrow{y}_{*\text{curb}}) | OB_t\}).$$

where $\overrightarrow{y}_{*\text{curb}}$ is the best point from current observation set $OB_t$ and the next best point is calculated by:

$$\overrightarrow{y}_{nb} = \arg\max_{\overrightarrow{y} \in Y} acq(\overrightarrow{y} | OB_t).$$

**c: INITIAL POINTS**

As described by Line 4 in Algorithm 1, in order to build the initial GP model for function $g(\cdot)$, mREMBO will generate a few random points (usually 3-5 in our experiments) in space $Y$ and then observe the median indexing throughput under the corresponding configurations in $X$. It is worth noting that these initial operations only need to be executed once for the same objective function and can be done offline before the execution of mREMBO. Hence, these initial observation times generated by this step will not be calculated into the total observation times.

**d: EMBEDDED DIMENSION AND BOUNDED REGION SET**

The setting of embedded dimension $d$ and the bounded region set of the embedded space $Y$ are two of the most important hyper-parameters of mREMBO. Specifically, if we ignore the observation times constraint, we can achieve the following theorem:

**Theorem 2:** For the given function $f_{res}(\cdot)$ with effective dimension $d$, subject to the box constraint $X_{res} = [-1, 1]^D$, let $\overrightarrow{y}^* \in X_{res}$ be an optimal point of $f_{res}(\cdot)$. If the embedded matrix $A$ is a $D \times d$ ($d \geq d_e$) random matrix with independent standard Gaussian entries, then with probability at least $1 - \epsilon$, there exists an optimizer $\overrightarrow{y}^* \in Y$ such that $f(A\overrightarrow{y}^*) = f_{res}(\overrightarrow{y}^*)$ and $\|\overrightarrow{y}^*\|_2 \leq \frac{d_e}{\epsilon}$.

**Proof:** The proof of Theorem 2 follows [26], [27] and is omitted here for brevity.

Therefore, as the same with REMBO, we also set the bounded region set of the embedded space $Y$ to be $[-\sqrt{d}, \sqrt{d}]^d$. Specifically, according to Theorem 2, in order to increase the probability that $Y$ contains the optimum (i.e., the probability of finding the optimal configuration for Elasticsearch), we should increase the embedded dimension $d$. However, the number of observations required by the prior distribution in mREMOB grows exponentially with the dimension and the global optimization for the high dimensional acquisition function is also difficult to obtain. Therefore, considering the specified observation times constraint, we must carefully choose the value of $d$. The impact of different settings of embedded dimension in mREMBO will be discussed in Section VI-A.

**C. IMPLEMENTATION**

We implement **HDConfig** for Elasticsearch using Python 3.7 [51], considering its impressive simplicity and strong compatibility. As shown in Figure 3, in addition to iteratively generate new configurations with the mREMOB algorithm, **HDConfig** is also required to be able to automatically execute configuration updates and launch performance tests on the Elasticsearch cluster. Generally speaking,
configuration parameters from Elasticsearch, runtime and kernel usually have quite different updating methods. For instance, changing the Java runtime configurations need to modify the file "./elasticsearch-*//config/jvm.options" and then restart the whole cluster, while updating most kernel configurations only need a `sysctl -p` command. As a result, we use the IT automation tool Ansible [52] to automate this complex configuration updating procedure through writing dedicated playbooks. On the other hand, Rally benchmark tool [53] is utilized in HDConfigor to measure the median indexing throughput of Elasticsearch under newly generated configuration. In order to automate these performance tests, we also write dedicated playbooks in Ansible to automatically launch Rally and collect the resulting performance metrics.

The total line of code of HDConfigor is about 1500, mainly including the mREMBO algorithm and Ansible playbooks for automatic execution. We use Python to implement the major part of HDConfigor and utilize the interface provided by the Ansible tool to achieve the automatical configuration updating function in HDConfigor. Specifically, with dedicated playbooks written in YAML for deploying Elasticsearch to the local cluster, updating configuration parameters from kernel, JVM and Elasticsearch, running benchmarks with the Rally tool, HDConfigor is able to automatically generate the next best configuration according to the mREMBO algorithm and obtain the corresponding performance metrics after testing. With these Python codes and YAML scripts, HDConfigor will output the current best configuration for Elasticsearch when the observation times meet the specified constraint.

Besides, HDConfigor can be easily extended to support automatically configuration parameters optimization for other log search engines. Taking Solr as an example, in order to add support for Solr in HDConfigor, we only need to add Solr’s configuration files and modify the Ansible playbooks according to Solr’s deploying and testing procedures. Other components such as mREMBO and performance metric collecting can be directly adopt. We will open source the whole project after adding support for other log search engines such as Solr.

V. EXPERIMENTAL SETUPS

A. HARDWARE PLATFORM

HDConfigor is evaluated on a cluster of three nodes connected with LAN, each node has 16 GB memory and an Intel(R) Core(TM) i7-7700 processor with 4 physical cores. In our experiments, the source code of HDConfigor is deployed on one dedicated node while Elasticsearch and Rally are separately deployed on the other two nodes, in order to avoid performance interference as possible. We will modify HDConfigor to add support for a large scale Elasticsearch cluster in the future work. The operating system running on each node is Ubuntu 18.04 with kernel version 4.15.0. The versions of Elasticsearch, Rally, JDK and Ansible utilized in our experiments are 6.5.4, 1.3.0, 1.8.0 and 2.8.5, respectively.

| Name      | Documents | Uncompressed Size |
|-----------|-----------|-------------------|
| percolator| 20000000  | 104.9 MB          |
| geopoint  | 60844404  | 2.3 GB            |
| geonames  | 11396505  | 3.3 GB            |
| nested    | 11203029  | 3.4 GB            |
| noaa      | 33659481  | 9.0 GB            |
| eventdata | 20000000  | 15.3 GB           |
| pmc       | 574199    | 21.7 GB           |
| http_logs | 247249096 | 31.1 GB           |
| nyc_taxi  | 16534692  | 74.3 GB           |

In order to mitigate the impact of performance variability [54], [55] [45] resulting from workloads, networking and other hardwares, we use the median throughput as the objective performance metric and repeat all the experiments for 3 times to improve the confidence of our results and the correctness of our conclusions. A few previous work [22], [56] [57] have studied the problem that how to further reduce the performance variability, we leave these improvements as our future work.

B. BENCHMARKS

We employ Rally [53] to benchmark the Elasticsearch cluster under different configurations. Race, track and challenge are three important concepts in Rally. Specifically, a race is the execution of a benchmarking experiment. We can choose different benchmarking scenarios (called tracks) for a race and each track may have several different challenges, which record a list of operations to run such as checking cluster health, indexing, forcing merge and so on. Table 2 lists the detail information of representative online available tracks in Rally in ascending order of uncompressed data size. In our experiments, according to the hardware resources of the local cluster, we choose to use percolator, geonames and eventdata to simulate 3 different workload scenarios. Considering the potential performance fluctuation of the Internet, we download all the relevant data of these tracks to the Rally node before the experiments. In addition, we design our own challenges for each track in order to obtain the median indexing throughput with only necessary operations. To mitigate the impact of noise, we execute each race for 3 times and use the average as the observation result.

C. CONFIGURATION PARAMETERS

We choose 41 configuration parameters including numerical or categorical parameters in total according to the official guides [23], [58] as well as empirical information from our practical experiments and other developers. Instead of considering configuration parameters only from Elasticsearch itself, these parameters are chosen from the full stack configuration space, namely: kernel, JVM and Elasticsearch. It is worth noting that we have already omit many evidently low effective configuration parameters, and the whole configuration space of Elasticsearch is actually much higher dimensional. We list

Table 2. Representative online available tracks in Rally.

In order to mitigate the impact of performance variability [54], [55] [45] resulting from workloads, networking and other hardwares, we use the median throughput as the objective performance metric and repeat all the experiments for 3 times to improve the confidence of our results and the correctness of our conclusions. A few previous work [22], [56] [57] have studied the problem that how to further reduce the performance variability, we leave these improvements as our future work.
TABLE 3. The selected full stack configuration parameters of Elasticsearch.

| Category       | Parameter                                | Description                                      | Default          | Range             |
|----------------|------------------------------------------|--------------------------------------------------|------------------|-------------------|
| kernel.net     | tcp_max_syn_backlog                      | max connection number without acks               | 1024             | 128-4096          |
|                | tcp_slow_start_after_idle [categorical]   | use TCP slow start or not                       | 1                | [0.1]             |
|                | tcp_twReuse [categorical]                | reuse sockets in TIME_WAIT state for new connections or not | 0                | [0.1]             |
|                | tcp_abort_on_overflow [categorical]       | reset new connections when overflowed or not     | 0                | [0.1]             |
|                | netdev_max_backlog                       | max number of packets in the queue              | 1000             | 100-5000          |
|                | somaxconn                                | max number of server's listening socket          | 128              | 128-4096          |
| kernel.vm      | swappiness                               | frequency to drop caches and swap out application pages | 60              | 0-100             |
|                | dirty_ratio                              | percentage of dirty pages in memory             | 20%              | 10-100%           |
|                | dirty_background_ratio                   | max percentage of dirty pages before background writeback processes | 10%             | 0-100%            |
|                | dirty_expire_centisecs                   | max duration interval of dirty pages             | 3000             | 1000-10000        |
|                | dirty_writeback_centisecs               | interval between pfflush writeback               | 500              | 100-1000          |
|                | overcommit_memory [categorical]          | over-use physical memory or not                 | 0                | [0.1, 2]          |
|                | overcommit_ratio                         | max memory ratio allow to overcommit             | 50%              | 0-100%            |
|                | vfs_cache_pressure                       | frequency to retain dentry and inode caches      | 262144           | 262144-2621440    |
|                | max_map_count                            | max number of mmap counts in memory             | 1024             | 128-4096          |
| kernel.block   | read Ahead_kb                            | how much extra data kernel reads from disk       | 128              | 64-512            |
|                | rq_affinity [categorical]                | method to send back completed I/O requests       | 1                | [1.2]             |
|                | nr_requests                              | number of requests allocated in block layer      | 128              | 64-256            |
| JVM            | -Xms -Xmx                                | Java heap size                                  | 2G               | 2G-8G             |
|                | -XX:NewRatio                             | ratio of young size to tenured size              | 2                | 1-9               |
|                | -XX:SurvivorRatio                        | ratio of eden size to survivor size in young percentage of total GC time spent in the young Eden heap | 8                | 1-9               |
|                | -XX:G1RSetUpdatingPauseTimePercent       | Update RS phase                                  | 10%              | 5-50%             |
|                | -XX:MaxGCPauseMillis                     | soft target of max GC pause time                | 200              | 100-500           |
|                | -XX:ParallelGCThreads                    | thread number used during the parallel phases of GC | default | 1-8               |
| ES.index       | index.refresh_interval                   | how often to perform a refresh operation         | 1s               | 1s-60s            |
|                | index.merge.scheduler.max_thread_count   | max thread number per shard for merge           | default | 1-4               |
|                | indices.memory.index_buffer_size         | percentage of total heap size used as indexing buffer | 10%           | 10%-50%           |
|                | indices.memory.min_index_buffer_size     | absolute min indexing buffer size                | 48mb             | 48mb-200mb        |
|                | indices.fielddata.cache_size             | memory used for the field data cache            | default | 5%-40%            |
| ES.translog    | index.translog.duration [categorical]    | fsync and commit the translog after every request or every sync_interval | request | [request,async] |
|                | index.translog.flush_threshold_size      | how often translog is fsynced and committed      | 5s               | 1s-120s           |
| ES.threadpool | threadpool.index.size                    | max size of not safely persisted operations     | 512mb            | 128mb-1024mb      |
|                | threadpool.index_queue_size              | thread number in the index threadpool           | 2                | 1-4               |
|                | threadpool.write.size                    | thread number in the write threadpool           | 2                | 1-4               |
|                | threadpool.write_queue_size              | queue size of pending write requests            | default | 50-400            |
|                | threadpool.force_merge_size              | thread number in the force_merge threadpool     | 2                | 1-4               |
|                | threadpool.force_merge_queue_size        | queue size of pending force_merge requests      | default | 50-400            |
|                | threadpool.bulk.size                     | thread number in the bulk threadpool            | 2                | 1-4               |
|                | threadpool.bulk_queue_size               | queue size of pending bulk requests             | default | 50-400            |
| ES.global      | bootstrap.mlockall [categorical]         | use mlockall to lock the process address space into RAM or not | false   | [false, true]    |

all used parameters and give a short description in Table 3. Besides, according to our testbed and benchmark requirement, we also give the default value and value range of each parameter. These parameters can be divided into 3 categories and the detailed description are as follows.

- **Kernel configuration parameters.** Kernel configurations parameters can be modified by changing the content of files located in /proc/sys/ and online activated with a `sysctl -p` command. Linux kernel contains a wide range of parameters, covering network, disk, virtual memory and so on. As described in Section III-A, the indexing operation of Elasticsearh involves extensive memory and disk writing. Therefore, we finally select 18 parameters related to networking, disk I/O and virtual memory usage. The default value is the same with a clean system, and note that we modify the default value of `proc/sys/vm/max_map_count` from 65530 to 262144, according to Elasticsearch’s minimum requirement.

- **JVM configuration parameters.** As the runtime environment of Elasticsearh, configuration parameters of JVM are highly related to the performance of Elasticsearh. Parameters can be modified offline in the `jvm.options` configuration file and then Elasticsearh runs as a Java process. We choose to tune 6 JVM parameters in our experiments, which are related to the configuration of Java heap and the garbage collector. Generally speaking, the initial heap size and the...
maximum heap size will be configured as the same value and we count them as one parameter. Besides, we choose to use G1 Garbage Collector in our experiments due to its soft real-time and high throughput.

- Elasticsearch configuration parameters. Elasticsearch itself provides quite a large number of configuration parameters to developers. In general, index-related settings can be divided as static and dynamic [23]. Static parameters need to be offline modified in the `elasticsearch.yml` configuration file and dynamic parameters can be online changed through REST APIs. As described in Section III-A, indexing mainly involves `write`, `refresh`, `flush` and `fsync` operations, leading to massive memory and disk writing. Therefore, we select 17 configuration parameters related to the settings of `index`, `threadpool` and `translog`.

D. BASELINE ALGORITHMS

HDCConfigor utilizes mREMBO to solve the high dimensional black-box configuration parameters optimization problem described as Problem 1. In order to evaluate the effectiveness of mREMBO, we compare it with three other black-box optimization algorithms already widely used to tune configuration parameters for different software systems, namely Random search [18], [40], Simulated Annealing [59], [60] and Bayesian optimization [22], [61]. These baseline algorithms are also widely used in tuning hyper-parameters for machine learning models. Therefore, these algorithms are already widely utilized as baseline black-box optimization algorithms in previous work such as [17], [21], [30]. In the following, we provide a brief introduction of each algorithm and give a short description of their hyper-parameter settings in our experiments if necessary.

1) RANDOM SEARCH

Random search explores each dimension of the configuration parameters uniformly at random. With the same observation times constraint, it is more efficient than Grid search, especially in a high dimensional configuration parameters space. However, it is not guaranteed to find a better configuration than Grid Search. For each iteration, in order to generate a new random configuration, we use the `randint()` function in Python to generate a random value for numerical parameters and a random index for categorical parameters.

2) SIMULATED ANNEALING

SA explores new configuration in an iterative way controlled by temperature. For each temperature: it first generates a new configuration through randomly altering the old one, then it observes the performance under this configuration, finally, according to a probability calculated with the observation and current temperature, it will decide whether to accept this new configuration. Utilizing the acceptance probability, SA is able to avoid getting trapped by local optimal configurations early in the process. In our experiments, we use the open source implementation from [60] and set the number of iterations to be the specified observation times constraint.

3) BAYESIAN OPTIMIZATION

BO also works in an iterative way, the main steps during each iteration is to update the probabilistic model of the objective black-box function according to current observation set, and then utilize this model and the acquisition function to determine the next configuration to be observed. The detail description of BO can be found in Section IV-B. In our experiments, we use the open source implementation of BO from [61] and use GP as its prior function, EI as its acquisition function and 3 initial points to build the initial GP model.

4) mREMBO

mREMBO proposed in this paper is to solve the high dimensional black-box configuration parameters optimization problem for Elasticsearch. It first introduces a random embedding matrix to generate a low dimensional embedded space and then performs BO on this embedded space to iteratively search for the optimal configuration. The detail description of mREMBO can also be found in Section IV-B. In our experiments, mREMBO uses the same hyper-parameter settings with BO. In addition, without specially stated, the embedded dimensionality is set to be 10 for Elasticsearch.

VI. EXPERIMENTAL RESULTS

In this section, we first evaluate the effectiveness of mREMBO through comparing it with the three other black-box optimization algorithms. Then we show the performance of mREMBO when applied to different workload scenarios. Next, we discuss the impact of different settings of embedded dimensionality in mREMBO, and finally, we verify the necessity of full stack configuration parameters optimization in Elasticsearch.

A. EFFECTIVENESS OF MREMBO

Q1: Do previous black-box optimization algorithms already perform well in the high dimensional scenario? If not, how does mREMBO work?

To answer these question, we use the `percolator` track in Rally to evaluate the effectiveness of mREMBO and three other baseline black-box optimization algorithms in optimizing full stack configuration parameters for Elasticsearch. The defaults of all the 41 configuration parameters are set as Table 3 and the observation times constraint is 30. Experimental results are shown in Figure 5.

There are two key factors to measure the effectiveness of these algorithms: the best relative median indexing throughput over default and the necessary number of observation times to find this best configuration. Figure 5(a) illustrates the best relative throughput achieved along with observation times. We can find that mREMBO is able to find the best configuration with fewest observation times. In detail, the best relative throughput achieved by mREMBO, Random, SA and
BO are $2.07 \times$, $1.91 \times$, $1.97 \times$ and $1.87 \times$ with 3, 14, 18 and 4 observations, respectively. That is to say, under the same observation times constraint, mREMBO is able to improve the median indexing throughput of Elasticsearch by up to 107.26% over the default configuration, and at least a further 10.31% improvement over default compared with the other 3 baseline algorithms. Besides, with the smallest necessary number of observation times, mREMBO is more suitable for production systems with a considerable expensive cost for each observation. This impressive improvement is achieved since by introducing a random embedding matrix, mREMBO can take full advantage of the low effective dimension feature in Elasticsearch and actually performs BO in a low dimensional embedded space to tune the full stack configuration parameters.

Figure 5(b)-5(d) shows the relative throughput over default under each iteration of these 4 algorithms in detail. As can be seen in Figure 5(d), mREMBO and BO indeed work in a quite similar way to make a tradeoff between exploitation and exploration for each iteration. However, due to the course of dimension, BO is restricted to only 10-20 dimensions. Hence, it has the lowest median indexing throughput. On the other hand, although Random search and SA performs better than BO, the number of observation times may prevent them to be directly applied in optimizing full stack configuration parameters for Elasticsearch. As shown in Figure 5(b) and Figure 5(c), both algorithms need 10+ more observation times compared with mREMBO and BO to obtain their best configuration. What is worse, since Random search does not utilize any historical information and generates new configurations in a totally random way, it actually works without any reliable guarantees.

It is worth noting that both BO and mREMBO utilize 3 offline observations to build the initial GP model in our experiments. However, even taking these initial points into account, the total number of observation times spent by mREMBO on finding the best configuration is still much smaller than Random search and SA.

Q2: What is the impact of embedded dimension in mERMBO on its effectiveness? How to choose an appropriate embedded dimension for mREMBO?

The embedded dimension $d$ is one of the most important hyper-parameters in mREMBO. According to Theorem 2, it can directly impact the probability of finding the optimal configuration for Elasticsearch. In this experiment, we use the percolator track and set $d = 1, 5, 10, 15$, respectively. The observation times constraints are all set to be 30.

As shown in Figure 6, $d = 10$ indeed performs better than all other settings under track percolator. In detail, the best relative median indexing throughput of $d = 1, d = 5, d = 10, d = 15$ over default are $1.44 \times$, $1.78 \times$, $2.07 \times$ and $1.85 \times$ utilizing at least 6, 10, 3 and 19 observations, respectively. That is to say, when the embedded dimension is not appropriately set, mREMBO will fail to find a satisfied configuration.
FIGURE 6. The performance of mREMBO under different settings of embedded dimension.

FIGURE 7. The performance of mREMBO under different configuration parameters spaces.

for Elasticsearch while consuming more observation times. On one hand, when $d$ is too large, the embedded space itself will become high dimensional and the necessary number of observation times to find the best configuration may exceed the specified constraint. On the other hand, when $d$ is too small, the embedded space is not able to cover all the effective dimensions of Elasticsearch and as a result, mREMBO will probably miss the optimal configuration. Generally, the setting of embedded dimension is closely related to the number of effective configuration parameters of the target application and its hardware platform. For the Elasticsearch testbed used in our experiments, we believe that $d = 10$ is an appropriate choice according to the extensive experiments under different Rally tracks.

**Summaries:** Under the same observation times constraint, mREMBO performs much better than Random, SA and BO. Compared with these baseline black-box optimization algorithms, mREMBO is able to achieve at least a further 10.31% improvement in throughput.

**B. NECESSITY OF FULL STACK CONFIGURATION PARAMETERS OPTIMIZATION**

**Q3: Is it necessary to consider the JVM and kernel parameters?**

One of the most important contribution of this paper is that instead of only tuning Elasticsearch parameters, we also take runtime and kernel parameters into consideration and propose mREMBO to explore the best configuration in the resulting high dimensional full stack configuration parameters space. In order to verify the necessity, we evaluate the performance of mREMBO respectively working in full stack parameters space and Elasticsearch parameters space. In this experiment, we use the percolator track and set the observation times constraint to be 30. Experimental results are shown in Figure 7.

First of all, we can find that $d = 5$ is a better setting of embedded dimensionality when mREMBO is applied to tune only Elasticsearch parameters. As described in Section V-C, the total number of Elasticsearch parameters used in our experiments is 17 and therefore, $d = 10$ is certainly too large and instead we set $d = 5$ for this scenario. As can be seen in Figure 7, compared with the default configuration, mREMBO with $d = 5$ is able to improve the median indexing throughput by 90.41% through tuning Elasticsearch parameters only while tuning full stack configuration parameter can improve this throughput as much as 107.26%. That is to say, taking runtime and kernel parameters into consideration is able to achieve a 16.86% improvement over default compared with tuning Elasticsearch parameters alone.

**Summaries:** These results help us draw two conclusions. First, developers usually have to tune the Elasticsearch parameters to fit their own workload scenarios and cluster resources. Second, it is very necessary to optimize parameters from runtime environment, kernel and Elasticsearch, which can further improve the performance of Elasticsearch by 16.86% according to our experiments.

**C. IMPACT OF DIFFERENT WORKLOAD SCENARIOS**

**Q4: What is the impact of varying workload on configuration parameters tuning? How does mREMBO work under different workload scenarios?**

To answer these question, we choose 3 tracks in Rally to simulate different workload scenarios: percolator, geonames and eventdata. These tracks have a quite different document number as well as uncompressed size, detailed description of these tracks can be found in Table 2. The observation times constraint for these tracks is set to be 30.

Figure 8 illustrates the best relative indexing throughput achieved by mREMBO over the default configuration. Specifically, under tracks percolator, geonames and eventdata, mREMBO is able to improve the median indexing throughput by 107.26%, 46.5% and 6.75%, respectively. Therefore, mREMBO is able to be adaptive to different workload scenarios of Elasticsearch. In addition, we also present the improvement of minimum and maximum indexing throughput over default in Figure 8, and we can find that mREMBO is also with high effectiveness in optimizing these two objectives. Note that for the eventdata track, the best maximum indexing throughput is only 3.39% higher than the default. In fact, due to the huge number of documents (20 millions) as well as the large uncompressed size.
TABLE 4. Changes of official recommended parameters after tuning under different tracks.

| Parameter                     | Description                        | Default | percolator | geonames | eventdata |
|-------------------------------|------------------------------------|---------|------------|----------|-----------|
| swappiness                    | frequency to drop caches and swap out app pages | 60      | 58         | 23       | 6         |
| dirty_ratio                   | percentage of memory can be filled with dirty pages | 20%     | 86%        | 61%      | 80%       |
| jvm_heap_size                 | JVM heap size                      | 2GB     | 6GB        | 6GB      | 8GB       |
| index_buffer_size             | percentage of total heap size used as indexing buffer | 10%     | 30%        | 36%      | 31%       |
| refresh_interval              | how often to perform a refresh operation | 1s      | 26s        | 54s      | 58s       |
| translog_durations            | fsync translog per request or every sync_interval | request | async     | async    | async     |
| translog_sync_interval        | how often the translog is fsynced and committed | 5s      | 46s        | 91s      | 119s      |
| mlock_all                     | use mlockall to lock the process in RAM or not | false   | true       | true     | true      |

FIGURE 8. The performance of mREMBO under different workload scenarios.

(15.3 GB), the computing resource of the local single-node Elasticsearch cluster becomes the performance bottleneck. As a result, solely focusing on software configurations is not able to obtain a remarkable performance improvement any more. However, mREMBO can still improve the minimum indexing throughput by 13.15% under track eventdata.

In Table 4, we present the detailed changes of some recommended parameters for indexing optimization from the official guide [23]. These parameters are mainly related to memory and disk writing, which is consistent with the analysis of indexing data flow in Elasticsearch in Section III-A. As can be found in this table, the relationship between configurations and performance is very complex and actually varies with different workload scenarios. For example, in order to improve the indexing throughput, kernel parameter swappiness is modified to 6 under track eventdata, while almost the same with default under track percolator. Therefore, although Elasticsearch has the feature of low effective dimension, it is still impossible to directly find out its most effective configuration parameters. Instead, utilizing this feature, we propose mREMBO to solve the full stack configuration parameters optimization problem for Elasticsearch.

Summaries: Workload changes will lead to new optimal configurations because the effective configuration parameters for different workload scenarios are usually quite different. According to our experiments, mREMBO is always able to find an outstanding configuration under different workload scenarios.

VII. CONCLUSION

In this paper, we design and implement HDConfigor, an automatic full stack configuration parameters tuning tool for log search engines. HDConfigor solves the high dimensional black-box optimization problem through the proposed mREMBO algorithm. Utilizing the low effective dimensionality of Elasticsearch, mREMBO first introduces a random embedding matrix to generate an embedded space and then performs BO in this low dimensional embedded space. Therefore, HDConfigor is able to find a competitive configuration automatically and quickly. We evaluate HDConfigor on a local Elasticsearch cluster with 3 different tracks from the Rally benchmark tool. Experimental results show that mREMBO can improve the median indexing throughput compared with default configuration by up to 107.26%, and mREMBO is able to find a configuration with at least a further 10.31% throughput improvement over default compared to Random search, Simulated Annealing and BO under the same observation times constraint. Besides, we also verify the necessity of taking runtime and kernel configuration parameters into consideration to further improve the performance.

In the future, we will add support for other log search engines such as Solr in HDConfigor and open source our codes. Besides, how to tradeoff between different objectives such as indexing and searching during the configuration tuning process is also an open interesting problem. In addition, it is also necessary to study how to mitigate performance variability resulting from workloads, configuration and networking for experimental confidence and correctness.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “SherLog: Error diagnosis by connecting clues from run-time logs,” ACM SIGARCH Comput. Archit. News, vol. 38, no. 1, pp. 143–154, Mar. 2010.
[2] S. He, J. Zhu, P. He, and M. R. Lyu, “Experience report: System log analysis for anomaly detection,” in Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE), Oct. 2016, pp. 207–218.
[3] M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: Anomaly detection and diagnosis from system logs through deep learning,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS). New York, NY, USA: ACM, 2017, pp. 1285–1298.
[4] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, “CloudSeer: Workflow monitoring of cloud infrastructures via interleaved logs,” ACM SIGOPS Operat. Syst. Rev., vol. 50, no. 2, pp. 489–502, Mar. 2016.
[5] F. Wu, P. Anchuri, and Z. Li, “Structural event detection from log messages,” in Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD). New York, NY, USA: ACM, 2017, pp. 1175–1184.
M. Trotter, T. Wood, and J. Hwang, “Forecasting a storm: Divining optimal configurations auto-tuning of in-memory cluster computing,” ACM SIGPLAN Notices, vol. 53, no. 2, pp. 564–577, Nov. 2018.

M. Trotter, T. Wood, and J. Hwang, “Forecasting a storm: Divining optimal configurations using genetic algorithms and supervised learning,” in Proc. IEEE Int. Conf. Autonomic Comput. (ICAC), Jun. 2019, pp. 1–11.

J. Tan, R. Zhang, T. Zhang, F. Li, J. Chen, Q. Zhang, P. Zhang, H. Qiao, Y. Shi, and W. Cao, “IBTune: Individualized buffer tuning for large-scale cloud databases,” Proc. VLDB Endowment, vol. 12, no. 10, pp. 1221–1234, Jun. 2019.

A. Mahgoub, S. Chaterji, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, and S. Bagchi, “Rafiki: A middleware for parameter tuning of NoSQL datastores for dynamic metagenomics workloads,” in Proc. 18th ACM/SIGPLAN/SIGMOD Conference (Middleware), 2017, pp. 28–40.

D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, “Automatic database management system tuning through large-scale machine learning,” in Proc. ACM Int. Conf. Manage. Data (SIGMOD). New York, NY, USA: ACM, 2017, pp. 1009–1024.

L. Bao, X. Liu, Z. Xu, and B. Fang, “AutoConfig: Automatic configuration tuning for distributed message systems,” in Proc. 33rd ACM/IEEE Int. Conf. Autonomic Softw. Eng. (ASE). New York, NY, USA: ACM, 2018, pp. 29–40.

T. Ye and S. Kalyanaraman, “A recursive random search algorithm for large-scale network parameter configuration,” ACM SIGMETRICS Perform. Eval. Rev., vol. 31, no. 1, p. 196, Jun. 2003.

P. Cao, Z. Fan, R. X. Gao, and J. Tang, “Solving configuration optimization problem with multiple hard constraints: An enhanced multi-objective simulated annealing approach,” 2017, arXiv:1706.03141. [Online]. Available: http://arxiv.org/abs/1706.03141.

X. Bu, J. Rao, and C.-Z. Xu, “A reinforcement learning approach to online web systems auto-configuration,” in Proc. 29th IEEE Int. Conf. Distrib. Comput. Syst., Jun. 2009, pp. 2–11.

J. Zhang, L. Liu, M. Ran, Z. Li, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, and T. Cheng, “An End-to-End automatic cloud database tuning system using deep reinforcement learning,” in Proc. Int. Conf. Manage. Data (SIGMOD). New York, NY, USA: ACM, 2019, pp. 415–425.

P. Jamshidi and N. Siegmund, M. Velez, C. Kastner, A. Patel, and Y. Agarwal, “Transfer learning for performance modeling of configurable systems: An exploratory analysis,” in Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE). Piscataway, NJ, USA: IEEE Press, Oct. 2017, pp. 497–508.

Z. Yu, Z. Bei, and X. Qian, “Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing,” ACM SIGPLAN Notices, vol. 53, no. 2, pp. 564–577, Nov. 2018.

Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. De Freitas, “Bayesian optimization in high dimensions via random embeddings,” in Proc. 23rd Int. Joint Conf. Artif. Intell., 2013, pp. 1778–1784.

Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. De Freitas, “Bayesian optimization in a billion dimensions via random embeddings,” J. Artif. Intell. Res., vol. 55, pp. 361–387, Feb. 2016.

R. Moriconi, M. P. Deisenroth, and K. S. Sesh Kumar, “High-dimensional Bayesian optimization using low-dimensional feature spaces,” 2019, arXiv:1902.10675. [Online]. Available: http://arxiv.org/abs/1902.10675.

V. Nair, T. Menzies, N. Siegmund, and S. Apel, “Using bad learners to find good configurations,” in Proc. 11th Joint Meeting Found. Softw. Eng. (FSE)/ProgLang/PROS. New York, NY, USA: ACM, 2017, pp. 257–267.

Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, “BestConf: Tapping the performance potential of systems via automatic configuration tuning,” in Proc. Symp. Cloud Comput. (SoCC). New York, NY, USA: ACM, 2017, pp. 338–350.

Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, “Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems,” in Proc. USENIX Annu. Conf. (USENIX ATC), 2018, pp. 893–907.

D. A. Menascé, D. Barbará, and R. Dodge, “Preserving QoS of e-commerce sites through self-tuning: A performance model approach,” in Proc. 3rd ACM Conf. Electron. Commerce (EC), 2001, pp. 224–234.

N. Yigitbasi, T. L. Wilke, G. Lia, and D. Epema, “Towards machine learning-based auto-tuning of MapReduce,” in Proc. IEEE 21st Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst., Aug. 2013, pp. 11–20.

N. Siegmund, A. Grebhabn, S. Apel, and C. Kästner, “Performance-influence models for highly configurable systems,” in Proc. 10th Joint Meeting Found. Softw. Eng. (FSE/FSE). New York, NY, USA: ACM, 2015, pp. 284–294.

H. Chen, W. Zhang, and G. Jiang, “Experience transfer for the configuration tuning in large-scale computing systems,” IEEE Trans. Knowl. Data Eng., vol. 23, no. 3, pp. 388–401, Mar. 2011.

A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi, “Sogha: Online reconfiguration of clustered NoSQL databases for time-varying workloads,” in Proc. USENIX Annu. Tech. Conf. (USENIX ATC), 2019, pp. 223–240.

Z. Cao, G. Kuennwing, and E. Zadok, “Carver: Finding important parameters for storage system tuning,” in Proc. 18th USENIX Conf. File Storage Technol. (FAST), 2020, pp. 43–57.

T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzner, “Online parameter optimization for elastic data stream processing,” in Proc. 6th ACM Symp. Cloud Comput. (SoCC). New York, NY, USA: ACM, 2015, pp. 276–287.

S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, “Understanding and auto-adjusting performance-sensitive configurations,” ACM SIGPLAN Notices, vol. 53, no. 2, pp. 154–168, Nov. 2018.

J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” J. Mach. Learn. Res., vol. 13, pp. 281–305, Feb. 2012.

J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, “vCONF: A reinforcement learning approach to virtual machines auto-configuration,” in Proc. 6th Int. Conf. Autonomic Comput. (ICAC), 2009, pp. 137–146.

C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies, “Arrow: Low-level augmented Bayesian optimization for finding the best cloud VM,” in Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS), Jul. 2018, pp. 660–670.

A. Klimovic, H. Litz, and C. Kozyrakis, “Selecta: heterogeneous cloud storage configuration for data analytics,” in Proc. USENIX Annu. Tech. Conf. (USENIX ATC), 2018, pp. 759–773.

Apache Lucene. Accessed: Oct. 15, 2019. [Online]. Available: https://lucene.apache.org.

A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, “Tuning performance variability,” in Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI), 2018, pp. 409–425.

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,” 2010, arXiv:1012.2599. [Online]. Available: http://arxiv.org/abs/1012.2599.

J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, “Bayesian optimization with inequality constraints,” in Proc. ICML, 2014, pp. 937–945.

B. Shahrifi, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, “Taking the human out of the loop: A Review of Bayesian optimization,” Proc. IEEE, vol. 104, no. 1, pp. 148–175, Jan. 2016.

Y. Dalibard, M. Schnurr, and E. Yoneki, “BOAT: Building autotuners with structured Bayesian optimization,” in Proc. 26th Int. Conf. World Wide Web, Apr. 2017, pp. 479–488.

J. Snoek, H. Larochelle, and R. P. Adams, “Practical Bayesian optimization of machine learning algorithms,” in Proc. Adv. Neural Inf. Process. Syst., 2012, pp. 2951–2959.

Python. Accessed: Nov. 1, 2019. [Online]. Available: https://www.python.org.
H. Dou et al.: Hdconfigor: Automatically Tuning High Dimensional Configuration Parameters

Red Hat Ansible. Accessed: Nov. 1, 2019. [Online]. Available: https://www.ansible.com/products/engine

Rally. Accessed: Nov. 1, 2019. [Online]. Available: https://github.com/redhat-cop/rally

J. Dean and L. A. Barroso, “The tail at scale,” Commun. ACM, vol. 56, no. 2, p. 74, Feb. 2013.

Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok, “On the performance variation in modern storage stacks,” in Proc. 15th USENIX Conf. File Storage Technol. (FAST), 2017, pp. 329–344.

C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, and M. Borg, “Performance issues? Hey DevOps, mind the uncertainty,” IEEE Softw., vol. 36, no. 2, pp. 110–117, Mar. 2019.

A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup, “Is big data performance reproducible in modern cloud networks?” in Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2020, pp. 513–527.

Suse Linux Enterprise. [Online]. Available: https://documentation.suse.com

S. Szykman and J. Cagan, “A simulated annealing-based approach to three-dimensional component packing,” J. Mech. Des., vol. 117, no. 2A, pp. 308–314, Jun. 1995.

Python Module for Simulated Annealing. Accessed: Nov. 1, 2019. [Online]. Available: https://github.com/perrygeo/simanneal

Bayesian Optimization. Accessed: Oct. 22, 2019. [Online]. Available: https://github.com/fmfn/BayesianOptimization

HUI DOU graduated the Ph.D. degree from the Department of Computer Science, Xi’an Jiaotong University, in 2017. From September 2017 to June 2018, he worked as a Senior Engineer with Huawei. He is currently holding a postdoctoral position with an Associate Professor Pengfei Chen with the School of Data and Computer Science, Sun Yat-sen University. He has published more than ten articles in some international conferences and journals, such as ICDCS, SustainCom, and the IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING. His current research interests include performance and power optimization for distributed systems, including configuration tuning, resource utilization improving, and workload scheduling.

PENGFEN CHEN was selected as a member of the Rising Star Program of the Microsoft Research Asia, from July 2012 to November 2012. From June 2016 to January 2018, he worked as a Research Scientist with IBM Research, China. From February 2017 to April 2017, he visited IBM T.J. Watson. He is currently an Associate Professor with the School of Data and Computer Science, Sun Yat-sen University. He is also a Ph.D. Advisor. Especially, he has strong skills in cloud computing. He has published more than 30 articles in some international conferences, including the IEEE INFOCOM, WWW, the IEEE ICSCOC, the IEEE ICWS, and the IEEE BigData, and journals including, the IEEE TRANSACTIONS ON REDEPLENT AND SECURE COMPUTING, the IEEE TRANSACTIONS ON RELIABILITY, the IEEE TRANSACTIONS ON SERVICES COMPUTING, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, and the IEEE TRANSACTIONS ON CLOUD COMPUTING. His current research interests include distributed systems, AIOps, cloud computing, microservice, and blockchain. He serves as a program committee member of multiple conferences. He also serves as a Reviewer of some internal journals, such as the IEEE TRANSACTIONS ON CYBERNETICS, INFORMATION SCIENCE, and NEUROCOMPUTING.

ZIBIN ZHENG received the Ph.D. degree from The Chinese University of Hong Kong, in 2011. He is currently a Professor with the School of Data and Computer Science, Sun Yat-sen University, China. He is also the Chairman of the Software Engineering Department. He published over 120 international journal articles and conference papers, including three ESI highly cited articles. According to Google Scholar, his articles have more than 7000 citations, with an H-index of 42. His research interests include blockchain, services computing, software engineering, and financial big data. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE 2010, the Best Student Paper Award at ICWS 2010. He served as BlockSys’19 and CollaborateCom’16 General Co-Chair, SC2’19, ICICT’18, and IoT’14 PC Co-Chair.