Towards General and Efficient Online Tuning for Spark

연세대학교 컴퓨터과학과 이지은

2024년 3월

과제명: IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발

과제번호: 2017-0-00477

과학기술정보통신부
Ministry of Science and ICT

연세대학교
YONSEI UNIVERSITY

IITP 정보통신기술진흥센터
Institute for Information & communications Technology Promotion

ABSTRACT

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

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

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

1. INTRODUCTION

- The rapid growth of the World Wide Web, E-commerce, Social media and other applications is producing massive mounts of ever-increasing raw data every day.

- Companies often utilize this **"big data"** to innovate pioneering products and solutions, e.g., improving customer service, marketing, sales, team management, and many other routine operations. To this end, many distributed analytics platforms (e.g., **Hadoop Mapreduce, Spark, Storm, Flink, Heron, Samza**) have emerged to deal with this big data trend.

- **Spark** is one of the representative systems that enable the manipulation and analysis of large datasets with in-memory cluster computing.

- As of today, thousands of organizations, such as Google, Microsoft, Amazon, Meta, Oracle, Snowflake, Databricks, Tencent, and Alibaba are using Spark in production across **a vast range of fields** including data processing, machine learning, graph computing, stream computing and database management.

1. INTRODUCTION

- The performance of Spark jobs highly depends on the choice of Spark configuration parameters. Misconfiguration can lead to unsatisfying performance such as **long runtime, resource contention, and resource under-utilization.** For instance, an improper configuration may lead to over 100 times the execution time compared with an elaborately designed one.

- Therefore, it is crucial to choose proper configurations to achieve high performance (e.g., in terms of execution time or cost). The benefit of tuning is even more significant for **periodic (hourly, daily, weekly, etc) jobs**, where they occupy a large proportion of Spark jobs in production. In this paper, **we focus on tuning periodic Spark jobs.**

- To achieve a near-optimal performance of periodic tasks, users are required to determine a large number of performance-critical configuration parameters. However, it is very difficult to manually tune the configuration parameters of a Spark task due to **(1) the high-dimensionality of the parameter space** and **(2) complex and non-linear interactions among parameters.**

- In addition, manual tuning is usually time-consuming and labor-intensive, and thus it fails to scale to a huge number of tasks in a data platform. Recent studies propose to utilize auto-tuning techniques to optimize the Spark parameters. During our attempts to apply these approaches in real tuning tasks, **we realize three aspects of limitations:**

1. INTRODUCTION

C.1 Limited Functionality

- Lots of methods are designed to minimize execution time, i.e., finding the fastest configuration.
- However, the goal in many scenarios involves the execution cost, i.e., **the cheapest configuration** ,or more generalized objectives such as **a weighted combination between runtime and cost.**
- In addition, the tuning process **should satisfy some application requirements** ➔ the execution time < threshold (constraint)

C.2 High Overhead

- Many tuning frameworks belong to the offline tuning paradigm.
- Concretely, they collect training samples by running jobs with different configurations on a non-production cluster, and then train a performance model to suggest new configurations for Spark job running on the production cluster. Training such an accurate performance model involves lots of offline job executions (e.g., 1000-10000). ➔ **very time-consuming and expensive**
- In addition, since workloads may change as time proceeds, offline tuning methods cannot capture and adapt to these dynamics easily.

C.3 Inefficient Search

- The configuration search in many methods suffers from the low-efficiency issue that arises from two aspects:
- **(1) Huge search space** → the curse of dimensionality
- **(2) Intrinsic dilemma** of black-box optimization → facing the cold-start issue and challenging when dealing with the exploration and exploitation trade-off
- The meta knowledge across tasks could help to deal with cold-start and slow convergence issues during search.

1. INTRODUCTION

C.1 Limited Functionality

- A generalized formulation about the tuning problem, which supports various goals and multiple performance constraints conveniently.
- We develop a noise-robust Bayesian optimization-based solution.

C.2 High Overhead

- Different from offline approaches that require a large number of job executions in advance, we propose an **online tuning paradigm**, where we tune parameters along with the in-production periodic executions of each Spark job, thus incurring no additional execution cost as in offline methods.
- This online setting requires that the BO-based framework should **converge to good configurations quickly (efficiency)** and **explore as few bad configurations as possible to get there (safety)**. To this end, we design a configuration acquisition method, which models and utilizes the **safe region** from the Gaussian process **to achieve safe exploration**, and then leverages **the expected constrained improvement** to trade-off exploration and exploitation when suggesting new configurations.
- To accommodate the dynamic workload in the online setting, we **encode the workload characteristic** into the Gaussian process using mixed kernels in BO.

C.3 Inefficient Search

- We develop three innovative techniques within BO to accelerate tuning.
- **(1) Adaptive sub-space generation**
- **(2) Approximate gradient descent**
- **(3) Meta-learning module**

1. INTRODUCTION

C.3 Inefficient Search

- We develop three innovative techniques within BO to accelerate tuning.
- **(1) Adaptive sub-space generation**
  - BO is known to be difficult to scale to high dimensions.
  - The size of the sub-space will be automatically adjusted based on the intermediate tuning results to balance convergence speed and quality.

- **(2) Approximate gradient descent**
  - Since gradient methods have shown superior convergence speed compared with black-box methods in numerical optimization, we develop a novel **approximate gradient descent** method within BO to select the next configuration, which can estimate and leverage the derivative information of the objective function to speed up the search process.

- **(3) Meta-learning module**
  - Motivated by the observation that the regions of optimal/near-optimal configurations are similar between two similar tuning tasks, we design a **meta-learning module** that learns the similarity between tasks based on tuning history from previous tasks.
  - Further, this module can transfer useful knowledge, e.g., the optimal configuration or the distribution of good configurations, to the current tuning task.

2. BACKGROUD

2.2 Terminology

Our goal is to find the optimal or near-optimal configuration from the configuration space.

Configuration Space.

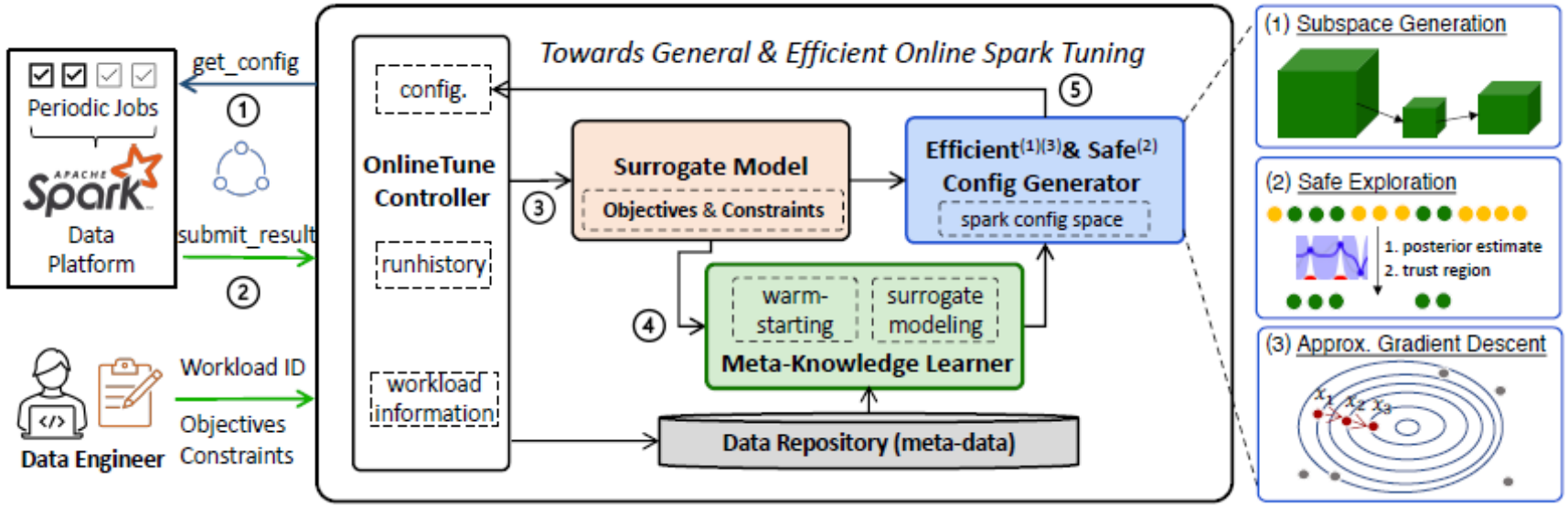
- N spark parameters : $x_{p1}, x_{p2}, \dots, x_{pN}$
  - Each p_i corresponds to a specific parameter in Spark, e.g., `spark.executor.instances`,`spark.executor.memory`.
- The range of the i^{th} parameter p_i is denoted by Λ^i.
  - $\Lambda_{cs} = \Lambda^1 \times \Lambda^2 \times \cdots \Lambda^N$
- x : a configuration instance (a vector) in Λ_{cs}.
  - The i^{th} element x^i corresponds to the value of i^{th} parameter p_i and $x^i \in \Lambda^i$.

Configuration Subspace.

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

3. SYSTEM DESIGN

3.1 Overview



1) The **OnlineTune controller** is responsible for orchestrating the entire configuration tuning process, along with interactions the data platform and end users.

2) The multi-purpose **surrogate models** are to learn complex relationships between configurations and objective metrics or performance constraints

3) The **efficient and safe configuration generator** is to suggest a promising configuration to evaluate for a tuning task.

4) The **meta-knowledge learner** can leverage tuning history form previous tasks to further accelerate the search for configurations.

5) The **data repository** is to store tuning-related data, including history, workload metrics, etc.

3.2 Generalized Problem Formulation

- Given a tuning task, our goal is to find the optimal or near-optimal Spark configuration that minimizes the objective and **satisfies performance/safety requirements.** To support general Spark tuning cases, we provide a **generalized tuning formulation** that supports **both different objectives and inequality constraints from real applications.**

- $T(x)$: the **runtime** function for tuning task

- $P(x)$: the **price per unit of time** for all computing resources
  - **not easy** to obtain and positively correlated with the amount of resource used in x given a fixed computing environment

- $R(x)$: the **amount of resource** used by the job execution. Using this to replace $P(x)$
  - in terms of **CPU cores and memory**
  - e.g., `spark.executor.instances, spark.executor.cores, spark.executor.memory`

- Further, we formulate this problem as follows:

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

- **Generalized Objective with Different $\boldsymbol{\beta}$**
  - $\beta \in [0,1]$
  - $\beta = 1$ or 0 or 0.5

3.3 Bayesian Optimization-based Framework

- Bayesian optimization (BO) is a framework to solve black-box problems **where the objective can only be observed via evaluation.**
- The main advantages of BO are that it is **robust to noise** and **can estimate uncertainty** to balance exploration and exploitation.

A typical loop in vanilla BO contains the following four steps:

1) Fitting a *surrogate model M* based on observed configurations $D = \{(x_1, y_1), \dots (x_{n-1}, y_{n-1})\}$
2) Choosing the next promising configuration that maximizes the *acquisition function* $x_n = argmax_{x \in \chi} \alpha(x; M)$
3) Evaluating the chosen configuration x_n to obtain its performance $y_n = f(x_n) + \epsilon$ with noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$
4) Adding the pair (x_n, y_n) to observations so that $D = \{(x_1, y_1), \dots (x_n, y_n)\}$

Algorithm 1: Pseudo code for Bayesian optimization.

Input: the search budget $\mathcal{B}$, the configuration search space $\mathbf{\Lambda_{cs}}$.
Output: the best configuration found.

```
initialize observations D with several configuration evaluations.
while budget B does not exhaust do
    train a surrogate M on the current observations D.
    choose the next configuration by x_i = arg max_{x ∈ Λ_cs} α(x; M).
    evaluate the selected x_i and obtain its performance y_i.
    augment D = D ∪ (x_i, y_i).
end while
return the best configuration in D.
```

3.3 Bayesian Optimization-based Framework

Surrogate Model – Gaussian Process

- hyperparameter-free and provides closed-form inference
- Given an unseen configuration x, the *predictive mean* and *covariance* are expressed as follows:

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

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

  - K : covariance matrix
  - X : the observed configuration vectors
  - τ^2 : the level of white noise

Acquisition Function

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

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

- where $\gamma(x) = \frac{y^* - \mu(x)}{\sigma(x)}$, y^* is the best performance observed.
- $\Phi(\cdot)$ and $\phi(\cdot)$ are *standard normal cumulative distribution function* and *probability density function.*

3.3 Bayesian Optimization-based Framework

Dynamic Workload Support

- The workload may change during the online tuning process, particularly the **data size**. Since the same configuration for a workload may achieve different results with different input data sizes, the change in data size could affect the tuning result.
- To accommodate this, we take the data size along with configuration into consideration and model the objective value with the Gaussian Process.
  - $\bar{x}_i = \{x_i^1, \ldots, x_i^N, ds_i\}$
  - ds_i is the data size of that run

- **Using mixed kernels**: Matern kernel (for numerical), Hamming kernel (for categorical), and SE kernel (for data size)

Initial configurations

- Sampling several configurations using low-discrepancy sequences to initialize the observation *D*.
- To accelerate the convergence in the beginning, the meta-learning module can suggest the initial configurations based on task similarity, instead using the low-discrepancy sequences.

Stopping & Restarting Criterion

- When the expected improvement is less than threshold, the stopping criterion is activated. If a continuous degradation is detected, re-tuning becomes necessary, and our framework will restart the tuning process, where meta-learning is utilized to extract knowledge from the previous runhistory to speed up the optimization.

4. EFFICIENT & SAFE CONFIG ACQUISITION

4.1 Sub-space Generation

An intuitive idea is to use a smaller configuration sub-space that includes the most influential parameters, instead of the original huge space. Therefore, when dealing with the high-dimensional space, we need to answer **two questions**:
(1) how to **measure the importance** of Spark parameters
(2) how to adjust the size of the sub-space adaptively to pursue efficiency and effectiveness simultaneously.

Parameter Importance & Sub-space

- While existing methods only consider the influence between each parameter and performance via Spearman Correlation Coefficient or weights of a learning model, we consider the importance of **both single parameters** and of **interactions between parameters.**

- We adopt **functional ANOVA** to assess the importance of parameters. Given the tuning history from a task, our parameter importance analysis module could rank the parameters according to their importance. **The final importance scores are obtained by averaging the scores from those tasks.**
  - $\Lambda_{sub} = \Lambda^1 \times \Lambda^2 \times \cdots \Lambda^K,$ (K is the size of the sub-space)

- **Note that, when starting tuning from zero**, there is **no tuning history available** to obtain the parameter ranking based on the importance scores. We start with an initial parameter ranking suggested by experts. Once new tuning history arrives, we will continuously update the importance score for each parameter based on FANOVA.

4.1 Sub-space Generation

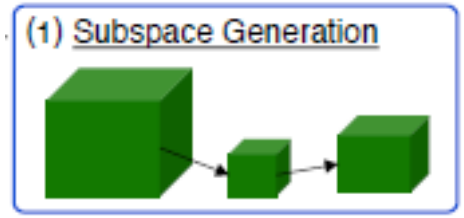
An intuitive idea is to use a smaller configuration sub-space that includes the most influential parameters, instead of the original huge space. Therefore, when dealing with the high-dimensional space, we need to answer **two questions**:
(1) how to measure the importance of Spark parameters
(2) how to **adjust the size of the sub-space adaptively** to pursue efficiency and effectiveness simultaneously.

Evolution Strategy of Sub-space

- Different from existing methods that use a fixed sub-space, we propose to **automatically adjust the size of the sub-space**, i.e., K.
- On one hand, a sub-space **should be sufficiently large to contain good configurations.** On the other hand, it should be **small enough** to ensure that BO is accurate and efficient within this sub-space. Therefore, the evolution of K is crucial.

- Similar to TuRBO, we adopt an intuitive design that we expand the sub-space when the BO optimizer **"makes progress"**, i.e., it finds better configurations in the current sub-space, and **"shrinks"** it when the optimizer appears stuck.
  - **"success"** : a configuration that improves over the best configuration found
  - **"failure"** : a configuration that fails.

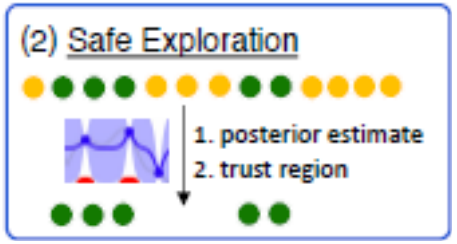
- After τ_{sucs} **consecutive successes** : $K \leftarrow \min(K_{max}, K+2)$
- After τ_{fail} **consecutive failures** : $K \leftarrow max(K_{min}, K-2)$



- Once the size of sub-space is changed, the counters for the success and failure events are set to zero.
- $\tau_{sucs} = 3,\ \tau_{fail} = 5,\ K_{min} = 4,\ K_{init} = 10,\ K_{max} = 30$

4.2 Safe Exploration and Exploitation

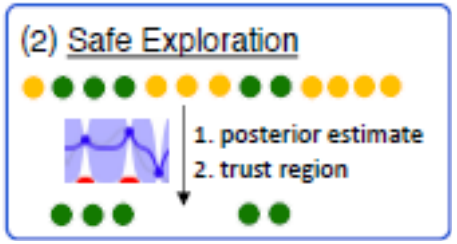
The main process for the configuration generator to produce a new configuration is:
(1) build the final **safe region** S
(2) choose the next configuration by solving the EIC over S.



- To explicitly ensure the safety of job execution, we further propose **safe exploration and exploitation** to select the next candidate. Given the predictive mean $\mu_t^T(x)$ and variance $\sigma_t^{T^2}(x)$ from the runtime surrogate built at the t^{th} iteration, the upper bound is defined as:
  - $u_t^T(x) = \mu_t^T(x) + \gamma\sigma_t^{T^2}(x)$
  - where γ is a constant that controls the bound and $\gamma \in (0,1]$
- The safe region for runtime at the t^{th} iteration is defined as $S_t^T = \{x \in \Lambda_{sub} | u_t^T(x) \leq T_{max}\}$.
- For multiple constraints, the final safe region is the intersection of single safe regions.

4.2 Safe Exploration and Exploitation

The main process for the configuration generator to produce a new configuration is:
(1) build the final safe region S
(2) choose the **next configuration** by solving the EIC over S.



- In online tuning scenarios, evaluating a configuration that violates the constraints will **inevitably downgrade the job performance.**
- To deal with constraints, we apply the **EI with constraints (EIC)** as the acquisition function.
- EIC takes the probability of satisfying the constraints into consideration.
  - $EIC(x) = Pr[T(x) \leq T_{max}] \cdot EI(x)$
  - where $Pr[T(x) \leq T_{max}]$ refers to the probability that satisfies the runtime constraints.
  - $Pr[T(x) \leq T_{max}] = \int_{-\infty}^{T_{max}} p(T(x)|x)dT(x)$

4.3 Approximate Gradient Descent

In Spark tuning, the objective in Eq. 1 is treated as a black-box function of input parameters with no available derivative information. However, some parts of the objective functions are not always black-box. Though the relationship between runtime and Spark parameters is complicated, the resource function has an analytic form:

$$R(x) = \#cpu\ vcores(x) + c \cdot \#mem(x)$$

- which is directly determined by the values of parameters
  - `spark.executor.instances, spark.executor.cores, spark.executor.memory`

- **Gradient-based techniques** perform well in optimizing differentiable functions. Compared with BO, the **derivative information** of the objective function provides more precise guidance on how to choose the next input parameters based on the current configurations.

- When observations D are sufficient to approximate the objective function $f(x)$, we can estimate the derivative information.

- However, **since AGD emphasizes the exploitation signal only during the search process**, it could be stuck in local configurations easily. To address this and integrate the benefits of both methods, we **alternately apply AGD and BO** to select the next configuration so that the combined framework **balances exploration and exploitation** well with a convergence guarantee provided in previous work [SMAC].

- Therefore, we develop the approximate gradient descent (AGD) technique within BO to utilize the derivative information of the objective function and further accelerate the convergence over the search space.

4.3 Approximate Gradient Descent

- To apply gradient descent, the partial derivative of *f(x)* over each numerical parameter x^i is :

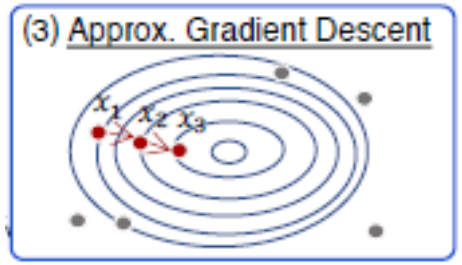
$$\frac{\partial f}{\partial x^i}(\boldsymbol{x}) = \beta\left(\frac{T(\boldsymbol{x})}{R(\boldsymbol{x})}\right)^{\beta-1}\frac{\partial T}{\partial x^i}(\boldsymbol{x}) + (1-\beta)\left(\frac{T(\boldsymbol{x})}{R(\boldsymbol{x})}\right)^{\beta}\frac{\partial R}{\partial x^i}(\boldsymbol{x}),$$

  - where x^i is the i^{th} Spark parameter. Since the **resource function *R(x)* is a white-box function**, the partial derivative of *R(x)* has an analytical form.

- However, the partial derivative of the runtime ***T(x)* cannot be calculated directly**. We **approximate the partial derivative** by:

$$\frac{\partial T}{\partial x^i}(\boldsymbol{x}) \approx \frac{T(\boldsymbol{x}+\boldsymbol{\epsilon_i}) - T(\boldsymbol{x}-\boldsymbol{\epsilon_i})}{2|\boldsymbol{\epsilon_i}|}$$

  - where all dimensions except the i^{th} element in ϵ_i are zero. ($\epsilon_2 = [0, e, 0, \dots, 0]$), maybe.. $\epsilon \sim \mathcal{N}(0, \sigma^2)$
  - The values of $T(x+\epsilon_i)$ and $T(x-\epsilon_i)$ are predicted by the surrogate for runtime.



- Then, the value of each parameter x^i is updated by:

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

  - where η is the learning rate with 0.001
  - During the tuning procedure, AGD is integrated into BO.

- Every $N_{AGD} = 5$ BO iterations, instead of BO, the next configuration *x* is generated by conducting AGD based on the best configuration found

4.3 Approximate Gradient Descent

Algorithm 2: Pseudo code for configuration generation.

Input: Evaluation observations D, constants, e.g., N_{AGD}, η, etc.
Output: the suggested configuration $\boldsymbol{x_i}$ for the i^{th} iteration.

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

5. META-LEARNING BASED ACCELERATION

5.1 Task Characterization & Similarity Learning

- We extract the feature vector of a tuning task (i.e., meta-features) from `SparkEventLog`, in which Spark event information is logged during execution. The meta-features summarize information in **two levels: stage and task**.
  - **Stage** information contains the **Spark actions and transformations** used in the Stage, which shows core Spark function calls made during the job execution.
  - **Task** information describes whether, overall, the task was **read or write-intensive, CPU intensive**, etc.
  - A total of 75 features: 11 Stage information and 64 Task information

- A straightforward method that measures the similarity is to compute the **Euclidean distance** between the two meta-features of corresponding tasks. However, the type and scale of each meta-feature are heterogeneous, which greatly decreases the effectiveness of Euclidean distance. Besides, each meta-feature poses a diverse impact on similarity learning.

- To address these issues, we propose to use **a supervised learning method** (i.e., regression model) to learn the similarity given two tasks.

- Given the meta-features of two input tasks: v_1 and v_2, the regression model $M_{reg}: (v_1, v_2) \rightarrow d$ predicts their distances $d \in [0,1]$
  - A smaller distance d indicates that the two tasks are more similar to each other.

- The training data : collected a wide range of Spark jobs $T_{1,\dots,K}$ and the corresponding tuning history $H_{1,\dots,K}$
  - The tuning history H_i of the i^{th} task consists of the pairs of configuration and its performance (maybe meta-features..?).

5.1 Task Characterization & Similarity Learning

- **To define the distance metrics..**
  - by calculating the number of discordant pairs of predictions (i.e., pair-wise ranking) using **the negative Kendall-tau coefficient.**
  - **What is the Kendall-tau coefficient?**
    - A kind of rank correlation coefficients (순위 상관 계수)
    - 두 변수들 간의 순위를 비교하여 연관성을 계산
    - [-1, 1]

  - That is, given two configurations x_k and x_l, and surrogates model M^i and M^j on the i^{th} and j^{th} task,
  - the pair (x_k, x_l) is discordant if the sort order of $(M^i(x_k), M^i(x_l))$ and $(M^j(x_k), M^j(x_l))$ disagrees.

  - The distance is defined as **the ratio of discordant pairs**:

$$Dist\left(M^i, M^j\right) = \frac{1 - \tau^{D_{rand}}\left(M^i, M^j\right)}{2}$$

    - where $\tau^{D_{rand}}\left(M^i, M^j\right)$ is the Kendall-tau coefficient of two vectors by applying surrogates on **randomly sampled configurations $\boldsymbol{D_{rand}}$.**
    - The range of distance is scaled to [0,1]

  - The regression model is LightGBM regressor.

5.2 Warm-starting & Surrogate Improvement

Initial design with warm-starting

- In vanilla BO, the search process starts from scratch.
- We rank the previous tasks using the predictions of meta-learner and **choose the top-3 most similar tasks.** Then, we **select the best Spark configuration found in these top-3 tasks** and **set them as the initial configurations** before starting the main BO loop.
- In this way, our framework could achieve a better performance in the beginning. In addition, we also can suggest the sub-space for a new tuning task using task similarity.

Surrogate modeling with meta-learning

- We propose to build a meta-learning surrogate ensemble M_{meta}.
- The prediction of M_{meta} at configuration x is given by $y \sim \mathcal{N}(\mu_{meta}(x), \sigma^2_{meta}(x))$:

$$\mu_{meta}(\boldsymbol{x}) = \sum_i w_i \mu_i(\boldsymbol{x}), \quad \sigma^2_{meta}(\boldsymbol{x}) = \sum_i w_i^2 \sigma_i^2(\boldsymbol{x})$$

  - where μ_i and σ_i^2 are the predictive mean and variance form base surrogate M^i.
  - The weight of base surrogate w_i reflects the similarity between the previous and current task.
    - $w_i = 1 - Dist(M^i, M^t)$ where $\Sigma_i w_i = 1$

6. EXPERIMENTS AND RESULTS

6. Experiments and results

To evaluate our framework, in this section, we list three insights we want to investigate.

1) **Practicality**
   - Achieving significant cost reductions compared with human experts when tuning 25K real-world Spark tasks in Tencent

2) **Generality**
   - Supporting different tuning objectives (runtime and cost) and Spark task types (MR and SQL)
   - Consistently outperforming state-of-the-art Spark tuning approaches on various standard benchmarks

3) **Efficiency**
   - Greatly accelerating the tuning process and requires only a few trials to find near-optimal configuration.

6.1 Experimental Setup

Benchmark Programs

- HiBench (Bayes, Kmeans, NWeight, WordCound, PageRank, TeraSort)
- 16 tasks is used in the meta-learning experiment

Environment

- Tencent data platform
  - x86 clusters with four nodes
  - 2 AMD EPY7K62 2.80GHz 48-core processors and 512 GB PC4 memory
  - Spark 3.0

Spark Paramaeters

- 30 parameters used in Tuneful
- the value ranges of the parameters are set differently depending on the cluster size

Compared Methods

1) *Random Search*
2) *RFHOC* random forests for each task + Genetic algorithm
3) *DAC* a datasize-aware auto-tuning approach; hierarchical regression tree models + Genetic algorithm
4) *CherryPick* BO-based approach; aims at minimizing the user cost and subjects to a runtime threshold
5) *Tuneful* BO-based approach in an online manner
6) *LOCAT* BO-based online approach for Spark SQL

6.1 Experimental Setup

Objectives

- *Runtime* ($\beta = 1$) , *Cost* ($\beta = 0.5$)

Metrics & Settings

- *Speedup*
  - the **execution time** of the best-found configuration relative to the one from random search
- *Cost*
  - The *Cost* at i^{th} iteration is the **execution cost** of the i^{th} tried configuration.
  - "Min Cost" refers to the execution cost of the best configuration found.
- *Cost Reduction*
  - $= \frac{Cost_{ref} - Cost}{Cost_{ref}}$
  - reference methods: random search or default configuration, etc.
- All experiments running on HiBench are repeated 10 times with different random seeds and the average metrics are reported.

6.3 Results on Public Benchmarks

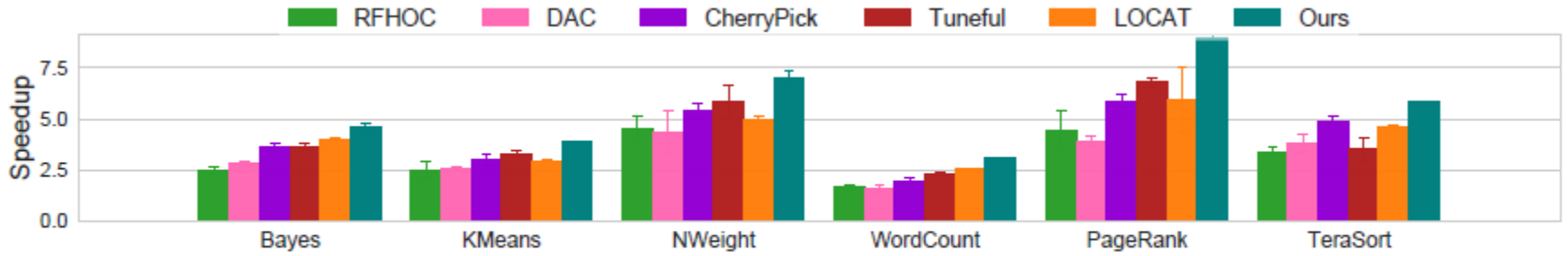


Figure 4: Speedup of compared methods relative to random search on 6 HiBench tasks.

- The overall budget : 30 iterations
- runtime constraint : $2 \times Runtime_{def}$

Observations

a. ML-based approaches (RFHOC and DAC) achieve a relatively lower speedup compared with BO-based approaches.
   - ➔ ML-based models need a large number of training samples
   - ➔ 30 iterations are insufficient in the search space

b. BO-based approaches (CherryPick, Tuneful, LOCAT, and ours) get a better result under the limited budget.
   - ➔ CherryPick does not reduce the search space, tuning large space
   - ➔ Tuneful and LOCAT use different techniques to select important parameters and their rankings are unstable

c. Our framework achieves the best and consistent speedup among all compared methods

6.3 Results on Public Benchmarks

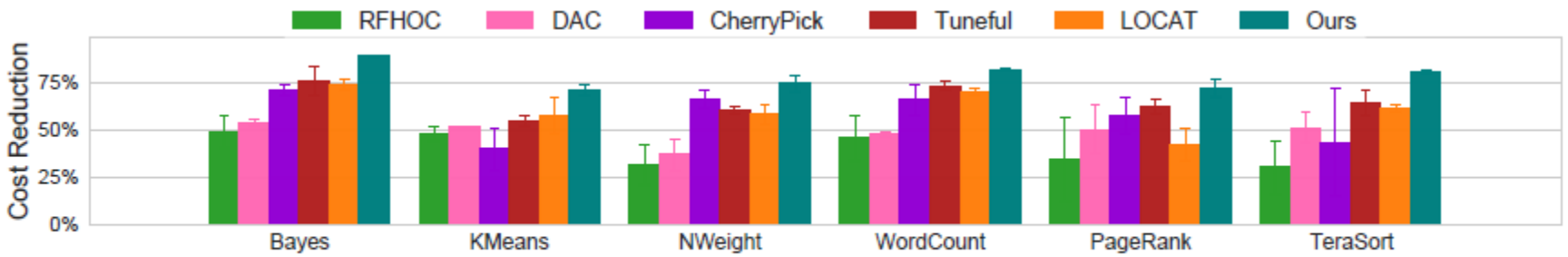


Figure 5: Cost reduction of compared methods relative to random search on 6 HiBench tasks.

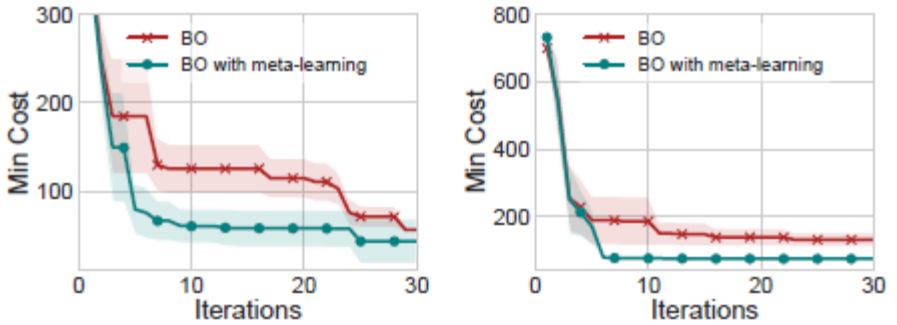
- we also present the relative execution cost reduction of all methods compared with random search.
- Compared with runtime, the execution cost is a more difficult objective to optimize.

- Concretely, our framework achieves a cost reduction of 71.22-88.97% relative to random search and obtains a cost reduction of 38.43% and 45.20% on average compared with competitive baselines Tuneful and LOCAT, respectively.

6.4 Meta-learning Experiments

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

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



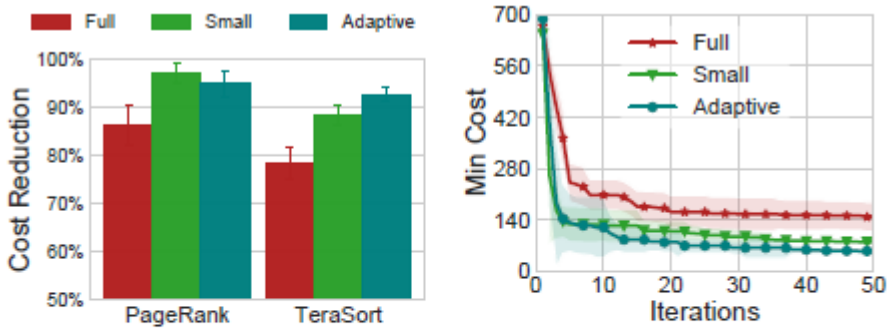
(a) WordCount (b) TeraSort

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

Warm-starting

Acceleration with ensemble surrogate

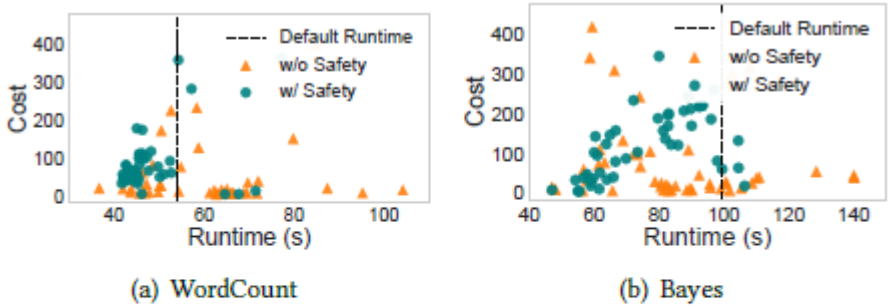
6.5 Ablation Studies



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

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

Sub-space Generation



(a) WordCount (b) Bayes

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

Safe Exploration and Exploitation

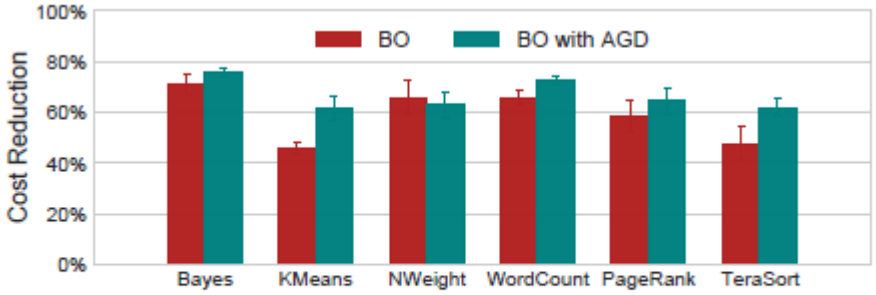


Figure 9: Tuning results when using approximate gradient descent relative to random search.

Approximate Gradient Descent