Towards General and Efficient Online Tuning for Spark

Yang Li
Tencent Inc.
thomasyngli@tencent.com

Huaijun Jiang
Peking University &
Tencent Inc.
jianghuaijun@pku.edu.cn

Yu Shen
Peking University
shenyu@pku.edu.cn

Yide Fang
Tencent Inc.
yidefang@tencent.com

Xiaofeng Yang
Tencent Inc.
felixxfyang@tencent.com

Danqing Huang
Tencent Inc.
daisyqhuang@tencent.com

Xinyi Zhang
Peking University
zhang_xinyi@pku.edu.cn

Wentao Zhang
Mila - Québec AI Institute
wentao.zhang@mila.quebec

Ce Zhang
ETH Zürich
ce.zhang@inf.ethz.ch

Peng Chen
Tencent Inc.
pengchen@tencent.com

Bin Cui
Peking University
bin.cui@pku.edu.cn

ABSTRACT

The distributed data analytic system – Spark is a common choice for processing massive volumes of heterogeneous data, while it is challenging to tune its parameters to achieve high performance. Recent studies try to employ auto-tuning techniques to solve this problem but suffer from three issues: limited functionality, high overhead, and inefficient search.

In this paper, we present a general and efficient Spark tuning framework that can deal with the three issues simultaneously. First, we introduce a generalized tuning formulation, which can support multiple tuning goals and constraints conveniently, and a Bayesian optimization (BO) based solution to solve this generalized optimization problem. Second, to avoid high overhead from additional offline evaluations in existing methods, we propose to tune parameters along with the actual periodic executions of each job (i.e., online evaluations). To ensure safety during online job executions, we design a safe configuration acquisition method that models the safe region. Finally, three innovative techniques are leveraged to further accelerate the search process: adaptive sub-space generation, approximate gradient descent, and meta-learning method.

We have implemented this framework as an independent cloud service, and applied it to the data platform in Tencent. The empirical results on both public benchmarks and large-scale production tasks demonstrate its superiority in terms of practicality, generality, and efficiency. Notably, this service saves an average of 57.00% memory cost and 34.93% CPU cost on 25K in-production tasks within 20 iterations, respectively.

1 INTRODUCTION

The rapid growth of the World Wide Web, E-commerce, Social media and other applications is producing massive amounts of ever-increasing raw data every day [32]. Companies often utilize this "big data" to innovate pioneering products and solutions, e.g., improving customer service, marketing, sales, team management, and many other routine operations. To this end, many distributed analytics platforms (e.g., Hadoop Mapreduce [16, 20], Spark [4, 81, 82], Storm [72], Flink [11], Heron [42], Samza [57]) have emerged to deal with this big data trend. Spark is one of the representative systems that enable the manipulation and analysis of large datasets with in-memory cluster computing [82]. As of today, thousands of organizations [70], such as Google, Microsoft, Amazon, Meta, Oracle, Snowflake, Databricks, Tencent, and Alibaba are using Spark in production across a vast range of fields including data processing [21, 46], machine learning [55], graph computing [77], stream computing [71] and database management [4, 78].

The performance of Spark jobs highly depends on the choice of Spark configuration parameters. Misconfiguration can lead to unsatisfying performance such as long runtime, resource contention, and resource under-utilization [2]. For instance, an improper configuration may lead to over 100 times the execution time compared with an elaborately designed one [79]. Therefore, it is crucial to choose proper configurations to achieve high performance (e.g., in terms of execution time or cost). The benefit of tuning is even more significant for periodic (hourly, daily, weekly, etc) jobs, where they occupy a large proportion of Spark jobs in production [2]. In this paper, we focus on tuning periodic Spark jobs.

To achieve a near-optimal performance of periodic tasks, users are required to determine a large number of performance-critical configuration parameters. However, it is very difficult to manually tune the configuration parameters of a Spark task due to (1) the high-dimensionality of the parameter space and (2) complex and non-linear interactions among parameters. In addition, manual tuning is usually time-consuming and labor-intensive, and thus it fails to scale to a huge number of tasks in a data platform. Recent studies propose to utilize auto-tuning techniques to optimize the Spark parameters. During our attempts to apply these approaches in real tuning tasks, we realize three aspects of limitations:

Table 1: Summary of several tuning methods. *General obj.* notes whether the method supports generalized objectives or not, e.g., runtime or cost. *Constr.* indicates if it supports performance constraints. *NOER* refers to the "No Offline Evaluation Required", which usually causes high overhead. *Safety* represents the ability to achieve safe config. acquisition during search. *Adaptive space* notes if it can adjust the search space adaptively to deal with the high-dimensional issue. *Meta-learn* indicates whether it utilizes meta-learning techniques to achieve quick convergence. △ means the method partially supports it with some limitations.

Method	General obj.	Constr.	NOER	Safety	Adaptive space	Meta-learn
RFHOC	×	×	×	×	×	×
DAC	×	×	×	×	×	×
CherryPick	×	△	✓	×	×	×
Tuneful	×	×	✓	×	△	✓
LOCAT	×	×	✓	×	△	×
Ours	✓	✓	✓	✓	✓	✓

(C.1 Limited Functionality) Lots of methods (e.g., DAC [79], LOCAT [76]) are designed to minimize the execution time, i.e., finding the fastest configuration. However, the goal in many scenarios involves the execution cost [2, 64] (i.e., the cheapest configuration), or more generalized objectives such as a weighted combination between runtime and cost. In addition, the tuning process should satisfy some application requirements. For instance, the execution time of a configuration should be lower than a given threshold [2]. These scenarios call for a generalized formulation and framework for tuning, which is not supported by existing methods.

(C.2 High Overhead) Many tuning frameworks [7, 33, 45, 79] belong to the offline tuning paradigm. Concretely, they collect training samples by running jobs with different configurations on a non-production cluster, and then train a performance model to suggest new configurations for Spark job running on the production cluster. Training such an accurate performance model involves lots of offline job executions (e.g., 1000-10000) [76]. This process is very time-consuming and expensive, and often incurs data security issues when accessing business data during multiple job executions in the non-production cluster. In addition, since workloads (e.g., data size) may change as time proceeds [79], offline tuning methods cannot capture and adapt to these dynamics easily.

(C.3 Inefficient Search) The configuration search in many methods [2, 79] suffers from the low-efficiency issue that arises from two aspects: (1) huge search space and (2) intrinsic dilemma of black-box optimization. As dimensionality increases, the number of samples required to learn the performance model grows exponentially (i.e., the curse of dimensionality). Therefore, these methods fail to provide quick convergence in high-dimensional space. Furthermore, this tuning task, in essence, the black-box optimization problem, faces the cold-start issue and challenge when dealing with the exploration and exploitation trade-off. These factors greatly affect the convergence speed. In addition, as studied in optimization community [25, 28], meta knowledge across tasks could help to deal with cold-start and slow convergence issues during search [5, 49, 50, 52]. However, nearly none of the existing methods utilize the tuning history from previous tasks to accelerate the current tuning process.

To address these issues simultaneously (See Table 1), we present a general online tuning framework for Spark, which can efficiently unearth the optimal/near-optimal configurations that minimize the objectives under different scenarios (e.g., runtime or cost) and satisfy application requirements. Concretely, to deal with limited functionality (**C.1**), we first introduce a **generalized formulation** about the tuning problem, which supports various goals and multiple performance constraints conveniently. To solve this generalized optimization problem, we develop a noise-robust Bayesian optimization (BO) [63] based solution. Second, considering the high overhead issue (**C.2**), different from offline approaches that require a large number of job executions in advance, we propose an **online tuning paradigm**, where we tune parameters along with the in-production periodic executions of each Spark job, thus incurring no additional execution cost as in offline methods. This online setting requires that the BO-based framework should converge to good configurations quickly (efficiency) and explore as few bad configurations as possible to get there (safety). To this end, we design a configuration acquisition method, which models and utilizes the safe region from the Gaussian process [61] to achieve safe exploration, and then leverages the expected constrained improvement to trade-off exploration and exploitation when suggesting new configurations. To accommodate the dynamic workload in the online setting, we encode the workload characteristic into the Gaussian process using mixed kernels in BO.

As for the inefficient search issue (**C.3**), we develop three innovative techniques within BO to accelerate tuning. First, Bayesian optimization is known to be difficult to scale to high dimensions. To overcome this high-dimensionality issue, we propose an **adaptive sub-space generation** method to obtain a much smaller space for configuration search, instead of the original huge space. The size of the sub-space will be automatically adjusted based on the intermediate tuning results to balance convergence speed and quality. Second, since gradient methods [62] have shown superior convergence speed compared with black-box methods in numerical optimization, we develop a novel **approximate gradient descent** method within BO to select the next configuration, which can estimate and leverage the derivative information of the objective function to speed up the search process. Third, motivated by the observation that the regions of optimal/near-optimal configurations are similar between two similar tuning tasks, we design a **meta-learning module** that learns the similarity between tasks based on tuning history from previous tasks. Further, this module can transfer useful knowledge, e.g., the optimal configuration or the distribution of good configurations, to the current tuning task. Therefore, it takes fewer configuration trials to find the near-optimal configuration.

(Contributions) The main contributions of this paper can be summarized as follows: **(1)** We propose a general and efficient online tuning framework for Spark. To our knowledge, it is the first framework that can deal with the three issues simultaneously. **(2)** To solve the generalized tuning problem while pursuing safety in the online setting, we design a Bayesian optimization (BO) based solution along with a safe configuration acquisition approach. **(3)** To further accelerate configuration search, we develop three innovative techniques within BO: adaptive sub-space generation, approximate gradient descent, and meta-learning-based knowledge transfer. **(4)** We have implemented this tuning framework as a cloud service, and applied it to the Tencent data platform. The empirical results on public benchmarks and large-scale production tasks demonstrate its generality and efficiency when tuning different objectives. Notably,

compared with manual settings, our service saves 57.00% and 34.93% of memory and CPU cost on average over 25K in-production Spark tasks within 20 iterations, respectively.

2 BACKGROUND

2.1 Spark Framework

Spark is an open-source distributed framework, which simplifies the development of applications that execute in parallel on computing clusters. Spark stores the data in memory as Resilient Distributed Datasets (RDDs), and this design choice significantly reduces I/O costs and speeds up iterative job execution time. RDDs are immutable collections distributed over a cluster of machines and maintained in a fault-tolerant way. Spark provides two types of programming abstractions: transformations and actions, to manipulate RDDs. When a Spark application (job) is submitted, the Spark framework generates a directed acyclic graph (DAG) based on the RDD dependency behind the Spark program. Subsequently, the DAG is split into a collection of stages with each containing a set of parallel tasks. Each task, one per RDD partition, computes partial results of a Spark job. Next, the DAG scheduler schedules the stages to execute, and the cluster manager assigns the tasks to run on executors in parallel.

The resource that can be used by an executor is specified by configuration parameters. For example, `spark.executor.memory` and `spark.executor.cores` determine the memory size and the number of CPU cores allocated for an executor, respectively. A Spark application is controlled by up to 160 configuration parameters, which determine many aspects including dynamic allocation, scheduling, shuffle behavior, data serialization, memory management, execution behavior, networking, etc. For details about more Spark parameters, please refer to the official documentation [68]. Some parameters such as `spark.application.name` do not affect performance. Following previous work [24], in this paper, we focus on tuning 30 parameters that significantly affect job performance.

2.2 Terminology

Configuration Space. Our goal is to find the optimal or near-optimal configuration from the configuration space, which contains N Spark parameters: $x_{p1}, x_{p2}, ..., x_{pN}$. Each p_i corresponds to a specific parameter in Spark, e.g., `spark.executor.instances`, `spark.executor.memory`. The range of the ith parameter p_i is denoted by Λ^i and the whole configuration space can be represented by $\Lambda_{cs} = \Lambda^1 \times \Lambda^2 \times ...\Lambda^N$. Note that, we use x to denote a configuration instance (a vector) in Λ_{cs}, where the ith element x^i corresponds to the value of ith parameter p_i and $x^i \in \Lambda^i$.

Configuration Subspace. A configuration subspace Λ_{sub} is the set of all possible combinations of values of each parameter in the space, which only includes a part of Spark parameters. Therefore, the size of the configuration subspace usually is much smaller than the original configuration space Λ_{cs}.

Tuning Procedure. Given a configuration space and tuning budget, a tuner applies a certain strategy to navigate the configuration space. The goal of the tuning task is to find the optimal configuration that minimizes the underlying objective. Common tuning strategies include random search [8], grid search, genetic algorithm [7, 58, 79], Bayesian optimization [2, 9, 36, 66], etc. In each iteration, the tuner

suggests a configuration, and then a job execution is launched to evaluate this configuration. The budget is some kind of *cost* that is allowed to tune a Spark job; here, the tuning budget is set to the number of iterations. Most existing approaches [7, 32, 79] are designed to optimize execution time only or resort to the offline tuning paradigm. In the following, we investigate the generalized and efficient solutions for online Spark tuning.

3 SYSTEM DESIGN

In this section, we introduce the architecture overview, generalized problem formulation, and base solution framework in turn.

3.1 Overview

Figure 1 presents the workflow of our framework. The framework contains five components: (1) the OnlineTune controller is responsible for orchestrating the entire configuration tuning process, along with interactions with the data platform and end users. (2) The multi-purpose surrogate models are to learn complex relationships between configurations and objective metrics or performance constraints. (3) The efficient and safe configuration generator is to suggest a promising configuration to evaluate for a tuning task. (4) The meta-knowledge learner is capable of leveraging tuning history from previous tasks to further accelerate the search for configurations. (5) The data repository is to store tuning-related data, including runhistory, workload metrics, etc.

To define a tuning task for a Spark job, end users first need to specify the objective and constraints they want to optimize or restrict, and set the overall optimization budget. Then, the iterative tuning workflow is activated. Before the execution of a periodic job starts, the data platform requests the **OnlineTune controller** to pass a new configuration for this run (Step 1 in Figure 1). After the job execution (i.e, the evaluation of the corresponding configuration) is finished, the data platform submits the execution results to the controller (Step 2 in Figure 1). This tuning process can be stopped or resumed when the stopping or restarting condition is reached. When the tuning is stopped, the controller returns the best configuration found during tuning. *Note that, this online tuning procedure does not require any additional job executions (i.e., configuration evaluations) as in offline methods.*

Once the OnlineTuner controller receives a configuration request from the data platform, it starts to train the **surrogate models** (in Section 3.3) on the runhistory related to the current task (Step 3 in Figure 1); the surrogates consider parameter configurations, objectives, constraints and workload characteristics. Then, based on the surrogates, the **efficient and safe configuration generator** (in Section 4) elaborately selects the next configuration over a safe and concise sub-space, together with an efficient approximate gradient descent method (Step 5 in Figure 1). Finally, the configuration is sent out to the OnlineTune controller as the response to the configuration request from the data platform. After the job equipped with the configuration is finished, the controller receives its execution results (i.e., evaluation metrics and workload information) and stores them in the **data repository**. During the above workflow, the **meta-knowledge learner** (in Section 5) aids the configuration generator to efficiently identify and suggest promising configurations (Step 4 in Figure 1) by leveraging the auxiliary knowledge
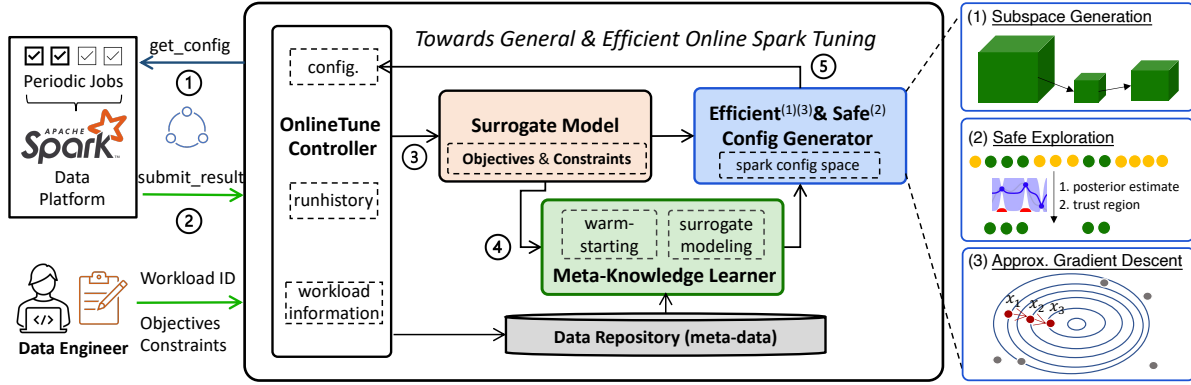
Figure 1: Online Spark tuning architecture and interaction with Spark.

from the tuning history of previous tasks, which is stored in the data repository. This meta-learner could greatly decrease the number of runs to find the near-optimal configurations.

3.2 Generalized Problem Formulation

Given a tuning task, our goal is to find the optimal or near-optimal Spark configuration that minimizes the objective and satisfies performance/safety requirements. To support general Spark tuning cases, we provide a generalized tuning formulation that supports both different objectives and inequality constraints from real applications. Formally, we use $T(\boldsymbol{x})$ to denote the runtime function for the tuning task. The runtime depends on the configuration vector $\boldsymbol{x}$, which includes resource configurations and other execution-related configurations. Let $P(\boldsymbol{x})$ be the price per unit of time for all computing resources used in configuration $\boldsymbol{x}$. Due to the unknown hardware types, VM types, changeable pricing policy, etc, given a $\boldsymbol{x}$, the price per unit time (i.e, $P(\boldsymbol{x})$) is not easy to obtain. Note that, the price $P(\boldsymbol{x})$ is positively correlated with the amount of resource used in $\boldsymbol{x}$ given a fixed computing environment. For simplicity, we use the resource function $R(\boldsymbol{x})$ to replace $P(\boldsymbol{x})$, which indicates the amount of resource used by the job execution with configuration $\boldsymbol{x}$. Further, we formulate this problem as follows:

$$\begin{aligned} \underset{\boldsymbol{x} \in \Lambda_{cs}}{\text{minimize}} \quad & f(\boldsymbol{x}) = T(\boldsymbol{x})^{\beta} \times R(\boldsymbol{x})^{1-\beta} \\ \text{s.t.} \quad & T(\boldsymbol{x}) \leq T_{max}, R(\boldsymbol{x}) \leq R_{max}, etc \end{aligned} \quad (1)$$

where $f(\boldsymbol{x})$ is the objective result of Spark configuration $\boldsymbol{x}$, T_{max} and R_{max} is the maximum tolerated runtime and resource used, and β is a constant used to control the tuning goal with $\beta \in [0, 1]$. Resource function $R(\boldsymbol{x})$ indicates the resource used during the job execution using configuration $\boldsymbol{x}$ in terms of CPU cores and memory, and thus can be obtained directly based on the values of resource-related parameters (e.g., `spark.executor.instances`, `spark.executor.cores`, `spark.executor.memory`).

Generalized Objective with Different β. Users can customize their tuning objectives and reflect their application-specific tendency by setting different values of β. When β is set to 1, the target is to find the configuration with the lowest runtime; when β equals 0, the objective becomes minimizing the unit of resource used. Particularly, when $\beta = 0.5$, the objective is equivalent to optimizing

the execution cost by ignoring the square root. The execution cost is the product of runtime $T(\boldsymbol{x})$ and the price per unit time $P(\boldsymbol{x})$. For simplicity, we utilize the amount of resource $R(\boldsymbol{x})$ to approximate $P(\boldsymbol{x})$ because $P(\boldsymbol{x})$ is usually proportional to $R(\boldsymbol{x})$ and is hard to determine. If $P(\boldsymbol{x})$ can be accessed easily, users can use $P(\boldsymbol{x})$ directly in the definition of the objective function, instead of $R(\boldsymbol{x})$. Besides, the other choices of β could express users' objective tendencies. For instance, $\beta = 0.7$ indicates that the user pays more attention to the decrease in runtime while minimizing the execution cost.

Application Requirements. The generalized formulation also supports different constraints, which is essential to ensure the quality and safety of online Spark jobs. As shown in Eq. 1, users can specify the requirements about runtime, the amount of resource, etc. In addition, this framework allows users to customize different requirements for their tuning tasks. For example, the usage of CPU cores should be lower than a certain threshold.

Knowing $T(\boldsymbol{x})$ under all configurations would make it straightforward to solve Eq. 1, but it is infeasible because all candidates need to be evaluated. Instead, we use the Bayesian optimization (BO) based method to directly search for an approximate solution of Eq. 1 with significantly fewer online job executions.

3.3 Bayesian Optimization-based Framework

Bayesian optimization (BO) [9, 36, 51, 63] is a framework to solve black-box problems where the objective can only be observed via evaluation. The main advantages of BO are that it is robust to noise and can estimate uncertainty to balance exploration and exploitation. A typical loop in vanilla BO contains the following

four steps: (1) it fits a *surrogate model M* based on observed configurations $D = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_{n-1}, y_{n-1})\}$, where $\boldsymbol{x}_i$ is the ith configuration and y_i is the corresponding performance; (2) it chooses the next promising configuration that maximizes the *acquisition function* $\boldsymbol{x}_n = \arg\max_{\boldsymbol{x} \in \chi} \alpha(\boldsymbol{x}; M)$, where the acquisition function is designed to balance the exploration and exploitation; (3) it evaluates the chosen configuration $\boldsymbol{x}_n$ to obtain its performance $y_n = f(\boldsymbol{x}_n) + \epsilon$ with noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$; (4) add the pair $(\boldsymbol{x}_n, y_n)$ to observations so that $D = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_n, y_n)\}$. The pseudo-code is provided in Algorithm 1.

Surrogate. To model the objective function, different from machine learning algorithms used in offline approaches [7, 79], which incorporate additional algorithm hyperparameters, we choose Gaussian process (GP) [61] as the surrogate model because it is hyperparameter-free and provides closed-form inference. GP is a flexible class of models for specifying prior distributions over objective function: $f : \chi \rightarrow R$. The convenient properties of the Gaussian distribution allow us to compute marginal and conditional means and variances in closed form. Given an unseen configuration $\boldsymbol{x}$, the predictive mean and covariance are expressed as follows,

$$\mu(\boldsymbol{x}) = K(X, \boldsymbol{x})(K(X, X) + \tau^2 I)^{-1} Y,$$
$$\sigma^2(\boldsymbol{x}) = K(\boldsymbol{x}, \boldsymbol{x}) + \tau^2 I - (K(X, X) + \tau^2 I)^{-1} K(X, \boldsymbol{x}), \quad (2)$$

where K is the covariance matrix, X are the observed configuration vectors and τ^2 is the level of white noise.

Acquisition Function. To estimate the performance improvement of the unseen configuration, we apply the Expected Improvement (EI) [39] function. Given the marginal predictive mean $\mu(\boldsymbol{x})$ and variance $\sigma^2(\boldsymbol{x})$ by the surrogate model, the EI function is defined as the expected improvement over the best performance found,

$$EI(\boldsymbol{x}) = \int_{-\infty}^{\infty} \max(y^* - y, 0) p_M(y|\boldsymbol{x}) dy$$
$$= \sigma(\boldsymbol{x}) \left(\gamma(\boldsymbol{x}) \Phi(\gamma(\boldsymbol{x})) + \phi(\gamma(\boldsymbol{x})) \right), \quad (3)$$

where $\gamma(\boldsymbol{x}) = \frac{y^* - \mu(\boldsymbol{x})}{\sigma(\boldsymbol{x})}$, y^* is the best performance observed so far. The second line of Equation 3 is the closed form of EI under Gaussian Process, $\Phi(\cdot)$ and $\phi(\cdot)$ are standard normal cumulative distribution function and probability density function, respectively. While vanilla EI is designed for problems without constraints, we will introduce how we deal with constraints in Section 4.2.

Dynamic Workload Support. As aforementioned, the workload may change during the online tuning process, particularly the data size. Since the same configuration for a workload may achieve different results with different input data sizes, the change in data size could affect the tuning result. To accommodate this, we take the data size along with configuration into consideration and model the objective value with the Gaussian Process. Concretely, the configuration $\boldsymbol{x}_i$ contains the configuration values and the dataset size, which is represented as $\bar{x}_i = \{x_i^1, ..., x_i^N, ds_i\}$, where x_i^j is the value for the ith configuration and ds_i is the data size of that run. After evaluating n Spark configurations, we elaborate the objective $f(\bar{X})$ by Gaussian distribution as follows,

$$f(\bar{X}) \sim GP(0, K(\bar{X}, \bar{X}')), \quad (4)$$

where K denotes the covariance matrix, and $\bar{X}$ is a $n*(N+1)$ matrix where N is the number of Spark parameters. $\bar{X}$ and $\bar{X}'$ are matrices of the same size. We propose to use mixed kernels to implement this.

Concretely, we apply the Matern kernel for numerical parameters, the Hamming kernel for categorical parameters, and the SE kernel to deal with data size. However, due to data privacy issue, input data size is not always accessible in production tasks. In this case, besides Spark configurations, we train a datasize-aware surrogate by also taking the hour of the day or the current day of the week as input to characterize the periodic change of data. Therefore, the BO framework can adapt to dynamic workloads in online tuning.

Initial configurations. To estimate the performance in a wide area of configuration space, we sample several configurations using low-discrepancy sequences [67] to initialize the observations D (Line 1 in Algo. 1). In addition, to accelerate the convergence in the beginning, the meta-learning module can suggest the initial configurations based on task similarity, instead of using the low-discrepancy sequences. We will describe the detail in Section 5.2.

Stopping & Restarting Criterion. When the tuning budget exhausts or the stopping criterion is met, we stop the current tuning process and return the best configuration found so far for each upcoming config request. When the expected improvement in Eq. 3 is less than a threshold (e.g. 10%), the stopping criterion is activated. This prevents our framework from struggling to make small improvements. In addition, the controller compares the difference between the expected performance and the actual execution result of the workload over runs. If a continuous degradation is detected, re-tuning becomes necessary, and our framework will restart the tuning process, where meta-learning is utilized to extract knowledge from the previous runhistory to speed up the optimization.

4 EFFICIENT & SAFE CONFIG ACQUISITION

In this section, we present the methods in configuration generator for dealing with the slow convergence and safe exploration.

4.1 Sub-space Generation

As aforementioned, the complete configuration space is huge, which greatly limits the efficiency of the configuration search. An intuitive idea is to use a smaller configuration sub-space that includes the most influential parameters, instead of the original huge space. Tuning the sensitive parameters could bring significant performance improvement, and avoid wasting efforts in tuning those parameters that have little influence on performance. Furthermore, the size of the sub-space is crucial: a large sub-space leads to slow convergence of configuration search but has a high probability of finding the optimal configuration; BO tuning framework could converge to a local configuration quickly over a small sub-space, but this sub-space may not include the optimal or near-optimal configurations. Therefore, when dealing with the high-dimensional space, we need to answer two questions: (1) how to measure the importance of Spark parameters, and (2) how to decide/adjust the size of the sub-space adaptively to pursue efficiency and effectiveness simultaneously.

Parameter Importance & Sub-space. To obtain the sub-space, we should first measure the importance of each parameter. While existing methods only consider the influence between each parameter and performance via Spearman Correlation Coefficient or weights of a learning model [24, 76], we consider the importance of both single parameters and of interactions between parameters. To this end, we adopt FANOVA [35] to assess the importance of parameters.

It is a linear-time algorithm for computing marginals of random forest predictions and then leverages these predictions within a functional ANOVA framework to quantify the importance of both single parameters and of interactions between parameters. Given the tuning history from a task, our parameter importance analysis module could rank the parameters according to their importance.

During optimization, we can get the importance score of parameters based on its tuning history for each task and obtain the final importance scores by averaging the scores from those tasks. Based on this result, we define the sub-space as:

$$\Lambda_{sub} = \Lambda^1 \times ... \times \Lambda^K, \tag{5}$$

where Λ^i is the domain of parameter x^i with the ith largest importance score, and K is the size of the sub-space. Note that, when starting tuning from zero, there is no tuning history available to obtain the parameter ranking based on the importance scores. We start with an initial parameter ranking suggested by experts. Once new tuning history arrives, we will continuously update the importance score for each parameter based on FANOVA.

Evolution Strategy of Sub-space. Different from existing methods that use a fixed sub-space [76], we propose to automatically adjust the size of the sub-space, i.e., K. At the beginning of the tuning process, we initialize the size of sub-space K to a small constant $K \leftarrow K_{init}$, and then the value of K will be updated in a certain period, i.e., N_{space} iterations. The range of space size is $[K_{min}, K_{max}]$, where K_{max} is the number of Spark parameters in the entire space Λ_{cs}. On one hand, a sub-space should be sufficiently large to contain good configurations. On the other hand, it should be small enough to ensure that BO is accurate and efficient within this sub-space. Therefore, the evolution of K is crucial.

To dynamically control the space size, similar to TuRBO [23], we adopt an intuitive design that we expand the sub-space when the BO optimizer "makes progress", i.e., it finds better configurations in the current sub-space, and shrinks it when the optimizer appears stuck. We define a "success" as a configuration that improves over the best configuration found, and a "failure" if a configuration that fails. After τ_{sucs} consecutive successes, we increase the size of the sub-space to $K \leftarrow \min(K_{max}, K + 2)$. After τ_{fail} consecutive failures, we decrease the size of the sub-space to $K \leftarrow \max(K_{min}, K - 2)$. Once the size of sub-space is changed, the counters for the success and failure events are set to zero. Based on this mechanism, the sub-space can be adjusted adaptively to achieve high efficiency while getting good convergence results. In our implementation, $\tau_{sucs} = 3$, $\tau_{fail} = 5$, $K_{min} = 4$ and $K_{init} = 10$, respectively.

4.2 Safe Exploration and Exploitation

To deal with constraints as we introduced in Eq. 1, we apply the EI with constraints (EIC) [26] as the acquisition function. Different from the traditional EI [39] in Eq. 3, EIC takes the probability of satisfying the constraints into consideration. Concretely, EIC for the generalized tuning in Eq. 1 is defined as follows,

$$EIC(\boldsymbol{x}) = Pr[T(\boldsymbol{x}) \leq T_{max}] \cdot EI(\boldsymbol{x}), \tag{6}$$

where $Pr[T(\boldsymbol{x}) \leq T_{max}]$ refers to the probability that satisfies the runtime constraint. To predict the runtime of an unseen configuration, we also fit a surrogate (i.e., GP) on runtime metrics. To

customize other constraints, e.g., restricting resources, we can implement it by multiplying the term in Eq. 6 by $Pr[R(\boldsymbol{x}) \leq R_{max}]$. Without loss of generality, we take the runtime constraint as an example. The probability can be computed as,

$$Pr[T(\boldsymbol{x}) \leq T_{max}] = \int_{-\infty}^{T_{max}} p(T(\boldsymbol{x})|\boldsymbol{x})dT(\boldsymbol{x}), \tag{7}$$

which can be obtained by using the posterior prediction from GP.

In online tuning scenarios, evaluating a configuration that violates the constraints will inevitably downgrade the job performance. To explicitly ensure the safety of job execution, we further propose safe exploration and exploitation to select the next candidate. Given the predictive mean $\mu_t^T(\boldsymbol{x})$ and variance $\sigma_t^{T^2}(\boldsymbol{x})$ from the runtime surrogate built at the tth iteration, we define the upper bound as:

$$u_t^T(\boldsymbol{x}) = \mu_t^T(\boldsymbol{x}) + \gamma\sigma_t^T(\boldsymbol{x}), \tag{8}$$

where γ is a constant that controls the bound and $\gamma \in (0, 1]$. We maintain a safe region so that the configurations in that region are expected to satisfy the constraints in the worst case, which is theoretically analyzed in previous work [69]. The safe region for runtime at the tth iteration is defined as $S_t^T = \left\{ \boldsymbol{x} \in \Lambda_{sub} \mid u_t^T(\boldsymbol{x}) \leq T_{max} \right\}$. For multiple constraints, the final safe region is the intersection of single safe regions. The main process for the configuration generator to produce a new configuration is: (1) build the final safe region S, and (2) choose the next configuration by solving the EIC over S.

4.3 Approximate Gradient Descent

In Spark tuning, the objective in Eq. 1 is treated as a black-box function of input parameters with no available derivative information. However, some parts of the objective functions are not always black-box. Though the relationship between runtime and Spark parameters is complicated, the resource function has an analytic form, e.g., $R(\boldsymbol{x}) = \#cpu\_vcores(\boldsymbol{x}) + c \cdot \#mem(\boldsymbol{x})$, which is directly determined by the values of parameters such as `spark.executor.instances`, `spark.executor.cores`, and `spark.executor.memory`. Gradient-based techniques [62] perform well in optimizing differentiable functions. Compared with BO, the derivative information of the objective function provides more precise guidance on how to choose the next input parameters based on the current configurations (i.e., exploitation signal). When observations D are sufficient to approximate the objective function $f(\boldsymbol{x})$, we can estimate the derivative information. However, since AGD emphasizes the exploitation signal only during the search process, it could be stuck in local configurations easily. To address this and integrate the benefits of both methods, we **alternately apply AGD and BO** to select the next configuration so that the combined framework balances exploration and exploitation well with a convergence guarantee provided in previous work [36]. Therefore, we develop the approximate gradient descent (AGD) technique within BO to utilize the derivative information of the objective function and further accelerate the convergence over the search space.

To apply gradient descent, the partial derivative of $f(\boldsymbol{x})$ over each numerical parameter x^i is:

$$\frac{\partial f}{\partial x^i}(\boldsymbol{x}) = \beta(\frac{T(\boldsymbol{x})}{R(\boldsymbol{x})})^{\beta-1}\frac{\partial T}{\partial x^i}(\boldsymbol{x}) + (1-\beta)(\frac{T(\boldsymbol{x})}{R(\boldsymbol{x})})^{\beta}\frac{\partial R}{\partial x^i}(\boldsymbol{x}), \tag{9}$$

where x^i is the ith Spark parameter. Since the resource function $R(\boldsymbol{x})$ is a white-box function, the partial derivative of $R(\boldsymbol{x})$, i.e.

$\partial R(\boldsymbol{x})/\partial x^i$, has an analytical form. However, the partial derivative of the runtime function $T(\boldsymbol{x})$, i.e. $\partial T(\boldsymbol{x})/\partial x^i$, cannot be calculated directly. We approximate the partial derivative $\partial T(\boldsymbol{x})/\partial x^i$ by:

$$\frac{\partial T}{\partial x^i}(\boldsymbol{x}) \approx \frac{T(\boldsymbol{x} + \boldsymbol{\epsilon}_i) - T(\boldsymbol{x} - \boldsymbol{\epsilon}_i)}{2|\boldsymbol{\epsilon}_i|}, \tag{10}$$

where all dimensions except the ith element in $\boldsymbol{\epsilon}_i$ are zero. The values of $T(\boldsymbol{x} + \boldsymbol{\epsilon}_i)$ and $T(\boldsymbol{x} - \boldsymbol{\epsilon}_i)$ are predicted by the surrogate for runtime. Then, the value of each parameter x^i is updated by:

$$x^i \leftarrow x^i - \eta * \frac{\partial f}{\partial x^i}(\boldsymbol{x}), \tag{11}$$

where η is the learning rate with $\eta = 0.001$. During the tuning procedure, AGD is integrated into BO. Every $N_{AGD} = 5$ BO iterations, instead of BO, the next configuration $\boldsymbol{x}$ is generated by conducting AGD based on the best configuration found so far.

Algorithm Summary in Generator. Algorithm 2 gives the pseudo-code in configuration generation. During each iteration of BO, we train multiple surrogates for objective, runtime, other constraints and resource on current observations (Line 1). Every N_{AGD} iterations, we choose the next configuration via AGD (Line 4). Else, based on the constraint surrogates, we obtain the candidate region by intersecting the sub-space and safe regions for each constraint (Lines 6-7). Then, we choose the configuration that maximizes the EIC acquisition function over the candidate region (Line 8). Finally, the suggested configuration is returned (Line 10).

5 META-LEARNING BASED ACCELERATION

In this section, we describe the design of the meta-knowledge learner, which utilizes meta-learning based techniques to accelerate the configuration search in BO. We first introduce the method to learn the similarity between tuning tasks using the meta-features from workloads. Then, based on similarity learning, we present the meta-learning methods for warm-starting and surrogate modeling.

5.1 Task Characterization & Similarity Learning

Inspired by previous work [60], we extract the feature vector of a tuning task (i.e., meta-features) from `SparkEventLog`, in which Spark event information is logged during execution. The meta-features summarize information in two levels: stage and task. Stage information contains the Spark actions and transformations used in the Stage, which shows core Spark function calls made during the job execution. Task information describes whether, overall, the task was read or write-intensive, CPU intensive, etc. The resulting meta-features include a total of 75 features, where 11 are retrieved from Stage information and 64 are retrieved from Task information [60].

A straightforward method that measures the similarity is to compute the Euclidean distance between the two meta-features of corresponding tasks. However, the type and scale of each meta-feature are heterogeneous, which greatly decreases the effectiveness of Euclidean distance. Besides, each meta-feature poses a diverse impact on similarity learning. To address these issues, we propose to use a supervised learning method (i.e., regression model) to learn the similarity given two tasks. Concretely, given the meta-features of two input tasks: $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, the regression model $\mathcal{M}_{reg}$: $(\boldsymbol{v}_1, \boldsymbol{v}_2) \mapsto d$ predicts their distance $d \in [0, 1]$. A smaller distance d indicates that the two tasks are more similar to each other.

Algorithm 2: Pseudo code for configuration generation.

Input: Evaluation observations D, constants, e.g., N_{AGD}, η, etc.
Output: the suggested configuration $\boldsymbol{x}_i$ for the ith iteration.
1: Trains surrogates (e.g., M^f, M^T, M^R) on current observations D.
2: **if** $|D| + 1 \bmod N_{AGD} = 0$ **then**
3:     Set current best configuration: $\boldsymbol{x}_i \leftarrow$ get_best_config(D).
4:     Choose the next configuration via AGD: $x^i \leftarrow x^i - \eta * \frac{\partial f}{\partial x^i}(\boldsymbol{x})$.
5: **else**
6:     Update the size of the sub-space Λ_{sub} based on runhistory D.
7:     Build the safe region for all constraints: $S_i = S_i^T \cap S_i^R \cap \Lambda_{sub}$.
8:     Choose the configuration by solving $\boldsymbol{x}_i = \arg\max_{\boldsymbol{x} \in S_i} EIC(\boldsymbol{x})$.
9: **end if**
10: **return** the Spark configuration $\boldsymbol{x}_i$.

To train such a regression model, we prepare the training data as follows. We collect a wide range of Spark jobs $T_{1,\ldots,K}$ and the corresponding tuning history $H_{1,\ldots,K}$, where the tuning history H_i of the ith task consists of the pairs of configuration and its performance. As aforementioned, the meta-feature for each tuning task can be obtained by parsing the information in event log [60]; next, we need to define the distance metric. In the context of Spark tuning, we focus on the partial orderings over configuration performance. To this end, we measure the distance by calculating the number of discordant pairs of predictions (i.e., pair-wise ranking) using the **negative** Kendall-tau coefficient. That is, given two configurations $\boldsymbol{x}_k$ and $\boldsymbol{x}_l$, and surrogates model M^i and M^j on the ith and j^{th} task, the pair $(\boldsymbol{x}_k, \boldsymbol{x}_l)$ is discordant if the sort order of $(M^i(\boldsymbol{x}_k), M^i(\boldsymbol{x}_l))$ and $(M^j(\boldsymbol{x}_k), M^j(\boldsymbol{x}_l))$ disagrees. Then, the distance is defined as **the ratio of discordant pairs**, which is $Dist(M^i, M^j) = \frac{1 - \tau^{D_{rand}}(M^i, M^j)}{2}$ where $\tau^{D_{rand}}(M^i, M^j)$ is the Kendall-tau coefficient of two vectors by applying surrogates M^i and M^j on randomly sampled configurations D_{rand}. While the range of Kendall-tau coefficient is $[-1, 1]$, we scale the range of distance $Dist(M^i, M^j)$ to $[0, 1]$. Finally, we obtain the all basic ingredients to learn the meta-learner $\mathcal{M}_{reg}$ that could predict the distance between two tasks. In our implementation, we train a LightGBM [41] regressor to obtain $\mathcal{M}_{reg}$.

5.2 Warm-starting & Surrogate Improvement

Based on similarity learning, we could speed up the configuration search in two aspects: initial design and surrogate modeling.
Initial design with warm-starting. In vanilla BO, the search process starts from scratch. That is, it initializes observations D with several evaluations of random configurations (in Algorithm 1). By leveraging similarity across tasks, given a new tuning task, we can obtain similarity results between the new task and previous tasks. We rank the previous tasks using the predictions of meta-learner $\mathcal{M}_{reg}$, and choose the top-3 most similar tasks. Then, we select the best Spark configuration found in these top-3 tasks, and set them as the initial configurations before starting the main BO loop in Algorithm 1 (Line 1). In this way, our framework could achieve a better performance in the beginning. In addition, we also can suggest the sub-space for a new tuning task using task similarity.
Surrogate modeling with meta-learning. Given scarce observations at the beginning of the optimization, BO can not build an accurate surrogate to guide the search. In this case, BO fails to converge to optimal configuration quickly. To overcome this

issue, we leverage the tuning history from previous similar tasks to obtain an accurate surrogate. Therefore, we propose to build a meta-learning surrogate ensemble M_{meta}. The prediction of M_{meta} at configuration x is given by $y \sim \mathcal{N}(\mu_{\text{meta}}(x), \sigma_{\text{meta}}^2(x))$:

$$\mu_{meta}(x) = \sum_i w_i \mu_i(x), \quad \sigma_{meta}^2(x) = \sum_i w_i^2 \sigma_i^2(x), \qquad (12)$$

where μ_i and σ_i^2 are the predictive mean and variance from base surrogate M^i. The weight of base surrogate w_i reflects the similarity between the previous and current task. Here we set the weight to $w_i = 1 - Dist(M^i, M^t)$ and then normalize the weights with $\sum_i w_i = 1$. In addition, M_{meta} also includes a surrogate M^t trained on the current task, and the weight of M^t is determined by cross-validation strategy [25]. By integrating useful information from both the current task and previous tasks, M^{meta} could offer more accurate modeling for the objective function, thus further accelerating the convergence of the current tuning task.

6 EXPERIMENTS AND RESULTS

To evaluate our framework, in this section, we list three insights we want to investigate: 1) **Practicality** - Our framework achieves significant cost reductions compared with human experts when tuning 25K real-world Spark tasks in Tencent. 2) **Generality** - Our framework supports different tuning objectives (runtime and cost) and Spark task types (MR and SQL), and consistently outperforms state-of-the-art Spark tuning approaches on various standard benchmarks; 3) **Efficiency** - Our framework equipped with the three techniques greatly accelerates the tuning process and requires only a few trials to find near-optimal configuration.

6.1 Experiment Setup

Benchmark Programs: In addition to the real-world production tasks (described in Section 6.2), we also use programs from the well-known big data benchmark suite HiBench [34] in our experiments. HiBench contains various Spark tasks, including machine learning, graph, websearch, etc. We choose 6 representative tasks: Bayes, KMeans, NWeight, WordCount, PageRank, and TeraSort. A larger set with 16 tasks is used in the meta-learning experiment.

Environment: When tuning the in-production Spark tasks, the online job executions are conducted on the resource group of our customers, which is allocated by the Tencent data platform. For example, one of our customers owns a resource group with 100 computing units, where each unit corresponds to 20 Intel(R) Xeon(R) Platinum 8255C CPU cores with a base frequency of 2.50GHz and 50 GB memory. The HiBench tasks are executed on a small x86 cluster with four nodes. Each server is equipped with 2 AMD EPYC 7K62 2.80GHz 48-core processors and 512GB PC4 memory. In this environment, we use Spark 3.0 as our computing framework.

Spark Parameters: We use the same Spark configuration parameters as used in Tuneful [24], which contains 30 parameters that significantly influence the application performance. The value ranges of the parameters are set differently depending on the cluster size.

Compared Methods. When tuning Hibench tasks, we compare our framework with the following approaches: (1) *Random Search* [8] generates a random configuration at each iteration. (2) *RFHOC* [7]

trains several random forests for each task, and utilizes the generated models along with a genetic algorithm to explore the configuration space. (3) *DAC* [79] is a datasize-aware auto-tuning approach, which utilizes hierarchical regression tree models and genetic algorithm to efficiently identify the near-optimal configurations. (4) *CherryPick* [2] uses Bayesian optimization (BO) to find the optimal configuration for Spark applications. The model aims at minimizing the user cost and subjects to a runtime threshold. (5) *Tuneful* [24] utilizes BO to tune the configuration of in-memory cluster computing systems in an online manner. (6) *LOCAT* [76] is also a BO-based online approach to conduct auto-tuning for Spark SQL.

Objectives. To evaluate the generality, we consider two tuning objectives in the experiments: (1) The *Runtime* of Spark job, where $\beta = 1$ in Eq. 1. (2) The execution *Cost* of Spark job, where $\beta = 0.5$ in Eq. 1. Note that the scale of the cost depends on the definition of runtime (e.g., in seconds or hours) and resources in different experiments. The tuning objective of RFHOC, DAC, Tuneful, and LOCAT, is runtime. In the cost experiment, we modify some modules in their implementations to support cost minimization.

Metrics & Settings. We use the following three metrics: (1) The *Speedup* of execution time of the best-found configuration relative to the one from random search. (2) The *Cost* at the ith iteration, i.e., the execution cost of the ith tried configuration as defined in Eq. 1. Note that, the **Min Cost** refers to the execution cost of the best configuration found so far. (3) The *Cost Reduction* relative to a reference method (e.g., random search or default configuration), which is defined as $\frac{Cost_{ref} - Cost}{Cost_{ref}}$. The hyperparameters in our framework mentioned in previous sections are obtained via sensitivity analysis, and we apply the same settings in our implementation. The BO component is implemented using the OpenBox library [38]. All experiments running on HiBench are repeated 10 times with different random seeds and the average metrics are reported.

6.2 Results on In-production Spark Tasks

(Implementation & Application in Tencent) We have implemented the proposed framework as an independent tuning service deployed on Tencent Cloud, and this service has been applied to Tencent's internal data platform. It continuously tunes the recurring Spark jobs, which are scheduled and executed every hour, day, etc. The data platform also provides web UIs for users to start online tuning and observe the tuning results conveniently (See web pages in Figure 3). The tuning objective is set to execution cost with $\beta = 0.5$ in Eq. 1, and the constraints are set to twice the metrics of the manual configurations. Our framework optimizes the tuning objective (i.e., execution cost), which is positively correlated with the actual cost (i.e., **both** memory and CPU usage). In order to measure the actual cost directly and quantitatively, we also analyze the performance of configurations using the two metrics: memory usage (GB·hour), CPU usage (core·hour).

(Large-Scale Tuning Results) To demonstrate the practicability and efficiency of our service in real-world scenarios. We utilize it to tune around 25K in-production Spark tasks from Tencent's business scenarios, including advertisement, marketing, and social networking. These tasks are executed every hour and produce essential statistics for downstream tasks. Before auto-tuning, the configuration used for each Spark task is manually tuned by big data

(a) Memory usage reduction   (b) CPU usage reduction   (c) Average execution cost reduction
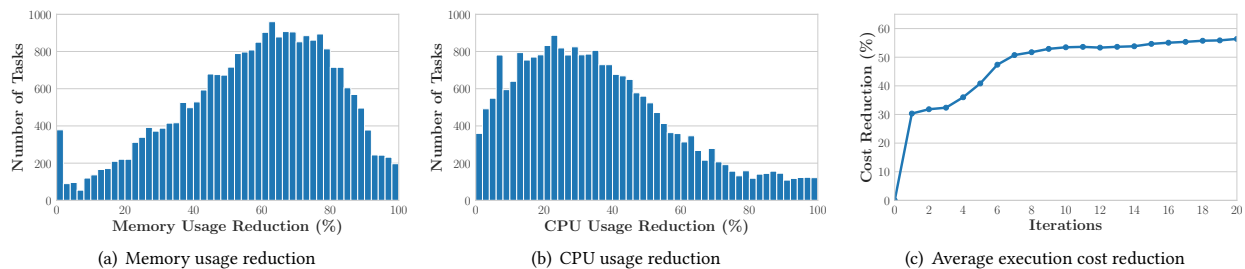
Figure 2: The reduction results of memory usage, CPU usage, and objective value on 25K Spark tasks in Tencent.

Table 2: Detailed comparison between manual and tuned configurations on eight in-production tasks related to advertisement.

| Task	Method	Memory_usage	CPU_usage	Runtime(s)	Execution cost	Executor.instances	Executor.cores	Executor.memory(GB)	#Iteration
Spark: Feature Extraction	Manual	1776.86	834.18	6768.99	208125.83	300	2	8	-
	Ours	811.12	273.82	5316.64	55718.35	183	3	1	12
Spark: User-Traffic Distrib.	Manual	3865.39	966.35	5543.30	170015.02	256	2	8	-
	Ours	1095	1117.16	4401.42	131699.25	300	3	3	14
Spark: DAU Analysis	Manual	2513.62	628.07	1220.81	67727.11	500	4	16	-
	Ours	495.20	248.31	1270.04	28008.57	728	1	2	8
Spark: Log Processing	Manual	2462.04	1076.11	4068.47	662950.82	656	4	9	-
	Ours	885.90	444.18	2219.14	49205.71	732	1	2	9
Spark SQL: Data Selection	Manual	0.6731	0.6770	48.29	243.53	16	6	6	-
	Ours	0.0189	0.0135	32.26	30.78	1	1	1	10
Spark SQL: Skew Detection	Manual	9.4702	0.9788	227.03	667.60	20	2	20	-
	Ours	0.4812	0.2883	213.08	147.82	1	1	1	7
Spark SQL: Feature Calculation	Manual	1374.84	0.93	518.34	321.78	3	2	1	-
	Ours	277.57	0.40	497.00	210.59	2	1	1	13
Spark SQL: Data Preprossing	Manual	34.7431	0.0222	23.1300	210.52	3	2	6	-
	Ours	10.4521	0.0102	19.9520	103.13	1	2	1	6
Avg Reduction on 8 tasks	-	**-76.52%**	**-56.29%**	**-17.58%**	**-62.22%**	-	-	-	**9.88**
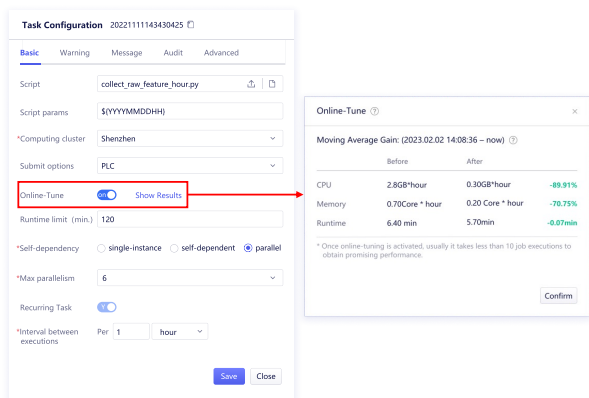
Figure 3: Web UIs of Online-Tune provided by Tencent's internal data platform.

engineers (abbr. manual). After activating online tuning, each task is tuned once an hour, and the total budget is set to 20 hours (i.e., 20 trials or iterations). Figures 2(a) and 2(b) show the number of tuning tasks with different cost reduction results after tuning. We observed that, the average memory and CPU cost reduction of the tuned configuration compared with manual configuration on 25K tasks are 57.00% and 34.93%, respectively. Moreover, 66.49% of the tasks get a memory usage reduction of over 50%, and 64.70% of the tasks achieve a CPU usage reduction of over 25%. In Figure 2(c), we plot the average execution cost reduction ratio of the best configuration

Table 3: Averaged cost reduction (%) of job execution metrics of under-tuning (under) within 20 iterations, and post-tuning (post) compared with pre-tuning (pre) on 25K real-world tasks in Tencent. '-' indicates the increase in metrics.

Metric	Cost Reduction(under vs. pre)	Cost Reduction(post vs. pre)
Memory usage	2.28%	**57.00%**
CPU usage	-5.82%	**34.93%**
Runtime	1.63%	**10.72%**

found during optimization on the 25K tasks. The tuning objective (i.e., execution cost) reduces significantly during early iterations, where we observe a reduction of 52.44% within only 9 iterations. Note that, our warm-starting technique with meta-learning used in the first 3 iterations leads to a huge improvement. This demonstrates the high efficiency of our service.

(Tuning Overhead Analysis) The tuning process is conducted only once, while the Spark job is repeatedly executed for a long time (e.g., months or years). The benefits of saving costs during each execution would accumulate, which is a large gain as time proceeds. We also demonstrate the **gains** and **overheads** of online tuning on 25K tasks in Table 3. By comparing the post-tuning results (post) with manual pre-tuning results (pre), we observe significant gains on the three metrics (post vs. pre) for each job execution on average. For example, the tuned configuration reduces 57.0% of the memory usage compared with the manual one. In addition to the gains, we also compute the additional overhead (i.e., execution cost) by comparing the average metrics of each job execution during the
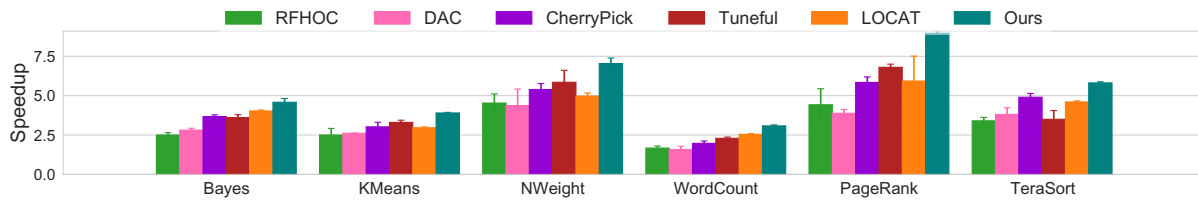
Figure 4: Speedup of compared methods relative to random search on 6 HiBench tasks.
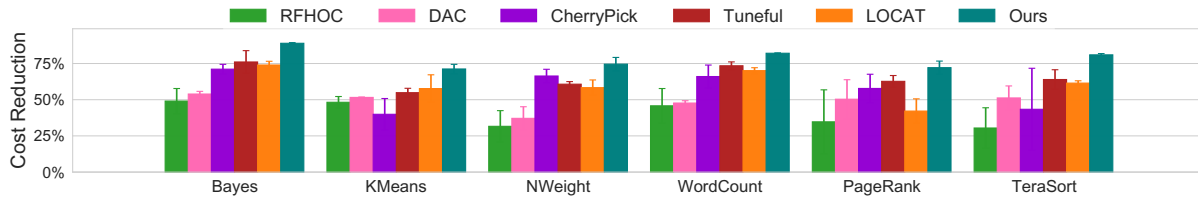

Figure 5: Cost reduction of compared methods relative to random search on 6 HiBench tasks.

tuning process (under) with those in pre-tuning. Since our framework needs to try configurations with different performance before finding a good one, the reduction ratios are lower during the tuning process (under vs. pre). But after 20 iterations, the framework stops tuning and directly applies the best-found configuration so that the additional tuning overhead will be amortized quickly as the Spark tasks execute continuously. Based on the results in Table 3, our service needs **no more than 4 extra executions to amortize the CPU usage overhead on average**. Besides, the average metrics of the suggested configurations satisfy the constraints during tuning. **(Analysis on Eight Specific Tasks)** To further demonstrate the generality and analyze the tuning behavior of our framework, we evaluate it on eight different online Spark tasks in detail and show the results in Table 2. Each task is collected from the advertisement business in Tencent, where the first four traditional Spark tasks are executed once a day and the other four Spark SQL tasks are executed once an hour (generality on task types). In the tuning process, all Spark tasks are tuned by trying different configurations in the production environment. We compare our framework with the manual configuration. The tuning objective and constraints keep the same as in the previous 25K tasks. Compared with manually tuned configurations, our framework reduces the execution cost by 62.22% on average. Meanwhile, we observe significant decreases **along with** the objective on the two metrics (i.e., average memory usage and CPU usage), which are 76.52% and 56.29%, respectively. Notably, we also report the number of iterations to achieve the tuned results, and the average iteration is **less than 10**, which demonstrates the efficiency of our framework. In addition to the tuning results, we also provide the parameter values related to the Spark executor for reference. For example, in the first task – feature extraction, compared with the manual configuration, spark.executor.instances (i.e., the number of executor instances) decreases from 300 to 183, spark.executor.cores increase from 2 to 3, and spark.executor.memory decreases from 8GB to 1GB.

6.3 Results on Public Benchmarks

To further investigate the generality, we conduct end-to-end experiments on 6 HiBench tasks on the small cluster. All methods are tested using two tuning objectives: (1) runtime with $\beta = 1$, and (2) cost by setting $\beta = 0.5$ in Eq. 1. The overall budget is set to 30 iterations. We also set a runtime constraint in the experiment, where the threshold is twice the runtime of the default configurations. This constraint ensures safety in online tuning so that the tuning procedure will not greatly affect online executions. For each method, we report the best observed objective value within 30 trials.

Figure 4 shows the speedup of all baselines relative to random search when the objective is runtime. We observe that: (1) The ML-based approaches, RFHOC and DAC, achieve a relatively lower speedup compared with Bayesian optimization-based methods. ML models often need a large number of training samples, and 30 iterations are not sufficient to identify a promising configuration in the search space. (2) The Bayesian optimization-based methods, CherryPick, Tuneful, LOCAT, and ours, get a better result under the limited budget. However, CherryPick does not reduce the dimension of search space when training the surrogate model, thus it cannot handle the large Spark search space well. Tuneful and LOCAT use different techniques to select important parameters in the optimization process, but require 10 to 20 executions before shrinking the search space. Across the six tasks, their rankings are unstable. (3) Our framework achieves the best and consistent speedup among all compared methods. Concretely, it leads to 3.08x-8.96x average speedups relative to random search, where the second best baselines in each task achieve only 2.54x-6.80x speedups.

In Figure 5, we also present the relative execution cost reduction of all methods compared with random search. Compared with runtime, the execution cost is a more difficult objective to optimize. Concretely, our framework achieves a cost reduction of 71.22-88.97% relative to random search and obtains a cost reduction of 38.43% and 45.20% on average compared with competitive baselines Tuneful and LOCAT, respectively.

Table 4: Execution cost of the top-3 configurations found by our warm-starting method.

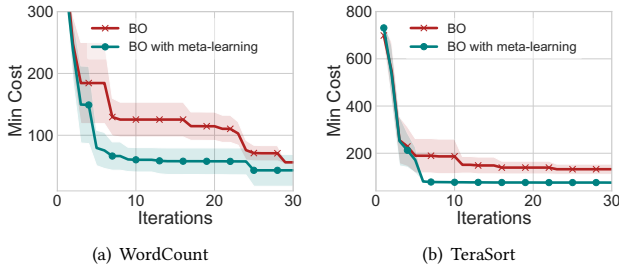| Target Task	Source Task	Default	Manual	Top1	Top2	Top3
TeraSort	Sort	844.70	91.3	54.51	40.66	43.77
TeraSort	WordCount	835.00	131.60	97.48	113.30	104.71
LR	PageRank	1431.21	245.90	183.35	333.39	214.73
KMeans	SVD	400.92	232.33	136.20	166.41	171.57

(a) WordCount   (b) TeraSort

Figure 6: Results of tuning KMeans and TeraSort using BO with and without the meta-learning surrogate.

Table 5: Top-10 Spark parameters ordered by importance.

#	Parameter Name	Importance Score (mean ± std)
1	spark.executor.instances	0.3788 ± 0.1965
2	spark.executor.memory	0.1501 ± 0.1365
3	spark.memory.storageFraction	0.0469 ± 0.0400
4	spark.default.parallelism	0.0366 ± 0.0530
5	spark.memory.fraction	0.0345 ± 0.0360
6	spark.executor.cores	0.0236 ± 0.0618
7	spark.io.compression.codec	0.0199 ± 0.0290
8	spark.shuffle.file.buffer	0.0146 ± 0.0187
9	spark.shuffle.compress	0.0138 ± 0.0142
10	spark.serializer	0.0083 ± 0.0099

6.4 Meta-learning Experiments

We conduct two experiments to evaluate the efficiency of meta-learning from the following two aspects:

Warm-starting. Table 4 shows the evaluation cost of configurations given by the warm-starting module in our meta-knowledge learner on three target tasks. The results of default and manually tuned configurations are slightly different on TeraSort (Lines 1-2) since we report the results of two independent evaluations on a computing cluster. We observe that, by transferring the top-3 configurations from similar source tasks, the evaluation cost can be greatly reduced even in the initial 3 trials. Note that, the best configuration from similar source tasks can be a good one, but not always the best one. The third-best configuration from Sort outperforms its best one on TeraSort, which implies the essence of warm-starting by transferring multiple good configurations instead of only one. To summarize, the warm-starting module reduces the evaluation cost within three iterations by 66.03-95.19% and 25.44-55.93% relative to the default and manually-configured settings, respectively.

Acceleration with ensemble surrogate. Figure 6 shows the execution cost on Wordcount and Teresort with and without the ensemble surrogate, which carries meta-knowledge learned from related tasks to speed up the tuning process. We observe a clear reduction of the average cost over vanilla BO in the beginning 10

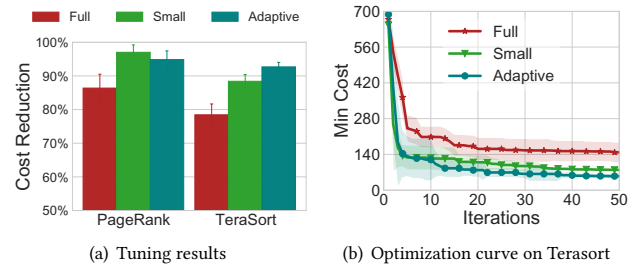(a) Tuning results   (b) Optimization curve on Terasort

Figure 7: Left: Results when tuning PageRank and TeraSort over different sub-spaces compared with default configurations. Right: Optimization curve on Terasort.

iterations. Concretely, BO with the ensemble surrogate takes at least three times fewer iterations to achieve the same average cost as vanilla BO using 30 iterations.

6.5 Ablation Studies

In this section, we present additional analysis on three methods in Section 4. Here the meta-learning module is disabled.

Sub-space Generation. Figure 7 shows the evaluation costs when tuning PageRank and TeraSort with different configuration spaces. We compare tuning with full space (30 parameters), small space (6 parameters with the highest importance in Table 5), and the proposed adaptively generated sub-space. Figure 7(a) presents the reduction ratio after 30 iterations relative to default settings while Figures 7(b) plot the average cost during the optimization. By introducing the sub-space, the average cost is consistently lower than optimization on the entire search space. Concretely, small space works well on PageRank; our adaptive method can detect this signal based on intermediate tuning results, and shrink the space size to achieve a similar result. However, although tuning converges quickly over the small space on Terasort, its final performance degenerates, where the small space dismisses the near-optimal configs. As shown in Figure 7(b), our adaptive method can converge to a better config by gradually increasing the size of the subspace.

Safe Exploration and Exploitation. The empirical results on six Hibench tasks show that the average percentage of safe configurations generated by our method is 93.00%, which is much higher than vanilla BO's – 69.67%. Figure 8 shows the objective (cost) and the constraint (runtime) during tuning WordCount and Bayes. We observe that, without the safety component, the ratio of infeasible configurations increases on both tasks. Concretely, the safety component reduces the ratio of infeasible configurations from 56% to 10% and 20% to 6% on WordCount and Bayes, respectively. In addition, the best objective found is slightly better on NWeight if we ignore the safety constraint. The reason is that, our framework conservatively restricts the search space with the safe region so that the optimal objective value may be worse than that in the entire space. However, while it is crucial to avoid infeasible configurations in online tuning tasks, it is essential to pursue safety at the expense of this insignificant objective degradation.

Approximate Gradient Descent (AGD). Figure 9 shows the relative cost reduction with and without AGD on 6 HiBench tasks. Though BO with AGD shows a slight performance degradation on
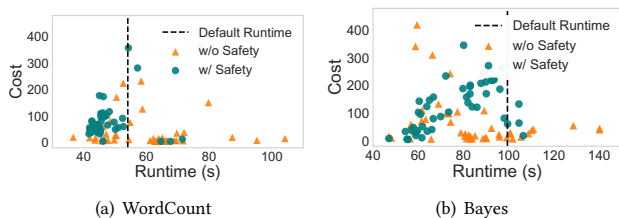
(a) WordCount

(b) Bayes

Figure 8: Analysis of safe exploration. Each point refers to the runtime (x-axis) and execution cost (y-axis) of each configuration during optimization. Configurations (circles or triangles) are infeasible on the right side of the dashed line.
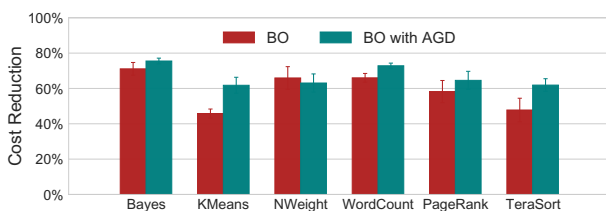

Figure 9: Tuning results when using approximate gradient descent relative to random search.

NWeight, it has a significant positive effect on other tasks. Concretely, it further reduces the cost by 7.47% on average relative to vanilla BO, which demonstrates its effectiveness.

7 RELATED WORK

In the database community, many ML-based methods have been proposed to automatically tune the knobs in DBMS [22, 24, 43, 54, 85, 86, 88], e.g., OtterTune [1, 83], Qtune [47], CDBTune [84], ResTune [87] and HUNTER [10]. When processing massive volumes of heterogeneous data, Spark tuning encounters special challenges compared with database tuning. Considering a longer job runtime or a more expensive cost, Spark tuning requires as fewer trials as possible to finish the tuning process. In addition, many researchers focus on resource management-related configuration optimization of cloud computing platforms [17, 18, 53, 89]. Here, we focus on tuning the Spark parameters efficiently.

The existing Spark tuning methods can be categorized into the following six classes [32]: (1) **Rule-based methods** [68] require an in-depth understanding of complex system mechanisms and utilize the experience of human experts, online tutorials, or tuning instructions [30] to assist users to tune Spark parameters; the tuning process is often knowledge-intensive and labor-intensive. (2) **Cost modeling methods** [13, 29, 65, 73, 75, 80] depend on a deep knowledge of system performance to build performance models. However, it is difficult for them to capture complex system behaviors. (3) **Simulation-based approaches** use modular or complete system simulation to conduct an execution with different parameter settings [3, 14, 15, 40], and further build performance models to guide the tuning process. These methods often fail to simulate complex internal dynamics in online tuning scenarios. (4) **Experiment-driven approaches** [6, 29, 59, 79, 90] execute

configuration evaluation, i.e., an experiment, repeatedly with different parameter configurations, and the configuration for each evaluation is suggested by a search algorithm. Most of them are offline methods, which incur high overhead as they require additional evaluations in an offline cluster. (5) **Machine learning (ML) approaches** [12, 31, 37, 56, 60, 74, 89] employ ML techniques to build performance models, which need a large number of training samples with high overhead; owing to the advanced design in terms of convergence efficiency, our method does not require such many expensive training samples within the online tuning paradigm. (6) **Adaptive approaches** [7, 19, 24, 27, 44, 48] tune parameters adaptively while an application is running, i.e., they can adjust the parameter settings online. They need to modify the source code of Spark to achieve adaptive updates, while it incorporates additional costs and risks. Besides, most of them focus on parameters in resource allocation (related to YARN containers) dynamically, while we concentrate on tuning all the Spark parameters.

8 CONCLUSION

In this paper, we presented a general and efficient online tuning framework for Spark, which can tune various objectives while satisfying application constraints. To solve the generalized tuning problem while preserving safety, we designed a Bayeisian optimization (BO) based solution along with a safe configuration acquisition approach. To further accelerate the configuration search, we developed three innovative techniques within BO: adaptive sub-space generation, approximate gradient descent, and meta-learning-based knowledge transfer. We implemented this framework as a tuning service and applied it to Tencent's data platform. The empirical results on extensive benchmark and production tasks demonstrate its superior performance. In the future, we plan to extend this framework to support more data analytics systems.

ACKNOWLEDGMENTS

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Conference*. ACM, 1009–1024.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[3] Danilo Ardagna, Enrico Barbierato, Eugenio Gianniti, Marco Gribaudo, Túlio BM Pinto, Ana Paula Couto da Silva, and Jussara M Almeida. 2021. Predicting the performance of big data applications on the cloud. *The Journal of Supercomputing* 77, 2 (2021), 1321–1353.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[5] Tianyi Bai, Yang Li, Yu Shen, Xinyi Zhang, Wentao Zhang, and Bin Cui. 2023. Transfer Learning for Bayesian Optimization: A Survey. *arXiv preprint arXiv:2302.05927* (2023).

[6] Liang Bao, Xin Liu, and Weizhao Chen. 2018. Learning-based automatic parameter tuning for big data analytics frameworks. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 181–190.

[7] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2015. RFHOC: A random-forest approach to

auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1470–1483.

[8] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[9] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.

[10] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *Proceedings of the 2022 International Conference on Management of Data*. 646–659.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[12] Yuxing Chen, Peter Goetsch, Mohammad A Hoque, Jiaheng Lu, and Sasu Tarkoma. 2019. d-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Transactions on Big Data* (2019).

[13] Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-effective resource provisioning for Spark workloads. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2477–2480.

[14] Leandro Batista de Almeida, Eduardo Cunha de Almeida, John Murphy, E Robson, and Anthony Ventresque. 2018. Bigdatanetsim: A simulator for data and process placement in large big data platforms. In *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 1–10.

[15] Leandro Batista de Almeida, Damien Magoni, Philip Perry, Eduardo Cunha de Almeida, John Murphy, and Anthony Ventresque. 2019. Multi-Layer-Mesh: A Novel Topology and SDN-based Path Switching for Big Data Cluster Networks. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[19] Xiaoan Ding, Yi Liu, and Depei Qian. 2015. Jellyfish: Online performance tuning with adaptive configuration and elastic container in hadoop yarn. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 831–836.

[20] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. 2012. Efficient big data processing in Hadoop MapReduce. *Proceedings of the VLDB Endowment* 5, 12 (2012), 2014–2015.

[21] Christos Doulkeridis and Kjetil Nørvåg. 2014. A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal* 23, 3 (2014), 355–380.

[22] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[23] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. 2019. Scalable global optimization via local bayesian optimization. *Advances in neural information processing systems* 32 (2019).

[24] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To tune or not to tune? in search of optimal configurations for data analytics. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2020), 2494–2504.

[25] Matthias Feurer, Benjamin Letham, and Eytan Bakshy. 2018. Scalable meta-learning for Bayesian optimization. *stat* 1050 (2018), 6.

[26] Jacob R Gardner, Matt J Kusner, Zhixiang Eddie Xu, Kilian Q Weinberger, and John P Cunningham. 2014. Bayesian optimization with inequality constraints.. In *ICML*, Vol. 2014. 937–945.

[27] Mikhail Genkin, Frank Dehne, Maria Pospelova, Yabing Chen, and Pablo Navarro. 2016. Automatic, on-line tuning of YARN container memory and CPU parameters. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 317–324.

[28] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.

[29] Anastasios Gounaris, Georgia Kougka, Ruben Tous, Carlos Tripiana Montes, and Jordi Torres. 2017. Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1891–1904.

[30] Spark Tuning guide. 2022. Tuning - Spark 3.2.1 Documentation. https://spark.apache.org/docs/latest/tuning.html

[31] Álvaro Brandón Hernández, María S Perez, Smrati Gupta, and Victor Muntés-Mulero. 2018. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems* 86 (2018), 1076–1092.

[32] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–37.

[33] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*, Vol. 11. 261–272.

[34] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 41–51.

[35] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*. PMLR, 754–762.

[36] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[37] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. 2016. Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading. In *2016 international conference on parallel architecture and compilation techniques (pact)*. IEEE, 387–400.

[38] Huaijun Jiang, Yu Shen, Yang Li, Wentao Zhang, Ce Zhang, and Bin Cui. 2023. OpenBox: A Python Toolkit for Generalized Black-box Optimization. *arXiv preprint arXiv:2304.13339* (2023).

[39] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

[40] Soroush Karimian-Aliabadi, Danilo Ardagna, Reza Entezari-Maleki, and Ali Movaghar. 2019. Scalable performance modeling and evaluation of MapReduce applications. In *International Congress on High-Performance Computing and Big Data Analysis*. Springer, 441–458.

[41] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[42] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*. 239–250.

[43] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *SIGMOD Conference*. ACM, 1667–1683.

[44] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.

[45] Palden Lama and Xiaobo Zhou. 2012. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*. 63–72.

[46] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: a survey. *AcM sIGMoD record* 40, 4 (2012), 11–20.

[47] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.

[48] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[49] Yang Li, Yu Shen, Huaijun Jiang, Tianyi Bai, Wentao Zhang, Ce Zhang, and Bin Cui. 2022. Transfer Learning based Search Space Design for Hyperparameter Tuning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 967–977.

[50] Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Zhi Yang, Ce Zhang, and Bin Cui. 2022. TransBO: Hyperparameter Optimization via Two-Phase Transfer Learning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 956–966.

[51] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, et al. 2021. Openbox: A generalized black-box optimization service. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3209–3219.

[52] Yang Li, Yu Shen, Wentao Zhang, Ce Zhang, and Bin Cui. 2022. VolcanoML: speeding up end-to-end AutoML via scalable search space decomposition. *The VLDB Journal* (2022), 1–25.

[53] Weiqing Liu, Jiannong Cao, Lei Yang, Lin Xu, Xuanjia Qiu, and Jing Li. 2016. AppBooster: Boosting the performance of interactive mobile applications with computation offloading and parameter tuning. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2016), 1593–1606.

[54] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving

Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[55] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[56] Nhan Nguyen, Mohammad Maifi Hasan Khan, and Kewen Wang. 2018. Towards automatic tuning of apache spark configuration. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 417–425.

[57] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[58] Randal S Olson and Jason H Moore. 2019. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*. Springer, 151–160.

[59] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. 2016. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*. Springer, 226–237.

[60] David Buchaca Prats, Felipe Albuquerque Portella, Carlos HA Costa, and Josep Lluis Berral. 2020. You only run once: spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2039–2051.

[61] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer school on machine learning*. Springer, 63–71.

[62] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[63] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[64] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An online Spark SQL tuning service via generalized transfer learning. *arXiv preprint arXiv:2302.04046* (2023).

[65] Rekha Singhal and Praveen Singh. 2017. Performance assurance model for applications on SPARK platform. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 131–146.

[66] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*.

[67] Ilya M Sobol. 1998. On quasi-monte carlo integrations. *Mathematics and computers in simulation* 47, 2-5 (1998), 103–112.

[68] SparkConf. 2022. Configuration - Spark 3.2.1 Documentation. https://spark.apache.org/docs/latest/configuration.html

[69] Yanan Sui, Alkis Gotovos, Joel Burdick, and Andreas Krause. 2015. Safe exploration for optimization with Gaussian processes. In *International conference on machine learning*. PMLR, 997–1005.

[70] Apache Spark Team. 2022. Apache Spark: Unified engine for large-scale data analytics. https://spark.apache.org/

[71] Spark Streaming Team. 2022. Spark Streaming. http://spark.apache.org/streaming/

[72] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 147–156.

[73] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for {Large-Scale} Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.

[74] Guolu Wang, Jungang Xu, and Ben He. 2016. A novel method for tuning configuration parameters of spark based on machine learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 586–593.

[75] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 166–173.

[76] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) (SIGMOD '22). 674–684.

[77] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. 1–6.

[78] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 13–24.

[79] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 564–577.

[80] Nikos Zacheilas, Stathis Maroulis, and Vana Kalogeraki. 2017. Dione: Profiling spark applications exploiting graph similarity. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 389–394.

[81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[82] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[83] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. 2018. A Demonstration of the Ottertune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1910–1913.

[84] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD Conference*. ACM, 415–432.

[85] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* (2022).

[86] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2023. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proceedings of the 2023 International Conference on Management of Data* (2023).

[87] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD Conference*. ACM, 2102–2114.

[88] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*. 631–645.

[89] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, et al. 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *Proceedings of the 2021 International Conference on Management of Data*. 2667–2680.

[90] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. 338–350.