Engineering Applications of Artificial Intelligence 158 (2025) 111332

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Research paper

An efficient and noise-robust framework for high-dimensional tuning in big data analytics

Jieun Lee, Sangmin Seo, Chanho Yeom, Huijun Jin, Sein Kwon, Sanghyun Park *

Department of Computer Science, Yonsei University, Seoul, 03722, Republic of Korea

ARTICLE INFO

Keywords:
Big data analytics
Configuration parameters
Spark
Bayesian optimization
Acquisition function

ABSTRACT

Spark has been a widely used platform for cluster-based big data analytics and large-scale data processing. Although the performance of Spark can be optimized by tuning parameters, its high-dimensional parameter space complicates this process and motivates the development of auto-tuning research. However, despite their success in reducing the parameter space, existing approaches require an exhaustive search to determine the optimal reduced or target dimensionality, which is time-consuming. Additionally, they overlook the impact of performance noise in Spark bottlenecks, resulting in unreliable optimal configurations. These drawbacks highlight the need for a more efficient and noise-resilient tuning strategy. To address these challenges, we propose a resource-efficient and reliable configuration tuning framework for Spark that leverages subspace-based Bayesian optimization and a noise-robust acquisition function. The proposed framework optimizes the Spark configuration by effectively reducing high-dimensional parameters without requiring a time-consuming and resource-intensive determination of the target dimensionality. Moreover, it an optimal configuration with reliable performance by minimizing the effects of noise. We validate our framework using eight HiBench workloads, comparing it to state-of-the-art baselines. Experimental results show that our framework achieves up to 14.13% faster task completion time. We provide open-source code available at: https://github.com/JIEUN-12/NoRTune.

1. Introduction

The rapid growth of the world wide web, Internet of Things, E-commerce, social media, and other applications has resulted in the daily generation of massive, continuously increasing amounts of raw data (Herodotou et al., 2020). Data analytic platforms for large-scale data processing systems, including parallel and distributed database systems such as Spark (Zaharia et al., 2010), have emerged to manage big data (Lu et al., 2019). These platforms can be deployed on both local resources and in cloud environments. Many organizations prefer to deploy big data analytic clusters in the cloud because it offers flexible, scalable, and affordable computing resources, making resource management easier than in physical environments (Islam et al., 2021).

Spark is an in-memory cluster computing system and a widely used big data analytic platform (Zaharia et al., 2016). To ensure good and robust Spark performance, database administrators (DBAs) must carefully configure its parameters, as they control various aspects of system execution, ranging from low-level memory settings and thread counts to higher-level decisions such as scheduling and resource management (Herodotou et al., 2020). However, the number of Spark parameters exceeds 150 and changes frequently with version updates, creating a high-dimensional parameter space that is challenging for DBAs to manage and understand. This challenge has spurred the development of auto-tuning research using artificial intelligence techniques, especially Bayesian optimization (BO), which has already been successfully employed in auto-tuning research (Van Aken et al., 2021; Zhang et al., 2022a; Van Aken et al., 2017; Shen et al., 2023; Zhang et al., 2023; Xin et al., 2022; Kanellis et al., 2022; Fekry et al., 2020; Dou et al., 2023; Li et al., 2023).

However, a critical limitation of BO is the curse of dimensionality, where the complexity of optimization increases exponentially with the number of tuning parameters, leading to a significant increase in sample complexity to ensure good coverage of the input space and convergence to a global optimum (Nayebi et al., 2019; Moriconi et al., 2020). Therefore, for efficient configuration tuning using BO, there are two classes of approaches for reducing this parameter space: selection- and creation-based approaches. The selection-based approach (Zhang et al., 2022a; Van Aken et al., 2017; Xin et al., 2022; Khan and Yu, 2021) analyzes the importance of parameters on the system performance by collecting massive samples, which consist of pairs of configurations and system performance metrics. It then

* Corresponding author.
E-mail address: sanghyun@yonsei.ac.kr (S. Park).

https://doi.org/10.1016/j.engappai.2025.111332
Received 22 January 2025; Received in revised form 27 April 2025; Accepted 27 May 2025
Available online 18 June 2025

selects the n most impactful parameters, dismissing the remaining parameters from the tuning process. Some selection-based methods define a small set of impactful parameters in advance using heuristics, feature analysis (Shen et al., 2023; Dou et al., 2023; Lyu et al., 2024; Li et al., 2025b), or proprietary techniques (Khan and Yu, 2021; Li et al., 2023). Conversely, the creation-based approach (Kanellis et al., 2022) creates a low-dimensional synthetic parameter space rather than selecting a subset of parameters. This approach implicitly optimizes the high-dimensional parameter space by mapping synthetic parameters to multiple dimensions of the original space, following a one-to-many mapping.

Nonetheless, these methods face three key limitations that hinder efficient and reliable tuning.

L1. High Sample Requirement. The selection-based approach requires collecting hundreds to thousands of samples to analyze the significance of parameters (Zhang et al., 2022a; Van Aken et al., 2017), which is an expensive and time-consuming process. In addition, because the importance of the parameters varies across different workloads (Xin et al., 2022; Khan and Yu, 2021), new samples must be collected for each specific workload, further increasing the demand for data and resources.

L2. Challenge of Defining Optimal Low Dimensionality. Both approaches rely on identifying an optimal low-dimensional space for effective tuning. Selection-based methods must determine not only which parameters are important but also how many to include, as this decision directly affects tuning performance, excluding even a single impactful parameter can lead to suboptimal results. Creation-based methods similarly depend on discovering the correct active subspace in which the objective function varies most significantly (Nayebi et al., 2019; Papenmeier et al., 2022). The active subspace refers to a lower-dimensional synthetic space, derived from the original high-dimensional configuration space, in which the objective function exhibits most of its meaningful variation. Unfortunately, because the active subspace is usually unknown, the tunable dimensionality must be determined by guessing or empirical searching, which incurs additional costs. Although the creation-based approach emphasizes sample efficiency, optimal performance is achieved only when the optimal low dimensionality is found, making resource-efficient tuning challenging.

L3. Lack of Noise-Aware Tuning. Beyond the limitations noted, noise in system performance metrics presents another critical challenge. In practice, Spark experiences performance bottlenecks mainly due to network input/output (I/O), disk I/O, garbage collection (Kang and Lee, 2020), and shared hardware in the cloud environment (Van Aken et al., 2021). However, previous studies have often overlooked these effects in their tuning frameworks. Instead, they mitigated this noise by establishing robust experimental setups such as repeating tuning sessions (Zhang et al., 2022a; Kanellis et al., 2022; Li et al., 2023) or conducting multiple evaluations for each configuration (Shen et al., 2023; Dou et al., 2023). Although these methods aimed to reduce variability, the optimal configuration identified by the tuner might perform well merely due to noise, leading to unreliable tuning performance that may not sustain good results. Table 1 categorized existing auto-tuning approaches, including our proposed method, accounting for the three practical limitations discussed.

To address the noted limitations in an integrated manner, we propose NoRTune, a noise-robust and sample-efficient framework for high-dimensional configuration tuning in Spark. NoRTune avoids the need to pre-select dimensionality or impactful parameters and improves tuning reliability in the presence of performance variability in cloud environments by integrating techniques from subspace-based BO and noise mitigation strategies. Specifically, our contributions are as follows:

- **Efficient high-dimensional tuning.** NoRTune leverages subspace-based BO to handle high-dimensional Spark parameter spaces without requiring the pre-selection of dimensionality or impactful parameters. This approach eliminates the need to estimate active subspace dimensions, thereby overcoming the resource-intensive drawbacks of selection- and creation-based approaches. As a result, it enables sample-efficient and scalable tuning across diverse workloads.
- **Incorporating noise mitigation techniques.** NoRTune incorporates a noise-robust acquisition function into its tuning process to address performance variability in cloud environments. This strategy ensures reliable configuration tuning, even under noisy conditions—a challenge that, to the best of our knowledge, remains unexplored in extant Spark tuning studies.
- **Experimental validation.** We validate NoRTune on eight Spark applications in the HiBench suite, demonstrating its scalability and superior performance over state-of-the-art models. We also conduct an ablation study to evaluate the contributions of individual components and analyze the alignment between tuned configurations and workload characteristics. Furthermore, we extend the evaluation to PostgreSQL, showcasing NoRTune's general-purpose applicability beyond Spark. These results confirm NoRTune's adaptability to diverse workloads and database systems, highlighting its practicality and effectiveness for real-world usage in cloud environments.

The remainder of this paper is organized as follows. Preliminaries are presented in Section 2, the proposed NoRTune framework is described in Section 3, and evaluation results are discussed in Section 4.

2. Preliminaries

2.1. Spark framework

Spark (Zaharia et al., 2010) is a cluster computing framework for large-scale data processing, typically consisting of a single master node and several worker nodes. Data are stored in memory as an immutable collection within resilient distributed datasets (RDDs), which are split into partitions distributed across worker nodes and operated in parallel. Spark performs operations on RDDs through transformations and actions. Transformation functions create new RDDs from existing ones and are executed lazily, building a lineage of transformations. Action functions trigger the execution of these lazy functions and produce meaningful results. To optimize execution, Spark constructs a directed acyclic graph (DAG), which represents the sequence of transformations. The DAG scheduler then divides the graph into stages and tasks for an efficient parallel execution across the cluster.

When submitting Spark applications, the Spark configuration parameters are specified to satisfy user requirements, which may include factors such as resource allocation and execution time. For example, *spark.executor.memory* and *spark.executor.cores* define the memory size and number of CPU cores per Spark executor, respectively. Additionally, *spark.sql.file.shuffle.partitions* controls the default number of partitions used during shuffle operations to optimize parallel processing and performance. Spark applications are controlled by up to 160 configuration parameters that determine the optimal performance structure (Spark, 2024). Therefore, an appropriate configuration setup can improve Spark performance. However, the number of tunable parameters is large and difficult to comprehend altogether, which has spawned many automatic parameter tuning studies (Shen et al., 2023; Xin et al., 2022; Khan and Yu, 2021; Fekry et al., 2020; Dou et al., 2023; Li et al., 2023; Lu et al., 2019; Herodotou et al., 2020; Li et al., 2025a; Lyu et al., 2024; Masouros et al., 2024; Wu et al., 2024; Li et al., 2025b).

Table 1
Comparison of auto-tuning works, including NoRTune, in terms of their sample pre-collection requirements (Sample Requirement), their strategy for defining or avoiding optimal low dimensionality (Defining Optimal Low Dimensionality), and whether their tuning framework accounts for variability or noise in system performance (Noise-aware Tuning).

Category	Work	Sample Requirement	Defining Optimal Low Dimensionality	Noise-Aware Tuning
Selection-based	OtterTune (Van Aken et al., 2017)	✔	Incremental Method	✗
	ROBOTune (Khan and Yu, 2021)	✔	Threshold-based Filtering	✗
	Zhang et al. (2022a)	✔	Empirical Search	✗
	LOCAT (Xin et al., 2022)	✔	Empirical Search	✗
	OpAdvier (Zhang et al., 2023)	✔	Empirical Search	✗
	Rover (Shen et al., 2023)	✔	Pre-defined by SHAP and Experts	✗
	Online-Tune (Li et al., 2023)	✔	Adaptively During Tuning	✗
	TurBO (Dou et al., 2023)	✔	Pre-defined Manually	✗
	HMOOC (Lyu et al., 2024)	✔	Pre-defined Manually	✗
	CSAT (Li et al., 2025b)	✗	Pre-defined Manually	✗
Creation-based	LlamaTune (Kanellis et al., 2022)	✗	Empirical Search	✗
	NoRTune (ours)	✗	**Not Needed**	✔

2.2. Bayesian optimization

The BO method is used to optimize expensive-to-evaluate black-box functions. The main advantages include its ability to estimate the uncertainty of unseen configurations based on observations and to balance exploration and exploitation when selecting the next configuration to be evaluated. BO aims to find globally optimal or near-optimal configurations. The BO pipeline integrated with configuration tuning is outlined in Algorithm 1.

Algorithm 1 Bayesian Optimization

Input: evaluation budget m, initial sample size n_{init}
Output: the best Spark configuration x^*
1: Sample initial observation $\mathcal{D}_0 \leftarrow \{(x_i, y_i)\}^{n_{init}}$
2: **for** $t = 1$ to m **do**
3: Fit surrogate model with $\mathcal{D}_{t-1}$
4: Select candidate configuration x_t by maximizing the acquisition function α
5: Evaluate y_t and update data $\mathcal{D}_t \leftarrow \mathcal{D}_{t-1} \cup \{(x_t, y_t)\}$
6: **end for**
7: **return** best configuration $x^* \in \mathcal{D}_m$

First, an initial set of observations is randomly sampled, denoted as $\mathcal{D}_0 = \{(x_1, y_1), \ldots, (x_{n_{init}}, y_{n_{init}})\}$, with n_{init} typically being small (e.g., 5–10 samples). This initial dataset serves as the starting point for the BO process, which then incrementally updates the surrogate model over a budget of m evaluations, where m is the total number of iterations allowed. In each iteration, the three steps are repeated sequentially. (1) BO fits a surrogate model using the current observation data, $\mathcal{D}_{t-1}$; (2) BO selects the next candidate configuration, x_t, which maximizes acquisition function α; (3) x_t is evaluated by the objective function to obtain y_t, and BO updates the observation data by adding the new observation, $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{(x_t, y_t)\}$. After completing m iterations, BO returns the best configuration, $x^* \in \mathcal{D}_m$.

The surrogate model and acquisition function are critical components of BO. The surrogate model approximates the objective function to estimate unknown data points instead of using an expensive function for efficient optimization. A Gaussian process (GP) (Williams and Rasmussen, 2006) is commonly used as it models the uncertainty of observations. It is a regression model that assumes a multivariate normal prior distribution, characterized by a mean vector and covariance matrix (Frazier, 2018). Given a new configuration, x_t, the posterior mean, μ, and variance, σ^2, functions are updated as follows:

$$\mu_t(x) = k_t(x)^{\mathrm{T}}(\mathrm{K}_t + \sigma^2\mathrm{I})^{-1} y_{i:t} \tag{1}$$

$$\sigma_t^2 x = k(x,x) - k_t(x)^{\mathrm{T}}(\mathrm{K}_t + \sigma^2\mathrm{I})^{-1} k_t(x) \tag{2}$$

where $k(x,x)$ is a kernel function and $k_t(x) = [k(x_i, x)]_{i=1}^{t}$. K_t is the positive definite kernel matrix, $[k(x_i, x_j)]_{i,j\in[t]}$. The kernel function can be defined using various methods depending on the data type. For instance, the Matérn kernel is often used for continuous variables, where as the Hamming distance is applied to categorical variables.

The acquisition function selects the next candidate configuration by considering both the prediction of the surrogate model and its associated uncertainty, thereby balancing the exploration of unknown regions with the exploitation of known high-performance regions. The next candidate configuration, x_t, is selected by maximizing the acquisition function, α, over N randomly sampled configurations, calculated as follows:

$$x_t = \arg\max_{x \in \{x_i\}_{i=1}^{N}} \alpha(x) \tag{3}$$

Maximizing the acquisition function is crucial in BO as it guides practical decisions, such as selecting the next potential optimal configuration, which is vital to optimizer convergence (Wilson et al., 2018). Several methods (Srinivas et al., 2010; Jones et al., 1998; Kushner, 1964; Hernández-Lobato et al., 2014) can be used for the acquisition function.

BO is widely used in tuning frameworks due to its ability to efficiently explore configuration spaces while minimizing expensive evaluations. As a result, BO-based optimizers have been commonly employed in recent studies (Zhang et al., 2022a; Van Aken et al., 2017; Shen et al., 2023; Zhang et al., 2023; Xin et al., 2022; Kanellis et al., 2022; Dou et al., 2023; Li et al., 2023; Eriksson et al., 2019; Zhang et al., 2022b), owing to their robustness to noise and effectiveness in solving black-box problems. These studies utilized advanced BO techniques to enhance optimization performance. For example, Zhang et al. (2022b) built on the trust region concept introduced in Eriksson et al. (2019) to constrain the search space, Zhang et al. (2022a) and Li et al. (2023) use mixed kernels to handle complex or heterogeneous data types. Kanellis et al. (2022) applied subspace-based BO to address high-dimensional search spaces.

2.3. High-dimensional configuration tuning

Spark includes hundreds of parameters that affect its structure and performance. This fact leads to an exponentially growing configuration space, which requires its complete traversal – a wicked hard problem – to explore and optimize efficiently. Recent studies on automatic configuration tuning have focused on reducing the complexity of this high-dimensional space using two distinct approaches: selection- and creation-based approaches , as summarized in Table 1.

Selection-based approaches. The first approach is to tune only a selected subset of the important parameters by computing their

contribution to overall system performance. Various importance measurements are employed to prune the configuration space and enhance optimization efficiency (Zhang et al., 2022a). For example, OtterTune (Van Aken et al., 2017) uses the Lasso path algorithm to rank parameters, but it assumes linear relationships that may fail to capture nonlinear effects. Recently, SHAPley additive explanation (SHAP) methods (Lundberg and Lee, 2017) have gained wide adoption (Zhang et al., 2022a; Shen et al., 2023; Zhang et al., 2023, 2021) as they support the selection of both beneficial and detrimental parameters. One of these, Rover (Shen et al., 2023), integrates SHAP scores with expert input using half SHAP-selected and half expert-selected parameters. LOCAT (Xin et al., 2022) is another that evaluates parameter importance using Spearman correlation coefficients, and ROBOTune (Khan and Yu, 2021) employs tree-based models to estimate importance via the coefficient of determination, selecting parameters based on predefined thresholds. In contrast, Online-Tune (Li et al., 2023) adjusts the subset of tunable parameters dynamically, based on the success rate of tuning attempts. Other methods define a small set of impactful parameters manually, using heuristics or domain expertise (Lyu et al., 2024; Li et al., 2025b).

Unfortunately, these methods have drawbacks. First, they require thousands of samples to calculate meaningful parameter importance, and this costly process must be repeated for every new workload, as both performance and parameter relevance vary depending on the workload (Zhang et al., 2022a; Van Aken et al., 2017). In addition, as shown in Table 1, most selection-based methods depend heavily on sample collection, which supports knowledge transfer and warm-start techniques across workloads (Van Aken et al., 2017; Khan and Yu, 2021; Xin et al., 2022; Zhang et al., 2023; Shen et al., 2023; Li et al., 2023), further increasing the data collection burden. Another drawback is determining the correct number of parameters to tune. A large configuration space increases the optimization time, whereas too few parameters would limit the ability to find the best configuration. To address this, some studies have empirically explored different subset sizes to find effective low-dimensional tuning spaces (Zhang et al., 2022a; Xin et al., 2022; Zhang et al., 2023). OtterTune introduces an incremental expansion approach, gradually increasing the parameter subset during tuning. Notably, this technique does not consistently yield improved performance across workloads (Zhang et al., 2022a). As a result, selecting the optimal number of parameters remains an open problem. However, some recent approaches (e.g., Online-Tune and Sparkle Masouros et al., 2024) have attempted to address the problem by dynamically adjusting the subset size or treating all parameters as tunable; yet, they still fundamentally rely on extensive sampling, which incurs substantial overhead.

Creation-based approach. To address these issues, the second approach creates a synthetic space instead of pruning the configuration space and worrying about the appropriate number of parameters. This approach reduces the size of the search space to facilitate efficient exploration in the BO, thereby implicitly optimizing a high-dimensional configuration space. Unlike selection-based methods, it avoids the need to collect thousands of samples and pre-identify important parameters. For example, LlamaTune (Kanellis et al., 2022), a sample-efficient tuner, proposes a novel approach that avoids the need to identify the exact set of important parameters while realizing the benefits of low-dimensional tuning. LlamaTune employs an advanced BO algorithm, hashing-enhanced subspace BO (HeSBO) (Nayebi et al., 2019), which generates a low-dimensional synthetic space from the original high-dimensional space through a random projection matrix. The synthetic parameters do not represent actual configuration parameters directly but determine the value of one or more real DBMS configuration parameters. LlamaTune successfully reduces the configuration search space through this low-dimensional tuning and handles special parameter values and their large value ranges, ensuring the tuning of the high-dimensional original configuration parameters. However, the effectiveness of such approaches critically depends on selecting an appropriate low dimensionality, which is typically unknown in practice. As a result, users must either engage in empirical searches to guess a suitable dimensionality or risk suboptimal performance—posing a burden similar that of selection-based approaches. We further discuss this challenge in Section 2.4.

2.4. Challenges

Guessing the proper target dimensionality. Although subspace-based BO such as the HeSBO successfully improves the tuning performance by effectively reducing the configuration space, an inherent challenge remains for successful optimization. A successful approach for high-dimensional BO identifies an effective subspace, the active subspace, with dimensionality d_e (Papenmeier et al., 2022). The active subspace refers to a synthetic lower-dimensional space obtained by projecting the original high-dimensional configuration parameters, where the objective function exhibits most of its significant variations. Note that this is distinct from simply selecting a subset of parameters. If the target dimensionality d is greater than d_e, the optimal solution is clearly included. Otherwise, finding the optimum becomes impossible. However, this active subspace is typically unknown, and determining the optimal target dimensionality remains at the discretion of the user. The main problem is that if the target dimensionality is set extremely small, the subspace cannot adequately represent the objective function. Conversely, if the dimensionality is extremely high, the curse of dimensionality slows down the entire tuning process, thereby requiring a significantly larger evaluation budget and reducing the efficiency of the optimization.

Because HeSBO approximates the active subspace by projecting the high-dimensional configuration space onto a randomly chosen lower-dimensional subspace, selecting an appropriate target dimensionality, d, becomes crucial for successful tuning. We conducted a case study to analyze the impact of target dimensionality on tuning performance. In this experiment, we used LlamaTune with 45 Spark parameters and four target dimensionality settings (5, 10, 20, and 30) for three workloads: PageRank, KMeans, and TeraSort. Note that these dimensionalities define synthetic subspaces obtained through random projections from the same original 45-parameter configuration space. As such, all target dimensionalities originated from the same base space, and no manual parameter subset selection, such as those used in selection-based methods, was involved. The evaluation budget was set to 50, and each configuration of iterations was evaluated three times to obtain the average performance, accounting for noise in the results. The models aim to minimize the runtime duration of Spark applications. Our experiments were conducted in the hardware environment described in Section 4.1.

The results are presented in Fig. 1, where the solid lines represent the average results, and the shaded region reflects the area between upper and lower bounds. The best-performing models were LlamaTune (20) for PageRank, LlamaTune (10) and LlamaTune (20) for KMeans, and LlamaTune (10) for TeraSort. Conversely, the worst-performing model was LlamaTune (5) for all workloads; the target dimensionality of the synthetic search space (dimensionality d) was too small (less than d_e) to contain optimal solutions; it should be greater than five. Meanwhile, the LlamaTune (30) model showed unstable tuning performance, indicating that a high target dimensionality makes the search space too large for effective exploration. This instability illustrates the difficulty of exploring high-dimensional spaces effectively. Furthermore, these results emphasize that the target dimensionality for a subspace should be carefully chosen: medium target dimensionalities of 10–20 provide robust tuning performance; however, it is unclear which dimensionality is the most robust. By contrast, both extremely small (5) and extremely large (30) target dimensionalities performed the worst on all workloads.

These results indicate that the appropriate target dimensionality varies across workloads and is not apparent in advance. Consequently, users must empirically determine a suitable dimensionality

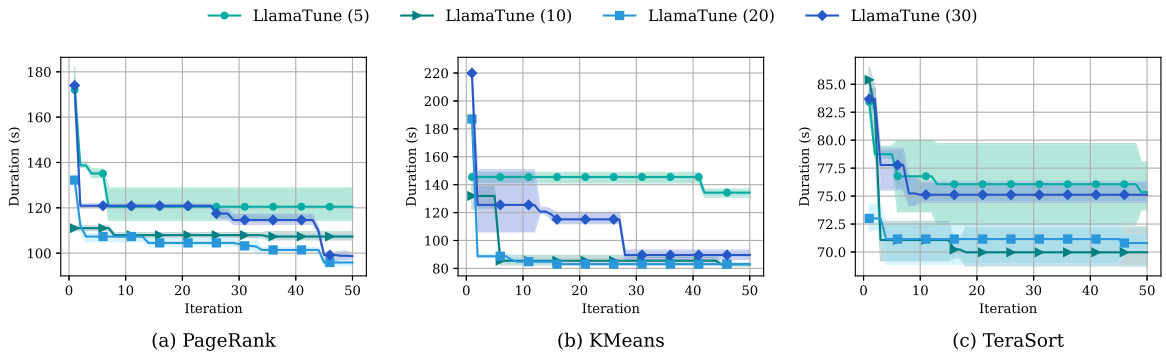

Fig. 1. Minimum durations at each iteration for four LlamaTune models with different target dimensionalities on three workloads: PageRank, KMeans, and TeraSort.

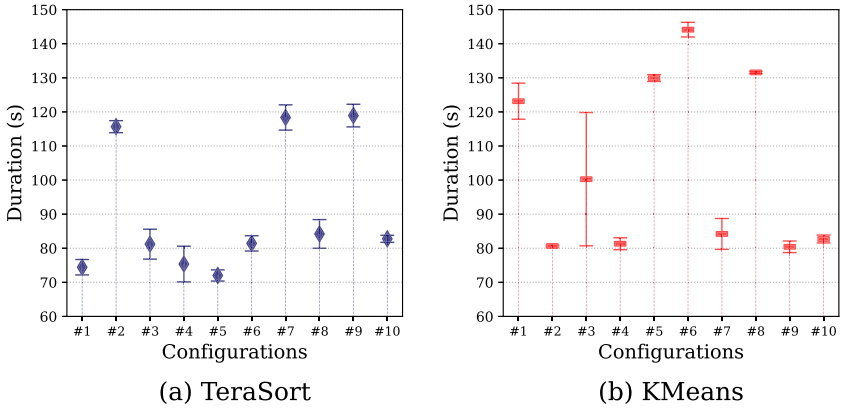

Fig. 2. Repeated performances on two workloads: TeraSort and KMeans. The X-axis denotes the index of configuration, and the Y-axis represents the duration.

for each case, either by guessing or trial-and-error, which risks sub-optimal tuning or leads to excessive search overhead. This creates a fundamental contradiction with respect to LlamaTune: although it aims to achieve sample-efficient tuning, users must engage in a time-consuming decision-making process to determine the target dimensionality, thereby undermining its efficiency. NoRTune addresses this challenge by leveraging an advanced BO algorithm that eliminates the need to define or guess the fixed target dimensionality.

Handling noisy observations. With BO, potential optimal configurations are generated at each iteration and evaluated by benchmarking workloads. However, results using the same configuration and workload are not always consistent, as shown in Fig. 2, which shows the results of 10 different configurations evaluated three times in the same environment. As shown, the benchmarking results for the same configuration are subject to certain degrees of error, which are influenced by multiple factors (e.g., local and cloud environmental variables) (Van Aken et al., 2021; Kang and Lee, 2020). Therefore, despite the tuner recommending optimal configurations based on previous observations, the actual configuration may exhibit only moderate performance gains instead of the significant improvements expected. For example, as shown in Fig. 2, configuration # 4 on TeraSort and configuration # 3 on KMeans achieved minimum durations while also showing moderate or worse overall performance. Particularly, configuration # 3 on KMeans exhibited a large variance in performance, indicating unreliability.

Therefore, studies continue to use experimental setups to mitigate noise, which can be divided into two distinct strategies. The first involves repeating tuning sessions 5–10 times using different random seeds and reporting the average performance results (Zhang et al., 2022a; Kanellis et al., 2022; Li et al., 2023). The second strategy involves evaluating each configuration three-to-five times during BO iterations with tuning sessions, including recording the mean best-observed performance results (Shen et al., 2023; Dou et al., 2023). Other studies avoid both strategies and record tuning performance from a single execution (Van Aken et al., 2017; Xin et al., 2022), which tends to ignore the effects of noise, again leading to unreliability.

Many studies apply the BO pipeline in noise-free cases, which also overlooks commonplace unexpected behaviors (Letham et al., 2019). Recognizing this limitation, we incorporate all these factors into our tuning framework, focusing on acquisition functions. Although there are several methods for handling noisy observations (e.g., modifying the kernel), (Picheny et al., 2013) showed that the choice of acquisition function leads to significant differences in performance in noisy environments, compared with kernel functions. NoRTune overcomes this by incorporating a noise-robust acquisition function, allowing us to improve the reliability and performance of the Spark tuning configuration.

3. NoRTune design

3.1. Problem definition

We formulated the Spark configuration tuning as a noisy black-box optimization problem whose input is the configuration $x = \{\theta_1, \dots, \theta_D\}$, where each $\theta_i \in \Theta_i$ denotes a tunable parameter, and the configuration space is defined as $\mathcal{X} = \Theta_1 \times \cdots \times \Theta_D$. The parameters include continuous-valued (e.g., spark.memory.fraction), categorical (e.g., spark.io.compression codec), Boolean (e.g., spark.shuffle.compress), and numerical types (e.g., spark.default.parallelism). Given a specific workload, the performance of a configuration x is observed as

$$y = f(x) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2) \tag{4}$$

where $f(x)$ denotes the latent (noise-free) performance function, typically measured by execution time, and ϵ models the runtime fluctuations commonly observed in real-world Spark environments. These

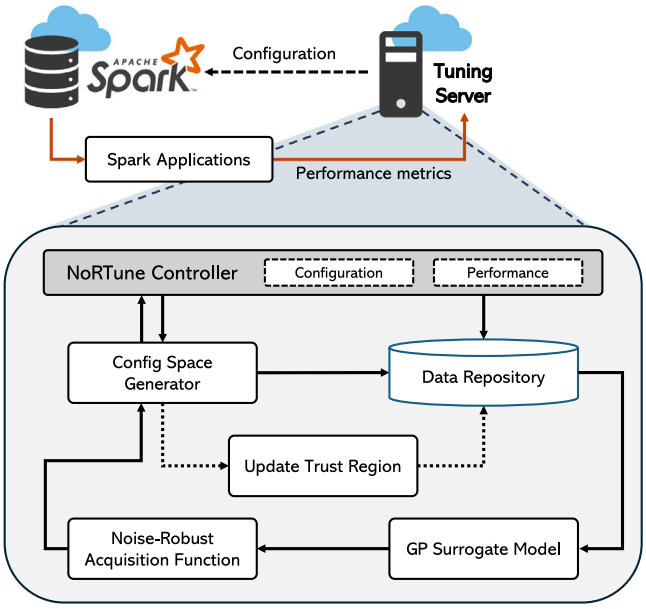

Fig. 3. Overview of NoRTune.

variations typically result from network or disk I/O delays, garbage collection, and resource contention in shared cloud systems (Van Aken et al., 2021; Kang and Lee, 2020). Our objective is to find an optimal or near-optimal configuration x^* that minimizes the performance metric while explicitly accounting for observation noise. Formally, the problem is defined as:

$$x^* = \arg\min_{x \in \mathcal{X}} f(x) \tag{5}$$

3.2. Overview

NoRTune is a high-dimensional configuration auto-tuning framework for Spark that efficiently handles performance variability and dimensional complexity. As illustrated in Fig. 3, NoRTune operates in an iterative loop, maintaining an active connection with the Spark master node throughout the tuning session. At each iteration, the **NoRTune controller** dispatches a Spark configuration to be evaluated and collects the corresponding performance metrics. The sampled configuration and performance data are forwarded to both the **config space generator** and the **data repository**. The **config space generator** constructs a low-dimensional synthetic space, which serves as a target search space for optimization, mapping from the original high-dimensional Spark configuration space. After synthetic parameter generation, a data triplet—(Spark configuration, its corresponding synthetic parameters, and the measured performance metric)—is stored in the **data repository**. These samples are referred to as observations.

Based on these observations, NoRTune fits a **Gaussian process (GP) surrogate model** over the synthetic space, where both synthetic parameters and performance metric are standardized for modeling. To select the next candidate configuration, a **noise-robust acquisition function** is applied, which identifies promising candidates while accounting for the noise variance of the current observations. These candidates, defined in the synthetic space, are then mapped back to the full configuration space by the **config space generator**. The **NoRTune controller** evaluates the new configuration on Spark and appends the results to the **data repository**. Following each iteration, the **trust region** within the synthetic space is updated in response to observed performance, enabling NoRTune to adaptively guide subsequent search steps. This process iterates until the evaluation budget is exhausted.

3.3. Nested subspace BO

To address the challenge of capturing the active subspace during optimization, one practical strategy is to use a target subspace of dimensionality d that exceeds the unknown effective dimensionality d_e. This strategy must satisfy two criteria: (1) all d_e active subspace dimensions are mapped to distinct d target dimensions (Papenmeier et al., 2022), and (2) if $d \geq d_e$, the target subspace contains the global optimum with probability one (Wang et al., 2016).

To meet these criteria, we propose the Nested Subspace-based Bayesian Optimization (NSBO) approach, based on random sparse embedding, which leverages the HeSBO family. This approach builds upon Bounce (Papenmeier et al., 2023), which we have modified to better suit Spark configuration tuning. Notably, Bounce addresses batch parallelism to efficiently evaluate objective functions. However, with Spark tuning, nodes must be provided with sufficient resources to simultaneously execute multiple jobs corresponding to the different parameter configurations. Therefore, we decided to exclude the batch method in our algorithm.

A key distinction between NSBO and HeSBO lies in how the target dimensionality is managed. NSBO incrementally increases the target dimensionality until it reaches the input dimensionality, whereas the HeSBO uses a fixed low-dimensional target space. NSBO's dynamic adjustment allows it to consistently satisfy the given criteria, guaranteeing a worst-case success probability of one. Consequently, NSBO can find a near-optimal configurations, even under challenging conditions. In contrast, HeSBO offers a low probability of success when the target dimensionality d is smaller than the effective dimensionality d_e (Letham et al., 2020), since it fails to satisfy the second criterion. Recall that d_e is unknown, making it more difficult to define the optimal d. Therefore, NSBO serves as a principled sparse embedding strategy that adaptively includes the active subspace and supports robust optimization in high-dimensional problems. In the following sections, we describe the key elements of NSBO and provide its high-level algorithm.

Nested random embedding. NSBO leverages nested random embedding, which has a few features in common with HeSBO. The sparse projection matrix, $S \in \{0, \pm 1\}^{d \times D}$ (i.e., the embedding matrix) has precisely one non-zero entry in each column. This matrix, $S : \mathcal{Z} \to \mathcal{X}$, maps a synthetic configuration, $z = \{\theta'_1, \ldots, \theta'_d\}$, including $z \in \mathcal{Z}$, to a Spark configuration, $x = \{\theta_1, \ldots, \theta_D\}$, with $x \in \mathcal{X}$, where $x = Sz$. The performance metric is $y = f(Sz) + \epsilon$, where ϵ $\mathcal{N}(0, \sigma_\epsilon^2)$ represents observation noise. σ_ϵ^2 denotes the likelihood noise variance in the context of the learned GP model, which is discussed later. Because the Spark configuration consists of mixed spaces, including categorical, binary, numerical, and continuous variables, those of synthetic configuration $z = \{z_{cont}, z_{nume}, z_{cate}, z_{bina}\}$ are represented separately. Continuous and numerical variables are normalized to $[-1, 1]$, binary variables are assigned to $\{-1, +1\}$, and categorical variables are represented using one-hot encoding. Details can be found in Papenmeier et al. (2023). Note that NSBO optimizes the synthetic configuration, z, then projects the optimized synthetic configuration onto the input-dimension space, returning the optimal Spark configuration, x^*.

Bin and split. Each synthetic parameter, θ'(i.e., a **bin**) is formed by grouping a subset of input parameters. The bin size is incrementally increased until it reaches the full input dimensionality D during tuning, using a strategy called **split**. At each step, all bins are split unless they can no longer be divided (e.g., when a bin contains only a single input dimension). When splitting occurs, the embedding matrix S is updated by reassigning input dimensions from the current bin to newly created bins. This nested structure allows the reuse of previous observations as the search space grows, without rendering past evaluations obsolete. To maintain consistency, only variables of the same type are grouped into the same bin.

Example. Fig. 4 presents an example of the splitting process. Here, we consider only numerical and binary parameters, and assumes two

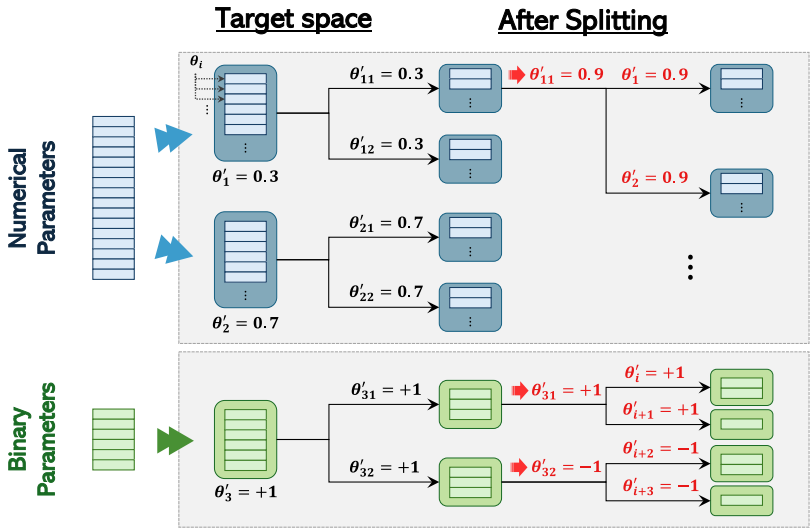

Fig. 4. An example of the splitting steps for the initial target dimensionality as 3 and $b = 1$. Two types are represented: numerical and binary parameters.

hyperparameters: $b = 1$ (the number of new bins per split) and $d_0 = 3$ (the initial target dimensionality). The number of input dimensions assigned to each target dimension (i.e., bin) is balanced within each variable type and determined independently across types. However, when assigning the initial target dimensionality d_0 exceeds the number of variable types, NSBO performs a proportional adjustment based on the relative number of parameters in each type. In this example, since $d_0 = 3$ and the variable types are numerical and binary (totaling 2 types), the target dimensionality is larger than the number of types. Given that number of numerical parameters (e.g., 16) exceeds that of binary parameters (e.g., 6), NSBO assigns two bins (θ'_1 and θ'_2) to numerical parameters and one bin (θ'_3) to binary parameters. Among the numerical parameters, one bin is assigned to approximately half of them, while the binary bin groups all binary parameters together.

If a split occurs (e.g., on θ'_1), the bin is split into two new bins—θ'_{11} and θ'_{12}—as determined by $b = 1$. The input dimensions originally grouped in θ'_1 are now randomly distributed between the two new bins. Importantly, the value of the original bin ($\theta'_1 = 0.3$) is copied to both new bins ($\theta'_{11} = 0.3$ and $\theta'_{12} = 0.3$) to maintain observation consistency. This copying ensures that the GP model can continue to leverage past observations even after the dimensionality of the synthetic space changes. Without this guaranteed consistency, previous observations would be incompatible due to dimension mismatch, making them unusable. By maintaining consistency across split steps, NSBO preserves information and supports robust optimization. The right half of Fig. 4 shows this progression through the split.

Computing budget and factors. As the target dimensionality, d, increases, the evaluation budget, m_D, which denotes the total budget for all iterations, is proportionally allocated to the different target dimensionalities. The evaluation budget, m_i, for the ith subspace with target dimensionality d_i is calculated as:

$$m_i = \left\lfloor \frac{b\left(m_D - n_{init}\right) d_i}{d_0\left((b+1)^{n_s+1} - 1\right)} \right\rceil \tag{6}$$

where d_0 is the initial target dimensionality, and n_{init} is the number of initial samples randomly generated before starting the BO process. n_s is the number of splits required to reach the input dimensionality D, and i represents the current split level (e.g., $i = 1$ means one split has occurred). NSBO calculates these factors in the following order. NSBO first computes n_s as $\lceil \log_{b+1} D/d_0 \rceil$, and then updates the split factor as $b = round\left(\log_{n_s} \frac{D}{d_0} - 1\right)$. The target dimensionality d_i is then defined as $d_0 (b+1)^{n_s}$. Finally, m_i is calculated as described. This formulation ensures that more evaluation budget is allocated to higher-dimensional subspaces, enabling finer optimization as the search space expands. Consequently, the NSBO performs fewer iterations in low-dimensional subspaces and allocates more effort to complex, high-dimensional regions.

Mixed kernel function. We used a GP surrogate model with a mixed kernel function to support different variable types. Following Papenmeier et al. (2023), the kernel is defined as a mixture of the sum and product of two separate Matérn kernels. This formulation allows the model to capture both continuous and combinatorial patterns in the configuration space. To simplify the kernel calculations, we defined continuous variables to include both continuous and numerical types and combinatorial variables to include both binary and categorical types.

Trust region. To support effective and reliable exploration – especially as the target dimensionality increases exponentially – NSBO adopts a trust region (TR) strategy, as provided in TuRBO (Eriksson et al., 2019). The TR is defined as a hyper-rectangle centered at the current best solution, with its size controlled by a base length L. It determines the range within which new configurations are explored. The TR volume is adjusted dynamically: it is expanded (i.e., L is increased) when a new configuration outperforms the previous best, and shrunk (i.e., L is decreased) otherwise. When L becomes smaller than a predefined threshold L_{min}, the TR is reset—i.e., the base length is restored and the center is updated. This mechanism encourages both local refinement and global exploration. NSBO maintains separate base lengths for continuous and combinatorial variables, denoted as L^{cont} and L^{comb}. Each has its own range, $L^{cont} \in [L^{cont}_{min}, L^{cont}_{max}]$. Notably, a reset of the TR also triggers a split in the target dimensionality.

Overall algorithm flow. Algorithm 2 presents the NSBO pseudocode, which aims to determine the best Spark configuration, x^*, that minimizes the objective function, f. The algorithm begins by initializing several hyperparameters, including the initial target dimensionality, d_0, maximum evaluation budget, m_D, number of new bins per split, b (i.e., split factor), initial sample size, n_{init}, and index of the target dimensionality, i.

Initialization (lines 1–5): The algorithm begins by setting i to zero and computes the required number of splits, n_s, adjusting b accordingly. The evaluation budget, m_i, for target dimensionality d_i is calculated, and the projection matrix, S, is initialized. Finally, initial data, D_0, are collected by sampling in pairs $(z_k, f(Sz_k))$.

Optimization Loop (lines 6–11): The main loop runs for up to m_D iterations. During each, the TR base lengths, L_{cont} and L_{comb}, are initialized. A nested loop continues as long as both base lengths exceeds

their minimum thresholds. Within this loop, the following four tasks are completed. (1) A GP model is fitted using the current dataset, $\mathcal{D}_{j-1}$. (2) The acquisition function, α_{AEI}, is calculated to select the next candidate, z_j, where α_{AEI} is described in the next section. (3) z_j is mapped to the input space via $x_j = Sz_j$ and evaluated; $\mathcal{D}_j$ is then updated. (4) The TR base lengths, L_{cont} and L_{comb}, are updated (i.e., expanded or shrunk) based on whether x_j improves upon the current best result.

Dimension increase (lines 13–17): When $d_i < D$, the algorithm increments the index i, updates the projection matrix S to increase dimensionality, and recalculates m_i. This step corresponds to the embedding split process.

Resampling (lines 18–20): After the target dimensionality reaches D, the algorithm resets the dataset by resampling and evaluating new initial points. The projection matrix, S, is resampled, and the TR base lengths and iteration counter j are also reset to enable continued optimization in the full-dimensional space.

Algorithm 2 Nested Subspace Bayesian Optimization

Input: initial target dimensionality d_0, maximum evaluation budget m_D, # new bins per split b, initial sample size n_{init}, index of target dimensionality i

Output: the best Spark configuration $x^* \in \arg\min_{x\in\mathcal{X}} f(x)$

1: $i \leftarrow 0$
2: Compute n_s and b in order, $n_s \leftarrow \lceil \log_{b+1} D/d_0 \rceil, b \leftarrow \left\lfloor \log_{n_s} \frac{D}{d_0} - 1 \right\rfloor$
3: Compute evaluation budget m_i for d_i
4: Initialize embedding matrix, $S : \mathcal{Z} \rightarrow \mathcal{X}$
5: Sample initial data $D_0 \leftarrow \{(z_k, f(Sz_k))\}_{k\in n_{init}}$
6: **for** $j = 1$ to m_D **do**
7: Initialize TR base lengths L_{cont} and L_{comb}
8: **while** $L_{cont} > L_{\min}^{cont} \wedge L_{comb} > L_{\min}^{comb}$ **do**
9: Fit $\mathcal{GP}$ with $\mathcal{D}_{j-1}$ and get candidate z_j by calculating α_{AEI}
10: Evaluate $f(Sz_j)$ and update $\mathcal{D}_j$, $\mathcal{D}_j \leftarrow \mathcal{D}_{j-1} \cup \{(z_j, f(Sz_j))\}$
11: Update L_{cont} and L_{comb}
12: **end while**
13: **if** $d_i < D$ **then**
14: $i \leftarrow i + 1$
15: Increase embedding matrix S
16: $d_i \leftarrow$ # rows in S
17: Re-compute evaluation budget m_i
18: **else**
19: Reset D by resampling and evaluating new initial points
20: Resample S, reset L_{cont} and L_{comb}, $j \leftarrow j + n_{init}$
21: **end if**
22: **end for**

3.4. Noise-robust acquisition function

As mentioned in the challenge, the acquisition function in BO is crucial because it balances exploration and exploitation, guiding the optimization process towards the most promising points to be evaluated next. The expected Improvement (EI) is the most popular choice, but is calculated as follows for noise-free observations:

$$\alpha_{EI}(z_t|f^*) = \mathbb{E}\left[\max(0,\ f^* - y_t)\middle| y_t \sim \mathcal{GP}(z_t|\mathcal{D}_{t-1})\right] \tag{7}$$

where $f^* = \min_{k\in\mathcal{D}_{t-1}} f(z_k)$ denotes the current best result, which is the minimum value among the current observations in $\mathcal{D}_{t-1}$. Because the evaluated result, y_t, of z_t is unknown at the current step, $t-1$, it is estimated using the GP surrogate model, fitted to the current observation data in $\mathcal{D}_{t-1}$.

Conversely, for noisy observations, the objective function, f, is no longer a deterministic problem because $y_i = f(x_i) + \epsilon_i$, where ϵ_i represents the observation noise. In such cases, directly minimizing the observed values yields an unreliable estimate of the optimum. Consequently, computing EI becomes challenging since the current best value f^* is not accurately known due to the noise (Letham et al., 2019). A common approach for handling noisy observations is to use arbitrary strategies that estimate an efficient representative of f^* rather than using the noisy minimum directly (Picheny et al., 2013). EI with plug-in Letham et al. (2019) and Vazquez et al. (2008) is one such method, which uses T as an alternative to f^*, as follows:

$$\alpha_{EI}(z_t|T) = \mathbb{E}[\max(0,\ T - y_t)|y_t \sim \mathcal{GP}(z_t|\mathcal{D}_{t-1})] \tag{8}$$

Here, T is computed as the minimum value of the posterior mean, μ, from the GP model over previously observed data points, $\mathcal{D}_{t-1}$:

$$T = \min_{k\in\mathcal{D}_{t-1}} (\mu_{t-1}(z_k)) \tag{9}$$

By using the GP posterior mean, T provides a smoother and more reliable estimate of the current best solution, especially in the presence of noisy observations. However, this method has a drawback in that it ignores the noise of future observation. That is, *the improvement is defined and its expectation is calculated as if the next evaluation is deterministic* (Picheny et al., 2013). Therefore, to account for subsequent noisy observations, replacing the current best solution method is not sufficient.

To adopt a noise-robust acquisition function, NoRTune employs an augmented EI (AEI) acquisition function (Huang et al., 2006), calculated by multiplying EI by a penalty value (Picheny et al., 2013). It is calculated as follows:

$$\alpha_{AEI}(z_t|T^*) = \alpha_{EI}(z_t|T^*) \times \left(1 - \frac{\sigma_\epsilon}{\sqrt{\sigma_{t-1}^2(z_t) + \sigma_\epsilon^2}}\right) \tag{10}$$

where σ_ϵ^2 is the noise variance modeled by the GP from accumulated observations. This penalty reduces the original EI values when the observation noise is strong. Notably, the AEI value equals the EI value when the noise level is zero. Hence, the dynamic AEI adjustments ensures the acquisition function remains robust under varying noise conditions. AEI's effective best solution is T^*, which is less sensitive to noisy observations than T, and it incorporates model uncertainty based on the standard deviation, σ_{t-1}:

$$T^* = \min_{k\in\mathcal{D}_{t-1}} (\mu_{t-1}(z_k) + \lambda\sigma_{t-1}(z_k)) \tag{11}$$

where λ is a hyperparameter, recommended to be set to one (Picheny et al., 2013).

In summary, the AEI recommends a candidate solution that either has a low noise level or performs optimally in terms of Spark performance, regardless of noise. Additionally, for robust tuning with noisy observations, NoRTune selects the best solution based on T^* throughout all iterations.

3.5. Discussion: Distinct features of NoRTune

To further highlight the distinct aspects of NoRTune, Table 2 summarizes the BO frameworks and acquisition functions employed by prior auto-tuning Spark studies. Although LlamaTune was originally developed for PostgreSQL rather than Spark, it is included because it serves as a major baseline in our experiments. Most methods adopt standard (vanilla) BO or minor extensions. For example, ROBOTune employs BO with Hedge, a strategy that adaptively selects from a portfolio of multiple acquisition functions (e.g., EI, lower confidence bound (LCB), and probability of improvement (PI)). LOCAT utilized a datasize-aware BO (DABO) model that augments the input space of GPs with workload data sizes, combined with EI with Markov chain Monte Carlo (EI-MCMC) techniques. Rover applies BO with expert rule trees, dynamically guiding candidate selection when the surrogate model underperforms. TurBO adopts BO with an adaptive pseudo-point

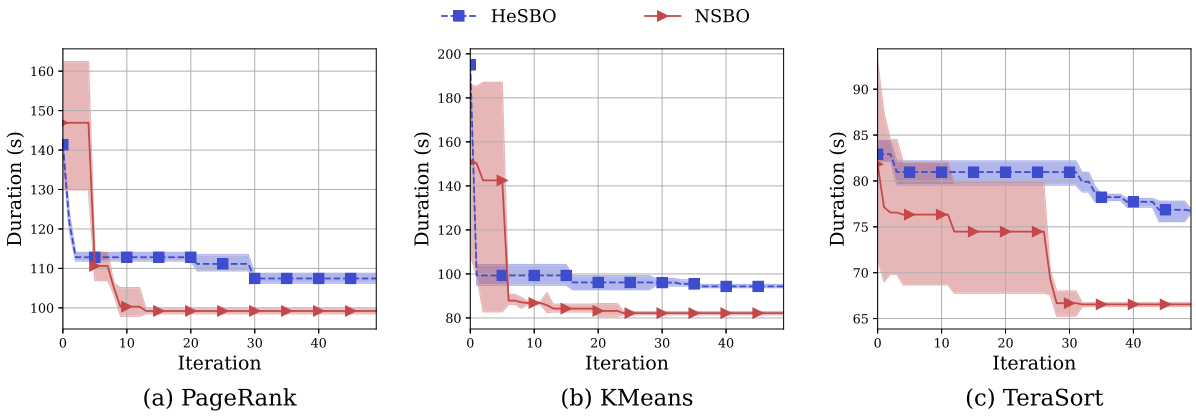

Fig. 5. Minimum durations at each iteration for HeSBO and NSBO on three workloads: PageRank, KMeans, and TeraSort.

mechanism, and Online-Tune applies an adaptive subspace method to control the search space, using EI with constraints (EIC).

In contrast, NoRTune introduces two key innovations. First, when handling high-dimensional parameter spaces, only LlamaTune and NoRTune operate over the full original configuration space rather than tuning a selected subset of parameters. However, LlamaTune employs HeSBO, which projects the space into a fixed low-dimensional subspace and suffers from uncertainty about whether the active subspace is included. NoRTune addresses this limitation by leveraging NSBO, which dynamically increases the target dimensionality throughout tuning, ensuring that the active subspace is eventually encompassed within the search space, given the evaluation budget.

To assess the effectiveness of NSBO, we conducted a direct comparison between NSBO and HeSBO. For HeSBO, the target dimensionality was fixed at 20, whereas NSBO was configured using the NoRTune settings under the experimental conditions described in Section 4.1. The results, presented in Fig. 5, show that NSBO consistently outperforms HeSBO across all workloads. Specifically, it converges more rapidly to high-quality configurations, whereas HeSBO tends to get trapped in local optima rather than finding the global best. This advantage arises because NSBO's dynamic subspace expansion increases the likelihood of capturing the active subspace during tuning, thereby enabling the discovery of globally optimal configurations.

Second, regarding system performance noise, most extant approaches rely on acquisition functions originally designed for noise-free settings. In contrast, NoRTune employs the noise-robust AEI acquisition function, which explicitly incorporates observation noise variance into the candidate selection process. This enhancement improves the reliability of tuning outcomes in noisy environments. A detailed evaluation of AEI's impact is provided in Section 4.4. In summary, by simultaneously addressing the challenges of unknown active subspace dimensionality and performance variability, NoRTune offers a more dependable and scalable approach to tuning Spark configurations than existing methods.

4. Evaluation

In this section, we present NoRTune's performance evaluation results to validate its contributions. We first compared NoRTune to state-of-the-art algorithms to evaluate its tuning performance. Second, we analyzed its consistency in suggesting promising configurations during tuning sessions to evaluate its reliability. Third, we conducted an ablation study to confirm the contributions of each component. Finally, we performed scalability experiments to evaluate its ability to handle larger datasets and scale to resource changes.

Table 2
Comparison of auto-tuning models based on their BO strategy and acquisition function.

Model	BO strategy	Acquisition function
TuneFul (Fekry et al., 2020)	Vanilla BO	EI
ROBOTune (Khan and Yu, 2021)	BO with Hedge	Choice from {EI, LCB, PI}
LOCAT (Xin et al., 2022)	DABO	EI-MCMC
Rover (Shen et al., 2023)	Expert-assisted BO	EI
TurBO (Dou et al., 2023)	BO-AdaPP	EI
Online-Tune (Li et al., 2023)	BO with Adaptive Space	EIC
LOFTune (Li et al., 2025a)	Vanilla BO	Hybrid sampling
LlamaTune (Kanellis et al., 2022)	HeSBO	EI
NoRTune (ours)	**NSBO**	**AEI**

4.1. Experiment setups

Environment. The experiments were conducted in a cloud environment on the Google Cloud Platform (GCP) using a total of four instances. The setup included a master node configured with 4 vCPUs (2cores), 15 GB RAM (n1-standard-4), and a 100 GB HDD, along with three slave nodes, each configured with 8 vCPUs (4 cores), 16 GB RAM (custom-8-16384-ext), and a 100 GB HDD. We employed Spark 3.1.3, Hadoop 3.2.3, Scala 2.12.14, and OpenJDK 1.8.0, using the pre-built Dataproc image version 2.0 available on GCP.

Workloads. We benchmarked Spark performance using HiBench (Huang et al., 2010), a comprehensive big data benchmark suite designed to evaluate frameworks like Hadoop and Spark. We selected eight representative workloads from four categories for our experiments: Micro Benchmark (WordCount and TeraSort), Machine Learning (Bayes and KMeans), SQL (Scan, Join, and Aggregation), and Web-Search Benchmark (PageRank). Input data sizes were categorized into six levels: tiny, small, large, huge, gigantic, and big data. Considering our hardware environment, we selected the large data size to better distinguish NoRTune from the baselines. The HiBench version used in the experiments was 7.1.1.

Spark parameters. Based on previous Spark tuning research (Shen et al., 2023; Xin et al., 2022; Fekry et al., 2020; Dou et al., 2023), we selected 45 tuning parameters (Table 3) after empirically evaluating their impact on the Spark performance and confirming their availability in our version. The ranges of values for these parameters were defined according to our environment. The selected Spark parameters include eight Boolean, two categorical, four continuous, and 31 numerical parameters. See Spark (2024) for parametric details.

NoRTune Implementation. For our hyperparameters, the number of new bins per split was set to $b = 1$, and the initial sample size was $n_{init} = 5$. Following the evaluation setup used with Rover (Shen et al., 2023), which benchmarks tuning methods on public workloads,

Table 3
Information of selected Spark parameters.

#	Configuration Parameters	Range	Default
1	spark.broadcast.compress	[true, false]	TRUE
2	spark.kryo.referenceTracking	[true, false]	TRUE
3	spark.memory.offHeap.enabled	[true, false]	FALSE
4	spark.rdd.compress	[true, false]	FALSE
5	spark.shuffle.compress	[true, false]	TRUE
6	spark.shuffle.spill.compress	[true, false]	TRUE
7	spark.speculation	[true, false]	FALSE
8	spark.sql.inMemoryColumnarStorage.compressed	[true, false]	TRUE
9	spark.memory.fraction	[0.5, 0.9]	0.6
10	spark.memory.storageFraction	[0.5, 0.9]	0.5
11	spark.speculation.multiplier	[1, 5]	1.5
12	spark.speculation.quantile	[0, 1]	0.75
13	spark.broadcast.blockSize	[1 m, 16 m]	4
14	spark.default.parallelism	[8, 100]	8
15	spark.driver.cores	[1, 16]	1
16	spark.driver.memory	[1024 m, 4096 m]	1024
17	spark.driver.memoryOverhead	[0, 10240]	384
18	spark.executor.cores	[1, 16]	1
19	spark.executor.instances	[2, 112]	2
20	spark.executor.memory	[1024 m, 4096 m]	1024
21	spark.executor.memoryOverhead	[0 m, 8192 m]	384
22	spark.io.compression.lz4.blockSize	[16k, 96k]	32
23	spark.io.compression.snappy.blockSize	[16k, 96k]	32
24	spark.io.compression.zstd.bufferSize	[16k, 96k]	32
25	spark.io.compression.zstd.level	[1, 5]	1
26	spark.kryoserializer.buffer	[2k, 128k]	64
27	spark.kryoserializer.buffer.max	[8 m, 128 m]	64
28	spark.locality.wait	[1 s, 6 s]	3
29	spark.memory.offHeap.size	[1 m, 1024 m]	1
30	spark.network.timeout	[60 s, 300 s]	120
31	spark.reducer.maxSizeInFlight	[2 m, 128 m]	48
32	spark.scheduler.revive.interval	[1 s, 5 s]	1
33	spark.shuffle.accurateBlockThreshold	[100 m, 1000 m]	100
34	spark.shuffle.file.buffer	[2k, 128k]	32
35	spark.shuffle.io.numConnectionsPerPeer	[1, 5]	1
36	spark.shuffle.service.index.cache.size	[10 m, 2048 m]	100
37	spark.shuffle.sort.bypassMergeThreshold	[100, 1000]	200
38	spark.speculation.interval	[10, 1000]	100
39	spark.sql.autoBroadcastJoinThreshold	[1 m, 100 m]	10
40	spark.sql.files.maxPartitionBytes	[16 m, 4096 m]	128
41	spark.sql.inMemoryColumnarStorage.batchSize	[5000, 20000]	10 000
42	spark.sql.shuffle.partitions	[100, 1000]	200
43	spark.storage.memoryMapThreshold	[2 m, 500 m]	2
44	spark.io.compression.codec	[snappy, lzf, lz4]	lz4
45	spark.serializer	[JavaSerializer, KryoSerializer]	JavaSerializer

including HiBench, we set the evaluation budget to $m_D = 50$ for all experiments, including baselines. This decision ensured alignment with previous tuning studies while reflecting practical resource constraints. The initial target dimensionality, d_0, must be at least equal to or greater than the number of data types; hence, we set it to $d_0 = 5$.

Objective. The objective of NoRTune is to find a configuration that minimizes the execution time (i.e., duration) of workloads. This objective can easily be adapted to other performance metrics.

Baselines. Because NoRTune is designed for resource-efficient tuning, we selected baseline methods that do not assume prior observations, such as by collecting data from previous tuning sessions or other workloads for the initial tuning session. Models relying on transfer learning or the accumulation of prior data were excluded, as they overlook the common infeasibility of collecting prior observations. Among the selected baselines, the first two were as follows: **Random search (Random)** (Zhang et al., 2022b)—a simple tuner that explores each dimension of parameters uniformly at random, and **LlamaTune**—a state-of-the-art method using subspace-based BO. As LlamaTune is agnostic to the underlying optimizer, we used both BO and the sequential model-based algorithm configuration (SMAC) method (Hutter et al., 2011) as optimizers. Additionally, as LlamaTune requires searching for the optimal target dimensionality, we empirically determined this to be 20 by comparing four different cases: {5, 10, 20, 30}. The remaining baseline was **CSAT** (Li et al., 2025b)—a configuration structure-aware tuner (CSAT) that uses an adaptive network-based fuzzy inference system and GP regression to model the latent structure of configuration spaces. In our experiments, we followed the original CSAT setup by tuning the 13 parameters recommended by the authors, as preliminary tests showed that tuning all 45 would result in degraded performance.

Additional Details. Each configuration was evaluated three times per iteration to account for noise. If a generated configuration failed to execute Spark during a tuning session due to incorrect parameter dependencies, we recorded the large-value results and replaced them with the maximum value from current observations prior to fitting the GP surrogate model.

4.2. Tuning evaluation

We evaluated the tuning performance of NoRTune by comparing the final optimized durations with those of the baselines. To reflect their strategy for handling noise, the optimal durations for Random, LlamaTune (SMAC), LlamaTune (BO), and CSAT were selected as the minimum values during their respective tuning sessions. In contrast, the results for NoRTune were determined using the effective best solution described in Section 3.4. Notably, we re-evaluated the best configurations for all models following their tuning sessions to account for variability in performance outcomes.

The results for all eight workloads are described in Table 4, where we report the average and [5%, 95%] confidence interval values based

Table 4
Relative performance improvement ratio compared to Random.

Models	PageRank		Aggregation		Scan		Join	
	Average	[5%, 95%] CI	Average	[5%, 95%] CI	Average	[5%, 95%] CI	Average	[5%, 95%] CI
LlamaTune (SMAC)	−1.17%	[−4.70%, 1.15%]	−7.39%	[−8.31%, −5.71%]	−2.97%	[−3.45%, −2.14%]	−2.46%	[−4.25%, −1.44%]
LlamaTune (BO)	12.96%	[7.62%, 17.26%]	−0.48%	[−2.76%, 1.11%]	**−2.65%**	**[−4.05%, −1.52%]**	−2.08%	[−4.59%, 1.16%]
CSAT	−14.79%	[−22.30%, −6.74%]	−1.78%	[−2.39%, −1.08%]	−3.57%	[−3.98%, −3.34%]	−3.38%	[−6.95%, −0.11%]
NoRTune	**13.18%**	**[11.78%, 15.24%]**	**−0.14%**	**[−1.44%, 0.91%]**	−2.72%	[−3.60%, 3.52%]	**−1.93%**	**[−3.25%, −0.11%]**
Models	KMeans		WordCount		TeraSort		Bayes	
	Average	[5%, 95%] CI	Average	[5%, 95%] CI	Average	[5%, 95%] CI	Average	[5%, 95%] CI
LlamaTune (SMAC)	16.53%	[7.61%, 27.55%]	2.66%	[1.17%, 5.30%]	2.42%	[1.08%, 4.16%]	**1.87%**	[1.37%, 2.59%]
LlamaTune (BO)	10.52%	[−10.61%, 22.64%]	5.47%	[2.97%, 8.51%]	−1.05%	[−4.62%, 2.71%]	−0.38%	[−2.24%, 2.80%]
CSAT	12.59%	[2.13%, 18.58%]	4.20%	[3.49%, 4.83%]	3.85%	[3.34%, 4.68%]	0.57%	[−2.17%, 3.74%]
NoRTune	**19.60%**	**[8.79%, 28.94%]**	**7.46%**	**[4.91%, 9.29%]**	**8.53%**	**[7.03%, 10.90%]**	1.27%	[0.25%, 1.86%]

on repeated evaluations. The results show the relative improvement ratios of LlamaTune models, CSAT, and NoRTune compared with Random. We made the following observations.

NoRTune outperformed LlamaTune (SMAC) by an average of 4.49%, with improvements reaching up to 14.13%. It outperformed LlamaTune (BO) by an average of 2.83% and up to 9.44%. It also outperformed CSAT by an average of 5.64%, with improvements reaching up to 24.17%. Compared to Random, NoRTune achieved an average improvement of 5.68%, reaching up to 19.60%. These results show that NoRTune consistently finds high-performance configurations under the same evaluation budgets as the comparison models.

We also found that LlamaTune performs similarly to or worse than Random on most workloads. For example, LlamaTune (SMAC) showed a significant improvement only on KMeans, with an average improvement of 16.53%, whereas LlamaTune (BO) outperformed Random on three workloads: PageRank (12.96%), KMeans (10.52%), and WordCount (5.47%). This is because LlamaTune reduces the parameter space through fixed-dimensional linear embedding, maintaining the same one-to-many mapping throughout the tuning session, which hinders more complex exploration and limits the ability to find the global optimum. Furthermore, since the proper low dimensionality must be determined experimentally, LlamaTune requires substantial time and resources to ensure optimal performance.

CSAT outperformed Random on three workloads: KMeans (12.59%), WordCount (4.20%), and TeraSort (3.85%). However, for PageRank, CSAT performed worse than Random, showing a degradation of −14.79%. This performance decline is likely due to the challenge of learning complex hidden configuration structures, which CSAT attempts to model through accumulated observations. Notably, CSAT generally requires a significantly larger evaluation budget (e.g., 100 to 300 iterations), as revealed in its original study. Under our constrained setting with limited iterations, CSAT was unable to adequately explore or model these relationships, thus failing to capture intricate parameter interactions. We further analyzed this limitation by reviewing the tuned parameters given in Section 4.6.

Based on the relative improvement ratios compared with Random, NoRTune delivered notable improvements over baselines such as PageRank (13.18% average improvement), KMeans (19.60%), WordCount (7.46%), and TeraSort (8.53%). These improvements were enabled by NSBO, whose nested structure forces broader and more flexible exploration of high-dimensional parameter spaces, resulting in more robust performance.

However, for Bayes, NoRTune showed a slight improvement, with an average improvement of 1.27%, lower than LlamaTune (SMAC) (1.87%). For Spark SQL workloads (Aggregation, Scan, and Join), NoRTune, LlamaTune, and CSAT failed to find a better configuration than Random, with LlamaTune (SMAC) showing an average decrease of 7.39%. This behavior can be attributed to the configuration insensitivity occasionally exhibited by SQL workloads (Xin et al., 2022). Specifically, SQL-based operations (e.g., scan, join, and aggregate) become sensitive to configurations primarily when processing large shuffle volumes, typically seen in large datasets (e.g., 50–100 GB). In our study, dataset size was constrained to approximately 300 MB due to hardware limitations, resulting in smaller shuffle volumes and minimal performance variations across models. Consequently, aggregation showed a performance differences of only −0.14% relative to Random, whereas scan and join exhibited similar trends, with average differences around -2.xx%. Another contributing factor is that Spark SQL workloads inherently benefit from extensive internal optimizations, such as the Catalyst query optimizer (Armbrust et al., 2015) and the Tungsten execution engine (Xin and Rosen, 2015), which reduces sensitivity to external configuration tuning.

To analyze the performance noise of the models, we calculated the coefficient of variation (CV) to assess the variability of repeated evaluations. CV, the ratio of the standard deviation to the mean, provides a normalized measure of dispersion and is widely used to quantify performance variability (Xin et al., 2022). In this context, a lower CV indicates that the model produces more stable configurations with less performance noise. The results are shown in Fig. 6. NoRTune recorded lower CV values on most workloads, with an average rank of 1.6 across all, demonstrating the best noise-robust performance. In particular, with PageRank and KMeans, NoRTune achieved significantly lower CV values, achieved by leveraging AEI as its acquisition function, thus locating configurations that achieve high performance while mitigating the impact of performance variability.

4.3. Efficiency evaluation

In this experiment, we evaluated the tuning efficiency by analyzing the quality of candidate configurations, which is crucial for the convergence of an optimization framework (Wilson et al., 2018). For quantification, we used the cumulative regret metric (Moriconi et al., 2020; Srinivas et al., 2010), which measures the difference between the selected configuration at each iteration and the optimal solution over iterations, calculated as follows:

$$R = \Sigma_{t=1}^{m_D} \log_{10} |f(x^*) - f(x_t)| \tag{12}$$

where $f(x^*)$ is the final best solution on the tuning session, and $f(x_t)$ is the best solution (i.e., minimum result) at iteration t. We first computed the immediate logarithmic regret and summed them to obtain the cumulative regret.

For a fair comparison, we determined the same $f(x^*)$ for all models using the best results from NoRTune. This metric measures how quickly the model approaches the best solution by consistently returning promising configurations through its acquisition function. As a model rapidly converges to an optimal solution, the cumulative regret decreases, resulting in a graph with a slowly increasing or even decreasing slope. In contrast, a steep slope in the graph indicates that the model is consistently failing to find better solutions, suggesting persistent optimization inefficiencies. Therefore, a model with a relatively flat slope in cumulative regret over time is more effective in achieving optimal performance. In some cases, the graph showed negative values

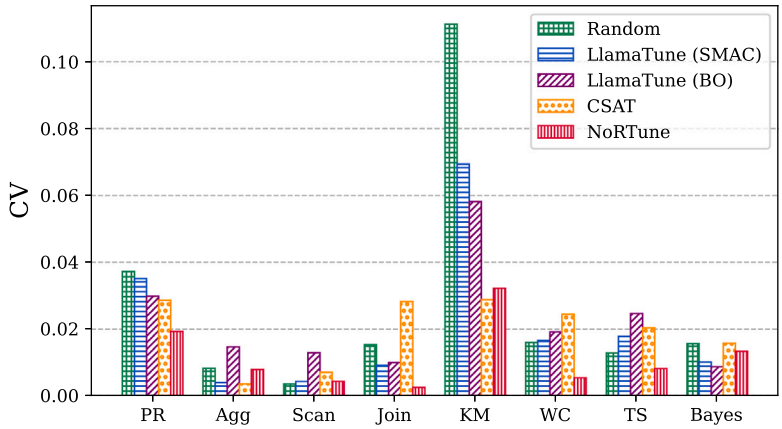

Fig. 6. Analysis of performance noise using the coefficient of variation on PageRank (PR), Aggregation (Agg), Scan, Join, KMeans (KM), WordCount (WC), TeraSort (TS), and Bayes workloads.

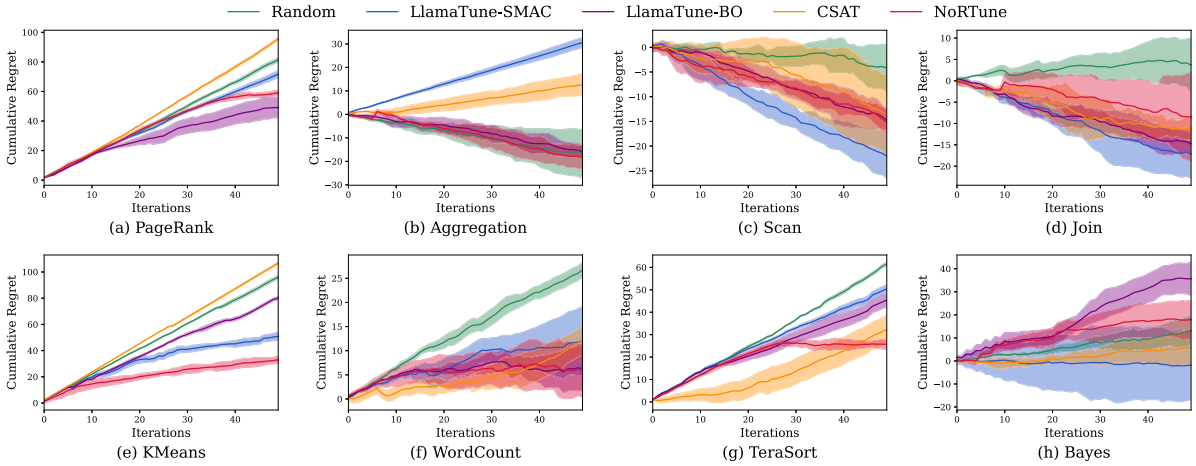

Fig. 7. Comparison of tuning efficiency. The *y*-axis shows the cumulative regret calculated at each iteration.

from the use of logarithmic functions in the calculation of cumulative regret. When the error between two results fell between zero and one, the logarithmic transformation produces a negative value. A negative slope in the graph indicates that the best solution remains consistent throughout all iterations, resulting in similar values. This phenomenon was observed only in workloads where the performance was comparable to that of random tuning.

Fig. 7 shows that NoRTune exhibited faster convergence to the best solution compared with other baselines. Notably, NoRTune quickly descended to low regret values on half the workloads: Aggregation, KMeans, WordCount, and TeraSort. We observed that the convergence performance of the models corresponded to their tuning performances, as shown in Table 4. For example, on the WordCount workload, Random showed the slowest convergence and the worst performance, whereas on the Aggregation workload, LlamaTune (BO) and Random showed similar performance levels. However, NoRTune exhibited a worse slope despite outperforming LlamaTune (BO) on the PageRank workload, indicating slower convergence towards the optimal configuration. This discrepancy arises because LlamaTune (BO) discovered a more promising configuration earlier, achieving better results until the 40th iteration, as illustrated in Fig. 8(a). At the 40th iteration, NoRTune identified a superior configuration, resulting in a sudden flattening of the slope. Similarly, for the TeraSort workload, the cumulative regret of NoRTune stabilized after iteration 30, indicating that it found a near-optimal configuration at that point, as shown in Fig. 8(b). However, CSAT displayed patterns that differed from the other models. That is, although it had the slowest convergence on the KMeans workload, it ultimately outperformed Random in overall performance. As shown in Fig. 8(c), CSAT identified its best configuration relatively early—around the 27th iteration. However, it failed to improve beyond that, resulting in a flat regret curve. In contrast, Random continued to make gradual improvements over time, producing a lower cumulative regret slope despite ultimately achieving an overall performance worse than CSAT.

LlamaTune showed rapid improvement on PageRank, Aggregation, WordCount, and TeraSort when using the BO optimizer and on Scan, Join, KMeans and Bayes when using the SMAC optimizer. However, results from the remaining workloads showed that LlamaTune failed to find promising configurations because, although it successfully reduced the configuration space using subspace-based BO, it did not address the variability of Spark performance within the tuning framework. LlamaTune simply minimized the observations to select the optimal configuration, whereas NoRTune considered the noise variance of Spark performance. These results confirm that NoRTune consistently returns promising candidate configurations for all workloads, improving convergence to the optimal solution and achieving better tuning efficiency.

4.4. Ablation study

To better understand the contribution of each core component of NoRTune, we conducted an ablation study that isolated and evaluated the effects of NSBO and AEI. The models were evaluated on the PageRank, TeraSort, and KMeans workloads, selected based on previous

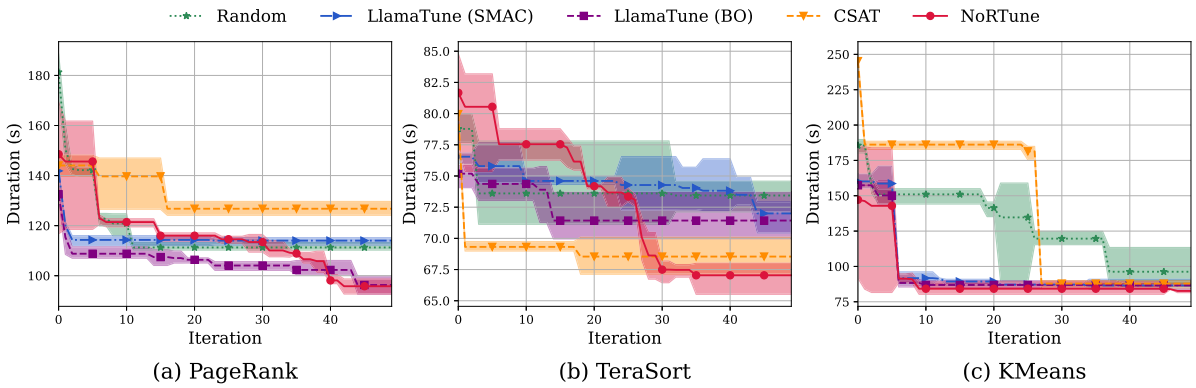

Fig. 8. Cumulative minimum duration over iterations for each baseline. The y-axis represents the best duration achieved at each iteration.

Table 5
Relative performance improvement ratio of ablation modules compared to Random.

Module		PageRank	KMeans	TeraSort
NSBO	AEI	Average [5%, 95%] CI	Average [5%, 95%] CI	Average [5%, 95%] CI
–	–	0.39% [−4.13%, 4.10%]	4.14% [−18.09%, 17.92%]	0.78% [−2.01%, 2.63%]
–	✓	0.09% [−3.94%, 5.06%]	6.64% [−9.17%, 21.13%]	−6.17% [−8.35%, −4.95%]
✓	–	10.60% [7.24%, 16.74%]	17.53% [4.25%, 26.81%]	2.84% [0.51%, 5.88%]
✓	✓	**13.18% [11.78%, 15.24%]**	**19.60% [8.79%, 28.94%]**	**8.53% [7.03%, 10.90%]**

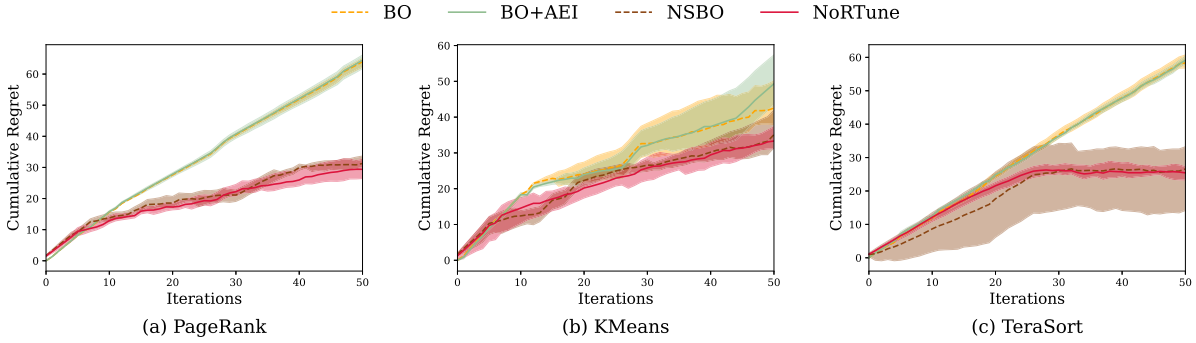

Fig. 9. Cumulative regrets from the ablation study.

evaluations that indicated significant parameter tuning performance improvements.

The results are shown in Table 5 and Fig. 9. Where we can see that the absence of NSBO had a significant negative impact on tuning efficiency. In Fig. 9, the slopes of BO and BO + AEI indicate that they often failed to find promising configurations. This is because, although BO + AEI considers Spark performance noise, the curse of dimensionality renders the optimization convergence more difficult and slower. Conversely, NSBO and NoRTune exhibited competitive optimization performance, as NSBO reduced the search space to an active subspace that sufficiently represented the objective function. This dimensionality reduction enables improved exploration and faster convergence.

Although NSBO excelled in reducing the search space, its standalone performance was affected by noise in candidate configurations. As depicted in the TeraSort workload, NSBO's noise handling resulted in wider confidence intervals and larger shaded regions in Fig. 9. However, integrating AEI with NSBO addressed these limitations as AEI balances exploration and exploitation in the reduced subspace, improving robustness against performance noise. This synergy between NSBO and AEI is the key to NoRTune's superior performance. Experimental results validated that combining these components enables NoRTune to outperform standalone methods. For example, as NSBO ensures efficient exploration, AEI enhances tuning reliability in noisy environments. Together, these components address the challenges related to dimensionality and noise, achieving reliable and efficient tuning outcomes across all evaluated workloads.

4.5. Scalability

In the previous experiments, we used workloads with a *large* input data size on four instances. To evaluate the scalability of NoRTune, we increased both the data size and the number of instances. We used *huge* input size, increasing the data volume tenfold, and expanded the Spark cluster from 4 to 10 instances, with nine slave nodes configured in the same environment, as described in Section 4.1. We also modified parameter ranges to match the hardware specifications, adjusting settings such as *spark.default.parallelism*, *spark.driver.memory*, and *spark.memory.offHeap.size*. These experiments were conducted on two workloads: WordCount and KMeans. The results are shown in Fig. 10.

We noted that NoRTune found better configurations than those tuned by Random, LlamaTune (SMAC), LlamaTune (BO), and CSAT, achieving improvements of up to 3.27%, 3.73%, 4.24%, and 2.35%, respectively, on WordCount, and up to 6.96%, 7.90%, 9.49%, and 7.15%, respectively, on KMeans. These results demonstrate that NoRTune is effective across various environments and data sizes, consistently achieving near-optimal performance. Additionally, we measured the tuning times of each model within the same evaluation budget to evaluate tuning overhead. NoRTune showed either faster or comparable tuning times than the baselines, with improvements of 1.96%, 1.44%, 0.15%, respectively, 2.77% on WordCount, and 7.79%, 5.50%, 6.09%, respectively, 8.65% on KMeans. These results indicate that NoRTune

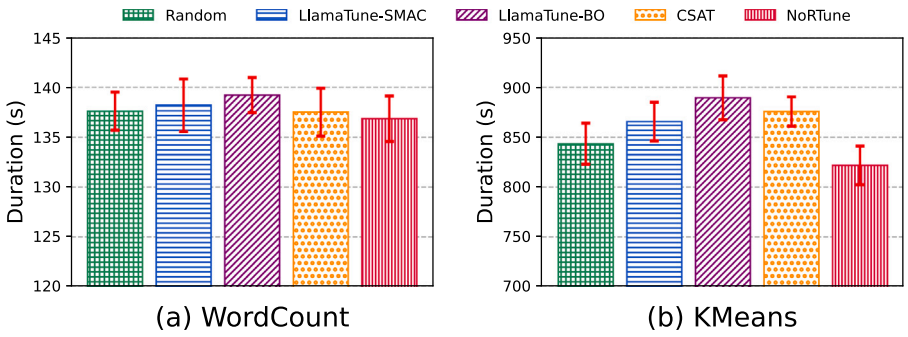

Fig. 10. Scalability evaluation with a huge input size on a ten-node Spark cluster.

achieves competitive computational efficiency, demonstrating that it is not computationally expensive compared with other models.

4.6. Tuning validation

To practically validate the tuning performance of NoRTune, we analyzed the tuned parameter results from two perspectives: (1) how the parameters were tuned according to the specific characteristics of each workload and (2) how the tuning results compared to the baselines. For each case, we selected five key parameters that significantly influenced performance and conducted a detailed analysis of their impact.

Tuned parameters moderated by workload characteristics. We selected two workloads, PageRank and KMeans, which exhibited the highest noise levels based on the CV values in Fig. 6. PageRank represents a graph processing workload characterized by frequent shuffle operations as it computes webpage importance calculation using iterative data transfers between nodes. Owing to the heavy shuffle workload, optimizing disk and network I/O efficiency is crucial for alleviating bottlenecks and ensuring smooth data movement across nodes. This directly improves execution performance. KMeans performs clustering through iterative computations and data redistributions. Unlike PageRank, KMeans involves repeated calculations on large datasets, making parallel processing capabilities essential to performance. Effectively balancing resource allocation and leveraging parallelism ensures that iterative tasks are efficiently distributed and executed. The tuning results for key parameters are shown in Fig. 11 and analyzed as follows:

- spark.executor.instances: This parameter determines the number of Spark executor processes for task execution. For PageRank, in which maximizing parallelism is critical due to its shuffle-intensive workload, the value was set to 112, resulting in significant performance improvements. For KMeans, a balanced value of 71 was chosen to effectively distribute resources without overloading the system.
- spark.shuffle.compress: This Boolean parameter enables compression of intermediate shuffle files. Given PageRank's shuffle-heavy nature, enabling compression (set to 1 or "True") reduced network I/O overhead and improved execution time. Similarly, for KMeans, enabling compression managed intermediate data efficiently and minimized network overhead.
- spark.memory.fraction: This parameter defines the memory allocation ratio between execution and storage. PageRank prioritized execution memory by setting the value to 0.9, thereby optimizing compute-intensive operations. For KMeans, a balanced ratio of 0.73 was chosen to ensure sufficient caching capacity while supporting efficient execution.
- spark.default.parallelism: This parameter specifies the number of partitions for shuffle operations. In PageRank, a lower value of eight minimized shuffle delays and mitigated network bottlenecks through efficient I/O. Conversely, KMeans required a higher value of 76 to improve task parallelism and fully utilize the compute resources.
- spark.broadcast.blockSize: This parameter sets the block size for broadcasted data. For PageRank, which processes large broadcast datasets, a larger block size of 16 MB reduced data transfer overhead. KMeans, dealing with smaller datasets, used a smaller block size of 4 MB to improve memory efficiency.

Tuned parameters moderated by baselines. We next focus on PageRank to investigate how NoRTune and the baselines differ in their parameter tuning results, as illustrated in Fig. 12. The analysis follows:

- spark.sql.files.maxPartitionBytes: This parameter determines the size of partitions, defining how much data are processed at once during shuffle operations. NoRTune set a small value to 16 MB, resulting in smaller partitions and more frequent shuffle operations, thus reducing the size of individual transfers, minimizing the risk of network congestion, and ensuring stable performance for the shuffle-intensive PageRank workload. Conversely, the LlamaTune models chose a large value of 2166 MB, reducing shuffle frequency and improving I/O efficiency. However, the larger partitions increased the risk of network congestion, which can cause PageRank performance to suffer. Because CSAT does not treat this parameter as tunable, we report its default value of 128 MB. Although this setting is generally reasonable, it was not fully optimized for minimizing shuffle overhead.
- spark.shuffle.file.buffer: This parameter specifies the buffer size for disk I/O during shuffle operations. NoRTune used a small buffer size of 2 kB, which could have led to frequent I/O operations. However, because the smaller buffer size avoided overwhelming the I/O system and prevented bottlenecks due to NoRTune's parameter-limited parallelism. LlamaTune (BO) adopted a moderate buffer size to 10 kB, thereby achieving improved I/O efficiency. However, LlamaTune (SMAC) set a larger buffer size of 14 kB, leading to excessive memory usage and I/O inefficiency, ultimately degrading performance. CSAT used a small buffer size of 5 kB, which may be acceptable in other scenarios. However, when paired with very high parallelism, this setting increased shuffle I/O overhead by triggering frequent small disk operations, ultimately contributing to degraded performance.
- spark.driver.memoryOverhead: This parameter defines the additional memory allocated for the driver process. NoRTune set the value to 0 MB because PageRank does not demand significant driver memory, thus ensuring efficient resource allocation to executors. Similarly, LlamaTune (BO) used an appropriate overhead value of 624 MB to maintain stability without resource waste. Conversely, LlamaTune (SMAC) allocated an excessive value of 5386 MB, leading to inefficient resource usage and unnecessary memory overhead. For CSAT, this parameter was not designated as tunable; hence, we report the default value of 384 MB. This default setting was sufficient and did not significantly affect workload performance.
- spark.executor.cores: This parameter determines the number of central processing unit (CPU) cores assigned to each executor.

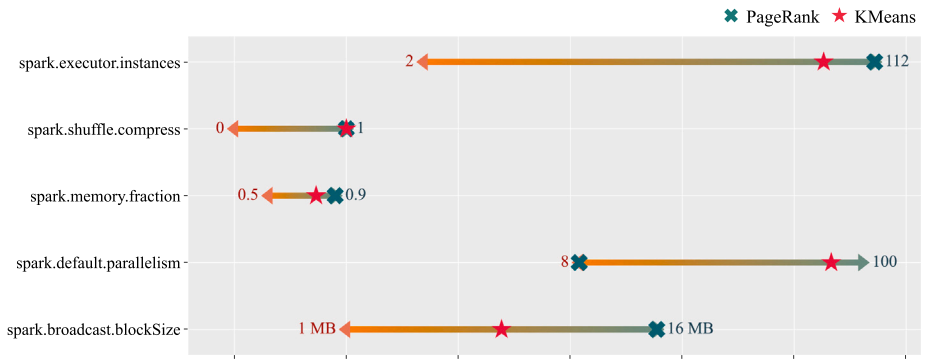

Fig. 11. Tuned parameter values of NoRTune for PageRank and KMeans.

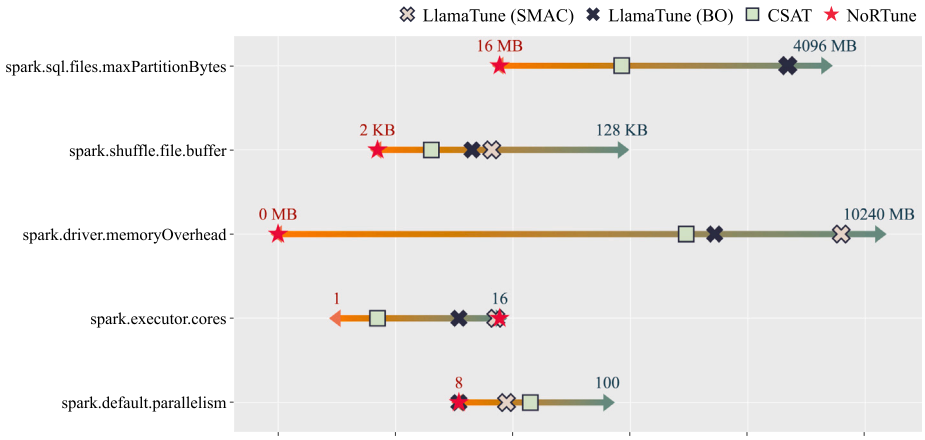

Fig. 12. Tuned parameter values of NoRTune and baselines for PageRank.

NoRTune set a high value to 16 but mitigated resource contention by reducing spark.default.parallelism, ensuring that network and CPU resources were utilized efficiently without overloading the system. LlamaTune (BO) opted for a smaller value of eight, which improved task distribution and I/O efficiency. Conversely, LlamaTune (SMAC) set a similarly high value of 15, amplifying network bottlenecks, which led to performance degradation due to excessive parallelism. CSAT assigned only two cores per executor, resulting in an excessive number of small executors. This increased scheduling and management overhead, severely degrading overall system efficiency.

- spark.default.parallelism: This parameter controls the degree of parallelism by specifying the number of partitions for shuffle operations. Both NoRTune and LlamaTune (BO) set a low value of eight to limit parallelism, effectively reducing network data transfer overhead and mitigating bottlenecks. In particular, NoRTune aligned this value with spark.executor.cores set to 16, achieving consistent and stable performance by efficiently utilizing resources. Conversely, LlamaTune (SMAC) increased the value to 18, leading to excessive parallelism and network I/O congestion, which significantly degraded performance. CSAT configured a higher value of 27, which overwhelmed the network with excessive shuffle tasks and caused severe performance degradation.

In these experiments, our performance analysis focused on Spark application runtimes, where tuned parameters were not confirmed. In the next experiment, we demonstrate that NoRTune effectively returns parameter values tailored to workload characteristics. Although baseline models primarily tune parameters to maximize immediate performance (e.g., parallelism), NoRTune considers both parallel execution, noise performance, and network bottlenecks. The results show that NoRTune suggests meaningful parameter values, ensuring both efficient resource utilization and stable performance.

4.7. Applicability beyond spark

NoRTune was proposed as a framework for tuning in Spark; however, it retains general-purpose applicability with respect to DBMSs. To validate this claim, we evaluated NoRTune outside of a Spark environment to assess its effectiveness and robustness across different systems. For this, we conducted experiments on PostgreSQL v13.18, a traditional relational DBMS, using hardware with an Intel i7-4700K CPU (28 vCPUs), 32 GB RAM, and a 250 GB SATA SSD. For parameter configuration, we followed the guidelines provided in Kanellis et al. (2022), which defined 112 tunable parameters. Benchmarking execution time was fixed to 5 min, following the same workload setup as LlamaTune, and NoRTune was implemented as described in Section 4.1. Two workload variants from the Yahoo! Cloud Serving Benchmark (YCSB) suite were used: YCSB-A (balanced 50%–50% read-write) and YCSB-B (read-heavy, 95% read and 5% write). The database was loaded with 40 GB data, and throughput served as the objective performance metric. Unlike Spark, where duration reflects overall performance, relational DBMSs prioritize throughput to maximize complex query efficiency within fixed execution times. The results presented in Fig. 13 show that NoRTune consistently outperformed baselines, demonstrating its effectiveness non-Spark environments. We found that NoRTune achieved an improvement of 9.68–35.81% over LlamaTune and a 14.54–26.71% improvement over CSAT across workloads. This highlights its robustness in handling expanded configuration spaces and adapting to new storage systems while maintaining reliable optimization performance. These findings validate NoRTune as a versatile and general-purpose tuning framework applicable to diverse database systems.

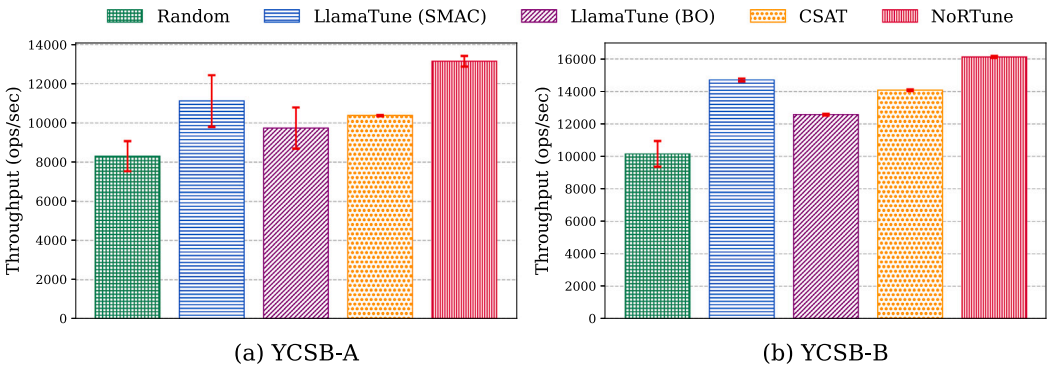

Fig. 13. Tuning evaluation on PostgreSQL.

5. Discussion

Our evaluation relied on experimentally validating the performance of NoRTune. Here, we discuss the theoretical and practical implications of our findings, highlighting key insights and potential contributions, particularly to Spark configurations.

Theoretical implications. This research offers two novel perspectives on Spark configuration tuning. First, for general cases, explicit parameter importance and dimensionality reduction decisions introduce additional resource costs beyond those of the tuning sessions, creating a challenge for efficient auto-configuration tuning. To quantify the influence of this problem, we reported the results of a case study in Section 2.4 based on LlamaTune, reporting the impacts of defining target dimensionality on tuning performance. We found that the optimal reduced dimensionality is not apparent, necessitating a time-consuming pre-decision step prior to the main tuning session. NoRTune addresses this issue by dynamically exploring the search space using nested subspaces, which grow from low- to high-dimensional input spaces, thus providing a novel resource-efficient tuning framework.

Secondly, we highlighted the significance of noise-aware optimization in real-world scenarios by integrating acquisition functions that explicitly model system performance variability. In practice, Spark performance noise arises from many factors, including network and disk I/O, garbage collection, and shared hardware in cloud environments. We investigated Spark performance noise with 10 different configurations on two workloads and confirmed that its magnitude varies across workloads in Section 2.4. Previous studies primarily mitigated noise effects by establishing robust experimental setups, such as repeating evaluations or averaging results. However, they relied on acquisition functions designed for noise-free environments, which are not suitable for real-world utility. Notably, the choice of acquisition function plays a critical role in BO convergence. To this end, we applied a noise-robust acquisition function, demonstrating its effectiveness in achieving reliable and robust tuning performance. The experimental results showed that NoRTune achieved less noisy performance than all baselines tested.

Overall, our approach is expected to inspire additional tuning research that ought to focus on configuration optimization as our work has mitigated the burden of trial-and-error pre-selection steps to address system performance variability.

Practical implications. NoRTune provides several practical and useful implications. First, its efficiency, robustness, and adaptability break through contemporary barriers to real-world configuration tuning. Traditional high-dimensional frameworks often depend on extensive preparatory steps, including collecting thousands of samples to assess parameter importance, gathering diverse workload samples for warm-start initializations, or determining the optimal dimensionality for search space reductions. These preparatory stages impose significant resource and time demands beyond the primary tuning sessions, presenting a critical barrier for resource-constrained and time-sensitive real-world applications. Conversely, NoRTune does not require these preparatory steps. With only parameter ranges and workload information, NoRTune achieves high performance for cold-start conditions, where no prior tuning data are available, facilitating robust tuning in dynamic workload scenarios, including workload variations and software version updates. This adaptability points to NoRTune being particularly valuable for environments requiring rapid and reliable tuning. To further emphasize NoRTune's practical efficiency, we compared its computational overhead and total tuning time to baseline methods. Table 6 presents the results across four workloads: PageRank, Aggregation, KMeans, and Bayes. The findings show that NoRTune delivers comparable or superior computational efficiency and tuning times, confirming that it not only eliminates the need for costly preparatory procedures but also maintains optimization efficiency during tuning, reinforcing its practicality for deployment in resource-constrained environments.

Secondly, a cluster computing system, Spark, frequently encounters variability in runtime performance due to network I/O, disk I/O, garbage collection, and shared hardware in cloud environments. Such variability can be mitigated by optimizing parameters. However, previous studies did not incorporate these performance variations into their tuning frameworks; instead, they ran multiple tuning sessions to average or a single tuning session, which led to unstable results. For example, a configuration with good average performance may be selected; however, it can yield results significantly below average due to performance noise, leading to substantial performance degradation and potential losses in real-world application environments. NoRTune, on the other hand, leverages performance variability data within its tuning framework, reducing variability and identifying configurations that consistently yield enhanced and reliable results.

Finally, we demonstrated the applicability of NoRTune beyond Spark environments. For tuning frameworks to be practically useful in real-world application environments, they must easily adapt to different workloads and systems. NoRTune achieves this by offering a generalizable structure that easily integrates with other cloud environments and DBMS platforms. We showed this capability via experiments with PostgreSQL, demonstrating NoRTune's applicability and versatility in a cross cross-platform, scalable tuning framework.

6. Conclusion

In this study, we proposed NoRTune, a noise-robust high-dimensional configuration tuning framework for Spark, addressing key challenges in BO tuning. By integrating NSBO and a noise-robust acquisition function, NoRTune eliminates the need for pre-selecting dimensionality, mitigates performance noise, and adapts to dynamic workloads with minimal preparatory steps. Comprehensive experiments validated its efficiency, reliability, and scalability across various Spark workloads, as well as its applicability to other systems (e.g., PostgreSQL). NoRTune advances high-dimensional optimization techniques

Table 6
Algorithmic overhead (Algorithm) and overall tuning time (Overall) comparisons between NoRTune and baselines on four workloads. The values represent the relative speedups measured against NoRTune (higher values indicate greater baseline tuning cost).

Models	PageRank		Aggregation		KMeans		Bayes	
	Algorithm	Overall	Algorithm	Overall	Algorithm	Overall	Algorithm	Overall
LlamaTune (SMAC)	1.03x	1.01x	0.90x	1.03x	1.26x	1.21x	1.20x	0.92x
LlamaTune (BO)	1.02x	0.93x	0.90x	0.98x	1.13x	1.38x	1.26x	1.02x
CSAT	1.23x	1.44x	1.10x	1.05x	1.39x	2.01x	1.15x	1.07x

and sets the foundation for highly practical and robust tuning frameworks applicable to real-world challenges that commonly face noise variability and scalability constraints.

Notably, we faced some limitations with this research. For example, we found that NoRTune's efficacy may be limited when handling workloads that are inherently insensitive to configuration changes (e.g., certain SQL-based queries). This remains a fundamental challenge in system tuning, where not all workloads benefit equally from configuration optimization. Future research should pursue two main directions. First, NoRTune's utility should be extended to online continuous tuning environments, where its rapid adaptability (e.g., production systems) will provide immediate benefits. Second, workload-aware optimization strategies should be integrated to dynamically evaluate the potential benefits of tuning based on early workload characteristics, which may mitigate issues related to the first limitation discussed. By identifying workloads where tuning is unlikely to produce significant gains, NoRTune could avoid unnecessary resource consumption and instead focus on the most impactful optimizations. This would further improve the model's overall tuning efficiency and resource utilization across a multitude of domains and platforms.

CRediT authorship contribution statement

Jieun Lee: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Sangmin Seo:** Writing – review & editing, Visualization, Formal analysis, Conceptualization. **Chanho Yeom:** Writing – review & editing, Methodology, Conceptualization. **Huijun Jin:** Writing – review & editing, Conceptualization. **Sein Kwon:** Writing – review & editing, Conceptualization. **Sanghyun Park:** Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was supported by the National Research Foundation (NRF) funded by the Korean government (MSIT) (No. RS-2023-00229822).

Data availability

Data will be made available on request.

References

Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al., 2015. Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1383–1394.

Dou, H., Zhang, L., Zhang, Y., Chen, P., Zheng, Z., 2023. TurBO: A cost-efficient configuration-based auto-tuning approach for cluster-based big data frameworks. J. Parallel Distrib. Comput. 177, 89–105.

Eriksson, D., Pearce, M., Gardner, J., Turner, R.D., Poloczek, M., 2019. Scalable global optimization via local Bayesian optimization. Adv. Neural Inf. Process. Syst. 32.

Fekry, A., Carata, L., Pasquier, T., Rice, A., Hopper, A., 2020. To tune or not to tune? in search of optimal configurations for data analytics. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 2494–2504.

Frazier, P.I., 2018. A tutorial on Bayesian optimization. arXiv preprint arXiv:1807.02811.

Hernández-Lobato, J.M., Hoffman, M.W., Ghahramani, Z., 2014. Predictive entropy search for efficient global optimization of black-box functions. Adv. Neural Inf. Process. Syst. 27.

Herodotou, H., Chen, Y., Lu, J., 2020. A survey on automatic parameter tuning for big data processing systems. ACM Comput. Surv. 53 (2), 1–37.

Huang, D., Allen, T.T., Notz, W.I., Zeng, N., 2006. Global optimization of stochastic black-box systems via sequential kriging meta-models. J. Global Optim. 34, 441–466.

Huang, S., Huang, J., Dai, J., Xie, T., Huang, B., 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, pp. 41–51.

Hutter, F., Hoos, H.H., Leyton-Brown, K., 2011. Sequential model-based optimization for general algorithm configuration. In: Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5. Springer, pp. 507–523.

Islam, M.T., Karunasekera, S., Buyya, R., 2021. Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. IEEE Trans. Parallel Distrib. Syst. 33 (7), 1695–1710.

Jones, D.R., Schonlau, M., Welch, W.J., 1998. Efficient global optimization of expensive black-box functions. J. Global Optim. 13, 455–492.

Kanellis, K., Ding, C., Kroth, B., Müller, A., Curino, C., Venkataraman, S., 2022. LlamaTune: sample-efficient DBMS configuration tuning. Proc. VLDB Endow. 15 (11), 2953–2965.

Kang, M., Lee, J.-G., 2020. Effect of garbage collection in iterative algorithms on spark: an experimental analysis. J. Supercomput. 76, 7204–7218.

Khan, M.M., Yu, W., 2021. Robotune: high-dimensional configuration tuning for cluster-based data analytics. In: Proceedings of the 50th International Conference on Parallel Processing. pp. 1–10.

Kushner, H.J., 1964. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise.

Letham, B., Calandra, R., Rai, A., Bakshy, E., 2020. Re-examining linear embeddings for high-dimensional Bayesian optimization. Adv. Neural Inf. Process. Syst. 33, 1546–1558.

Letham, B., Karrer, B., Ottoni, G., Bakshy, E., 2019. Constrained Bayesian optimization with noisy experiments. Bayesian Anal. 14 (2), 495–519.

Li, Y., Bao, L., Huang, K., Wu, C., 2025b. CSAT: Configuration structure-aware tuning for highly configurable software systems. J. Syst. Softw. 222, 112316.

Li, Y., Jiang, H., Shen, Y., Fang, Y., Yang, X., Huang, D., Zhang, X., Zhang, W., Zhang, C., Chen, P., et al., 2023. Towards general and efficient online tuning for spark. Proc. VLDB Endow. 16 (12), 3570–3583.

Li, J., Ye, J., Mao, Y., Gao, Y., Chen, L., 2025a. LOFTune: A low-overhead and flexible approach for spark sql configuration tuning. IEEE Trans. Knowl. Data Eng..

Lu, J., Chen, Y., Herodotou, H., Babu, S., 2019. Speedup your analytics: Automatic parameter tuning for databases and big data systems. Proc. VLDB Endow. 12 (12), 1970–1973.

Lundberg, S.M., Lee, S.-I., 2017. A unified approach to interpreting model predictions. Adv. Neural Inf. Process. Syst. 30.

Lyu, C., Fan, Q., Guyard, P., Diao, Y., 2024. A spark optimizer for adaptive, fine-grained parameter tuning. Proc. VLDB Endow. 17 (11), 3565–3579.

Masouros, D., Retsinas, G., Xydis, S., Soudris, D., 2024. Sparkle: Deep learning driven autotuning for taming high-dimensionality of spark deployments. IEEE Trans. Cloud Comput..

Moriconi, R., Deisenroth, M.P., Sesh Kumar, K., 2020. High-dimensional Bayesian optimization using low-dimensional feature spaces. Mach. Learn. 109, 1925–1943.

Nayebi, A., Munteanu, A., Poloczek, M., 2019. A framework for Bayesian optimization in embedded subspaces. In: International Conference on Machine Learning. PMLR, pp. 4752–4761.

Papenmeier, L., Nardi, L., Poloczek, M., 2022. Increasing the scope as you learn: Adaptive Bayesian optimization in nested subspaces. Adv. Neural Inf. Process. Syst. 35, 11586–11601.

Papenmeier, L., Nardi, L., Poloczek, M., 2023. Bounce: a reliable Bayesian optimization algorithm for combinatorial and mixed spaces. In: 37th Conference on Neural Information Processing Systems. NeurIPS 2023.

Picheny, V., Wagner, T., Ginsbourger, D., 2013. A benchmark of kriging-based infill criteria for noisy optimization. Struct. Multidiscip. Optim. 48, 607–626.

Shen, Y., Ren, X., Lu, Y., Jiang, H., Xu, H., Peng, D., Li, Y., Zhang, W., Cui, B., 2023. Rover: An online spark SQL tuning service via generalized transfer learning. In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. pp. 4800–4812.

Spark, 2024. Configuration - spark 3.1.3 documentation. https://archive.apache.org/dist/spark/docs/3.1.3/configuration.html.

Srinivas, N., Krause, A., Kakade, S., Seeger, M., 2010. Gaussian process optimization in the bandit setting: No regret and experimental design. In: Proceedings of the 27th International Conference on Machine Learning. Omni Press, pp. 1015–1022.

Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B., 2017. Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 1009–1024.

Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., Pavlo, A., 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. Proc. VLDB Endow. 14 (7), 1241–1253.

Vazquez, E., Villemonteix, J., Sidorkiewicz, M., Walter, E., 2008. Global optimization based on noisy evaluations: an empirical study of two statistical approaches. In: Journal of Physics: Conference Series. Vol. 135, IOP Publishing, 012100.

Wang, Z., Hutter, F., Zoghi, M., Matheson, D., De Feitas, N., 2016. Bayesian optimization in a billion dimensions via random embeddings. J. Artificial Intelligence Res. 55, 361–387.

Williams, C.K., Rasmussen, C.E., 2006. Gaussian Processes for Machine Learning, vol. 2, MIT press Cambridge, MA.

Wilson, J., Hutter, F., Deisenroth, M., 2018. Maximizing acquisition functions for Bayesian optimization. Adv. Neural Inf. Process. Syst. 31.

Wu, Y., Huang, X., Wei, Z., Cheng, H., Xin, C., Chen, Z., Chen, B., Wu, Y., Wang, H., Zhang, T., et al., 2024. Towards resource efficiency: Practical insights into large-scale spark workloads at ByteDance. Proc. VLDB Endow. 17 (12), 3759–3771.

Xin, J., Hwang, K., Yu, Z., 2022. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In: Proceedings of the 2022 International Conference on Management of Data. pp. 674–684.

Xin, R., Rosen, J., 2015. Project tungsten: Bringing spark closer to bare metal. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. Databricks Blog.

Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I., 2010. Spark: Cluster computing with working sets. In: 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10).

Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al., 2016. Apache spark: a unified engine for big data processing. Commun. ACM 59 (11), 56–65.

Zhang, X., Chang, Z., Li, Y., Wu, H., Tan, J., Li, F., Cui, B., 2022a. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. Proc. VLDB Endow. 15 (9), 1808–1821.

Zhang, X., Wu, H., Chang, Z., Jin, S., Tan, J., Li, F., Zhang, T., Cui, B., 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In: Proceedings of the 2021 International Conference on Management of Data. pp. 2102–2114.

Zhang, X., Wu, H., Li, Y., Tan, J., Li, F., Cui, B., 2022b. Towards dynamic and safe configuration tuning for cloud databases. In: Proceedings of the 2022 International Conference on Management of Data. pp. 631–645.

Zhang, X., Wu, H., Li, Y., Tang, Z., Tan, J., Li, F., Cui, B., 2023. An efficient transfer learning based configuration adviser for database tuning. Proc. VLDB Endow. 17 (3), 539–552.