The Journal of Systems & Software 180 (2021) 111028

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model ☆

Guoli Cheng, Shi Ying *, Bingming Wang

School of Computer Science, Wuhan University, Bayi Road 299, Wuhan, China

ARTICLE INFO

Article history:
Received 18 March 2021
Received in revised form 7 May 2021
Accepted 4 June 2021
Available online 16 June 2021

Keywords:
Spark
Configuration tuning
Multi-objective optimization

ABSTRACT

Choosing the right configuration for Spark deployed in the public cloud to ensure the efficient running of periodic jobs is hard, because there can be a huge configuration space to explore which is composed of numerous performance-related parameters in different dimensions (e.g., application-level and cloud-level). Choosing poorly will not only significantly degrade performance but may also lead to greater overhead. However, automatically searching for the optimal configuration of various applications to trade-off performance and cost is challenging. To address this issue, we propose a new optimal configuration search algorithm named AB-MOEA/D by combining multi-objective optimization algorithm and performance prediction model. AB-MOEA/D uses a decomposition-based multi-objective optimization algorithm to find the configuration with the objective of minimizing the execution time and cost, where the performance model constructed on the Adaboost algorithm is used to evaluate the fitness of each candidate configuration. Besides, we also present the configuration automatic tuning system with AB-MOEA/D as the optimization engine. The experimental results on six benchmarks with five data sets show that AB-MOEA/D significantly outperforms the previous work in terms of execution time and cost, with average improvements of approximately 35 and 40 percent.

1. Introduction

Spark (Salloum et al., 2016) has become one of the most popular distributed big data computing frameworks. Based on memory calculation, Spark improves the real-time performance of data processing in big data environments while ensuring high fault tolerance and scalability. It supports a range of advanced components, including Spark SQL for structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. These components ensure that Spark is well used in a wide range of domains including graph computing (Bonner et al., 2016), log file analysis (Mavridis and Karatza, 2017), database management (Armbrust et al., 2015), and machine learning (Assefi et al., 2017; Hai and Forouraghi, 2018). An important category of these applications usually runs repeatedly with certain size input data set, which called periodic jobs (Zhibi Yu and Qian, 2018). The benchmarking community has theoretically and practically proven that most long-running applications also can be represented by some periodic jobs (Zhu et al., 2017; Asanovic et al., 2006; Zhu et al., 2014).

The performance requirements of each periodic job are different, choosing the right configuration for them is critical for improving service quality and competitiveness. For instance, a proper configuration can achieve near-optimal performance at a lower cost. On the contrary, a bad configuration will result in several or even tens of times the overhead for the similar performance. However, there are several challenges that must be addressed simultaneously in choosing the optimal configuration for Spark applications. On the one hand, there is a huge configuration space to search. Spark has defined many configuration parameters to support efficient running, and overall performance is highly sensitive to the settings of these parameters. Even the stakeholder with professional knowledge and experience can hardly tune these parameters at the same time for various customized scenarios (Bei et al., 2016). Moreover, the searching process is further complicated by the common practice of deploying Spark applications on cost-effective public cloud, where the cloud configuration parameters that control the allocation of available resources also need to be considered. On the other hand, the optimization objectives of users are variable. Generally, the optimization objective of Spark applications is to reduce execution time or reduce execution costs. But in most cases, users require to satisfy multiple optimization objectives simultaneously, while these objectives may conflict with each other. Therefore, without automated support, it is difficult to select the

☆ Editor: J.C. Duenas.

* Corresponding author.
E-mail addresses: chengguoli@whu.edu.cn (G. Cheng), yingshi@whu.edu.cn (S. Ying), wbingming@whu.edu.cn (B. Wang).

https://doi.org/10.1016/j.jss.2021.111028
0164-1212/

optimal configuration to satisfy adaptability and complexity for different Spark applications.

Studies on the Spark optimal configuration selection problem are probably due to the existence of these types of problems in practice. However, existing solutions do not fully address all the above challenges. Gounaris and Torres (2018) and Petridis et al. (2016) investigated the impact of the most important of the adjustable Spark parameters on the application performance and guided developers on how to proceed to changes to the default values. Then they mapped their experience to a trial-and-error iterative improvement methodology and applied it to the Spark-enabled Marenostrum III (MN3) computing infrastructure. Later, the studies on the optimal configuration choosing problem were continued by Wang et al. (2016), Zhibi Yu and Qian (2018), Nguyen et al. (2018), and several others (Gu et al., 2018; Perez et al., 2018). For a detailed literature review, please refer to related works in Section 8. These works consider the part of performance-related parameters when selecting the Spark optimal configuration, and only involve single-objective optimization. Recently, Zhu et al. (2017) proposed an automatic configuration tuning system for general systems that combines multiple objectives through linear aggregation. But it is difficult to achieve true multi-objective optimization by a simple weighting method. Especially when the gap between the quantitative standards between each optimization objective is large, the allocation of weights will largely determine the direction of optimization.

In this paper, we present a new approach called AB-MOEA/D for the same Spark optimal configuration selection problem by combining multi-objective optimization algorithm based on decomposition (MOEA/D) and performance prediction model constructed by Adaboost algorithm. In the proposed approach, the optimal configuration selection is converted into a multi-objective optimization problem (MOP) with minimizing execution time and cost, which is solved based on the search framework of MOEA/D. The purpose of Adaboost performance prediction model is to evaluate each candidate configuration in AB-MOEA/D.

Overall, the main contributions of this paper are summarized as follows:

- To the best of our knowledge, this is the first approach using multi-objective optimization to search optimal configuration for Spark applications deployed on public clouds. AB-MOEA/D can produce a Pareto-optimal set of solutions to allow users to optimize specific objectives or trade-off between different objectives.
- One of the significant advances of AB-MOEA/D over the state-of-the-art is that it considers more factors related to Spark performance—unprecedented in previous work, including not only application-level but also cloud-level configuration parameters.
- We propose a Spark configuration automatic tuning system with AB-MOEA/D as the optimization engine. It has a highly scalable structure that can flexibly change optimization objectives or search algorithms.
- We conduct comprehensive experiments on the representative benchmark, including six benchmarks with five data sets. The results show that the new proposal significantly outperforms the previous work in dealing with multi-objective configuration tuning.

The rest of the paper is organized as follows. Section 2 discusses the background and motivation. Section 3 describes the AB-MOEA/D proposed. Section 4 gives the implementation of the configuration automatic tuning system with AB-MOEA/D. Section 5 outlines our experimental setup. Section 6 presents the results and analysis. Section 7 gives a discussion about our work. Section 8 reviews related works on configuration tuning and Section 9 concludes the work.

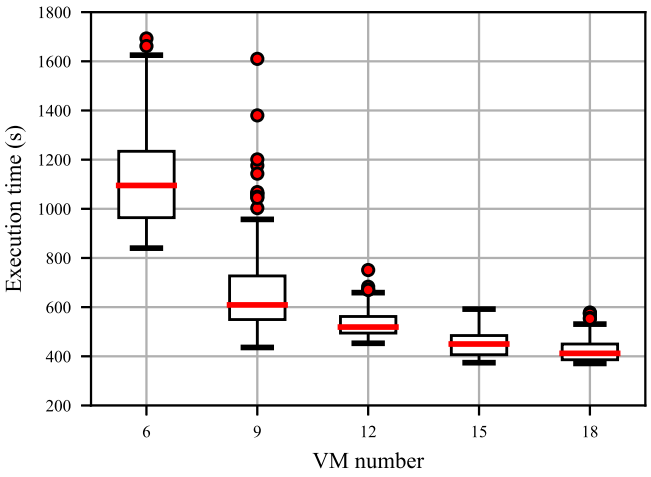


Fig. 1. *TeraSort* execution time with varying cluster size.

2. Background and motivation

2.1. Apache spark

Spark uses memory cache and distributed computing to achieve high performance while inherited high fault tolerance from MapReduce. Coupled with its flexible data structure and powerful functional programming interface, Spark is better than MapReduce in various large-scale data processing programs based on machine learning and iterative analysis. Generally, a Spark cluster consists of one or more Elastic Compute Service (ECS) instances on the cost-effective public cloud.

Users can configure each application to optimize performance by adjusting a large number of performance-related parameters (Li et al., 2018; Nguyen et al., 2017). These parameters can divide into two parts: One is the application-level parameters, including the parameters defined in Spark properties (e.g., *Spark.executor.memory* specifying the memory size of each executor). The other is cloud-level parameters, including parameters that specify the total amount of resources available in the cloud (e.g., the number and type of virtual machines). For more parameter details, please refer to Section 5.2. In the previous work (Zhibi Yu and Qian, 2018; Gu et al., 2018; Bei et al., 2018), the researchers only considered the impact of the former on Spark, but this is not comprehensive. Because when the total amount of available resources is not enough to break the bottleneck of CPU, network bandwidth, or memory, the performance improvement obtained by only adjusting the application-level parameters is limited. Fig. 1 shows the execution time variation of *TeraSort* benchmark with the same application-level configurations on five different cluster sizes where each virtual machine comes with 16 GB RAM and 4 cores in Ali ECS. As seen, there still has much scope for improvement in execution time as the cluster size increases. However, the benefits of adding the number of nodes are not always significant, which means that cloud configuration will affect Spark performance in a non-linear way. Therefore, we should also consider the cloud configuration when optimizing Spark.

Benefits: First of all, a good configuration can significantly improve the performance of Spark applications without excessively increasing or even reducing the monetary cost, while the automatic tuning system can enable users to avoid inefficient and time-consuming manual settings. Second, it is more important to choose a good configuration for periodic jobs. In the long run, users will gain more benefits and the cost of configuration searching can be amortized with the repeated execution of periodic jobs.

Recent studies (Ferguson et al., 2012) report that periodic jobs account for more than 40% of all jobs. Our approach is proposed for periodic jobs.

2.2. Problem formulation

In this paper, the Spark optimal configuration selection problem is regarded as a multi-objective optimization problem. For a given Spark application, our goal is to find the optimal configuration with the objective of minimizing the execution time and cost. Formally, we use $T(\boldsymbol{x})$ and $C(\boldsymbol{x})$ to denote the function of execution time and cost for an application, respectively. Then we can formulate the problem as follows:

$$\begin{aligned} &Minimize\ F(\boldsymbol{x}) = \{T(\boldsymbol{x}), C(\boldsymbol{x})\} \\ &Subjective\ to\ \boldsymbol{x} \in \Omega \end{aligned} \tag{1}$$

where $\boldsymbol{x} = (x_1, \ldots, x_j, x_{j+1}, \ldots, x_n) \in R^n$ is the configuration vector of the feasible configuration space Ω, in which $x_1, \ldots, x_j$ are application-level parameters and $x_{j+1}, \ldots, x_n$ are the cloud-level parameters. As Spark is deployed on a public cloud that charges by time, $C(\boldsymbol{x})$ is related to $T(\boldsymbol{x})$. We use $P(\boldsymbol{x})$ to denote the price per unit time, then $C(\boldsymbol{x})$ can be represented as:

$$C(\boldsymbol{x}) = P(\boldsymbol{x}) * T(\boldsymbol{x}) \tag{2}$$

$P(\boldsymbol{x})$ can be known from the cloud service provider. Knowing $T(\boldsymbol{x})$ of all candidate configurations is essential to solving Eqn (1). Next, we will explain the idea of obtaining $T(\boldsymbol{x})$ of all candidate configurations.

2.3. Searching for optimal configuration

A naive approach for searching the optimal configuration is to traverse the configuration space. Unfortunately, it is hard to work due to numerous configuration parameters and long accumulative running time of applications (Zhu et al., 2017). Recently, researchers have found that using evolutionary algorithms can effectively solve these problems, such as genetic algorithm (GA) (Zhibi Yu and Qian, 2018; Bei et al., 2018; Luo and Fu, 2019) and particle swarm optimization algorithm (PSO) (Wang et al., 2017). These algorithms explore the unknown area of the configuration space according to the information of existing configurations in an iterative manner until the optimal one is found. However, since the performance of each candidate configuration is unknown without practical running, we still need to measure new configurations in each iteration of the evolution algorithm, which would be a departure from the original intention of solving the problem in a light-weight way. Therefore, the key to applying evolutionary algorithms is how to evaluate the performance of each candidate configuration quickly and accurately.

Performance prediction models based on machine learning can mine problem-specific information from seemingly chaotic data, such as the mapping relationship between performance metrics and configuration space. We can build a sufficiently accurate model to predict the performance of Spark applications with the given configuration, which is much faster than an approach that requires executing the application. Currently, the idea of combining the search algorithm with the performance prediction model has been widely applied to the automatic search optimal configuration of the big data computing framework and has achieved good results (Zhibi Yu and Qian, 2018; Bei et al., 2016). Fig. 2 provides an overview of the joint process. In step 1, it starts with an initial configuration set, which can be generated by random or heuristic methods. In step 2, we use the performance model to predict the performance of each configuration and take the prediction as the fitness value during the search process (minimization problem). Next, in step 3, the evolutionary algorithm judges whether to end the process according to a predefined termination condition. If accepted, output the optimal configuration, otherwise, go to the next step. In step 4, the evolutionary algorithm applies a series of genetic operations, including one or more of selection, crossover, and mutation operators, to generate a new set of configurations and re-evaluate. Repeat steps 2 to 4 until the optimal configuration is found.

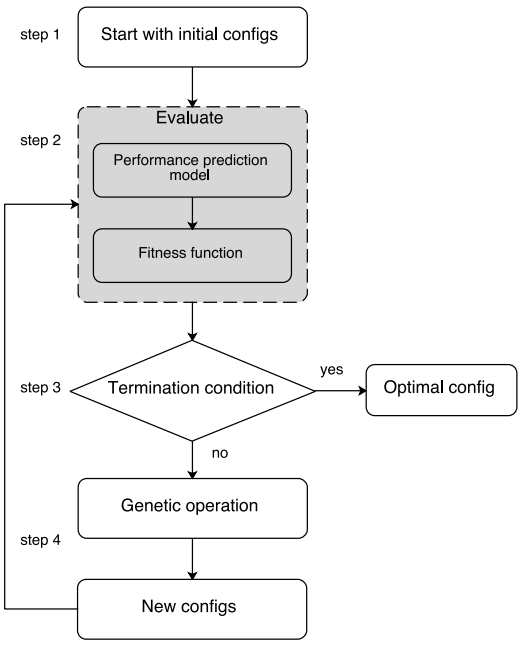


Fig. 2. The process of evolutionary algorithm combined with performance model.

The advantage of combining multi-objective optimization algorithms and performance prediction models is that the configuration space can be regarded as a black box, which allows us to pay more attention to the relationship between configurations and optimization objectives. In this way, we can tune the configuration without having deep insight into the parameters and the potential interaction among parameters. In this paper, we along this thought to design a multi-objective evaluation algorithm for searching Spark optimal configurations.

3. The proposed AB-MOEA/D

Our goal is to find the optimal configuration for Spark applications running on the public cloud. Since this "optimal" involves several conflicting objectives, we use multi-objective optimization to solve this problem. The framework of our proposed approach is shown in Algorithm 1. The basic idea of AB-MOEA/D is to combine decomposition-based multi-objective optimization algorithm (MOEA/D) with performance prediction model. The MOEA/D aims at exploring the configuration space, whereas the model constructed by Adaboost algorithm is introduced to evaluate the fitness value of each solution. AB-MOEA/D decomposes the problem into N single objective optimization subproblems and solves them simultaneously. In principle, any aggregation method can be used for this purpose. For convenience, the Tchebycheff approach is adopted in this paper. At each generation, AB-MOEA/D maintains two populations P and EP, where the former is composed of the current solution of each subproblem, and the latter is applied to store non-dominated solutions found during the search.

In AB-MOEA/D, the main algorithm components include population initialization (line 1), population evaluation (line 6), genetic

Algorithm 1 Framework of the proposed AB-MOEA/D

Input:
PM (performance prediction model),
N (the number of subproblem; the size of population),
T (the size of the neighborhood of each subproblem),
$\lambda^1, ..., \lambda^N$ (a set of N weight vectors)
1: Initialize population P with N random solutions and set $EP = \varnothing$
2: Compute the Euclidean distance between any two weight vectors and obtain T closest weight vectors to each weight vector
3: Initialize Z that stores the best value found so far for each objective
4: **while** the termination condition is not fulfilled **do**
5: **for** $i = 1, 2, ..., N$ **do**
6: Apply model-based evaluation function to calculate the fitness value of each solution and determine the Pareto dominance relationship
7: Randomly select two indices from the neighbor of each subproblem and apply genetic operators to generate an offspring population
8: Update Z, Neighboring Solutions, and EP
9: **end for**
10: **end while**
11: **return** Non-dominated valid solutions in EP
Output:
Non-dominated valid solutions in EP

operators (line 7), and population update (line 8). In the following sections, the preceding main components will be described in detail.

3.1. Population initialization

In our case, the individuals of the population represent Spark configurations which consist of application-level and cloud-level parameters. Among them, application-level parameters are related to the application settings, which define workload characteristics and they must be provided during the job execution. Specifically, application-level parameters include parameters defined in Spark properties and need to be configured separately for each application. Cloud-level parameters are what we must set when deploying Spark on the public cloud, including the number and type of virtual machines. Fig. 3 shows an example of the above parameters. Note that here we only use two parameters to represent a cloud configuration. This is because in current available cloud services, other cloud parameters, such as CPU speed per core and average RAM per core, are determined by the type of virtual machine. However, this does not affect the usability of the proposed method. When cloud services provide more configurable parameters in the future, all we need to do is to add them to individuals.

For all configuration parameters, they select a value within their respective ranges randomly. In this way, N random valid individuals are generated in the initial population P, which can cover the configuration space as uniform as possible. At the beginning of the algorithm, the EP is set to empty.

3.2. Model-based fitness function

In the multi-objective optimization algorithm, the fitness function is used to evaluate the quality of individuals. The selection of fitness functions will affect the convergence speed of the algorithm and whether the optimal solution can be found. For the problem of Spark optimal configuration selection, our goal is to find the configuration to minimize the execution time and cost. Considering the time and cost are both non-negative, we directly use the objective function as the fitness function. However, as we mentioned in Section 2.3, since the overhead of measuring a set of Spark configurations in each iteration is very high, we construct a performance model to predict the execution time and cost of each candidate configuration. Its main idea is to use a number of observations from the real Spark system to train a model via the machine learning algorithm. The model takes Spark configurations as input and outputs the performance prediction.

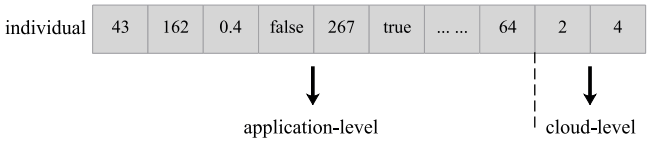


Fig. 3. Example of individual representation.

In our case, the performance model is built on the Adaboost algorithm which is a kind of ensemble learning in machine learning. It utilizes the boosting method to combine multiple learners for learning task. Compared with other machine learning algorithms, the Adaboost algorithm has significant advantages in prediction accuracy and resistance to over-fitting when dealing with high-dimensional complex problems, which is proved in 6.1. Moreover, it does not make any assumptions about the configuration parameters and has less algorithmic settings. The core principle of Adaboost is to convert a sequence of weak learners to strong learners. Intuitively, a weak learner is just slightly better than random guess, while a strong learner is very close to perfect performance.

Algorithm 2 The procedure of Adaboost algorithm

Input:
Data set $T = (\boldsymbol{x_1}, t_1), (\boldsymbol{x_2}, t_2), ..., (\boldsymbol{x_m}, t_m)$,
Base learning algorithm $\mathcal{L}_{tree}$,
Number of learning rounds LR,
Learning rate α
1: $\mathcal{T}_1(\boldsymbol{x}) = 1/m$ //Initialize the weight distribution
2: **while** *average loss* < 0.5 or $t \leq LR$ **do**
3: $h_t = \mathcal{L}(T, \mathcal{T}_t)$ //Train a regressor h_t from T under distribution $\mathcal{T}_t$
4: $L_i = |t_i^{(p)}(\boldsymbol{x_i}) - t_i|$ //Pass every sample through this regressor to obtain a prediction $t_i^{(p)}(\boldsymbol{x_i})$ and calculate the loss
5: Calculate an average loss $\bar{L}$
6: $\beta = \frac{\bar{L}}{1-\bar{L}}$ //β is the measure of confidence in the regressor
7: $\alpha_t = \alpha \ln \frac{1}{\beta}$ //Update the weights
8: $\mathcal{T}_{t+1}(\boldsymbol{x}) = \frac{\mathcal{T}_t(\boldsymbol{x})\beta^{(1-L_i)}}{Z_t}$ //Update the distribution, where Z_t is a normalization factor which enables $\mathcal{T}_{t+1}$ to be a distribution
9: **end while**
Output:
$f(\boldsymbol{x}) = Combine\_Outputs(h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), ..., h_t(\boldsymbol{x}))$.

To build the performance model, we first need to construct the training set T. T is a matrix, each row of T is the following vector:

$$v_k = (x_{1k}, x_{2k}, \ldots, x_{nk}, t_k), \quad k = 1, 2, \ldots, m \tag{3}$$

where v_k is the kth vector, x_{ik} is the ith Spark configuration parameter of the kth observation, n is the total number of Spark configuration parameter, t_k is the execution time of the kth observation, and m is the total number of vectors (observations or training samples) in matrix T. The size of m will be determined in Section 5.3.

Subsequently, we input the matrix T to Adaboost algorithm to build a performance model. The first step is to invoke the base learning algorithm (CART decision tree used in this paper) on the original training data and generate the weak regressor. Its weight depends on the average loss of all training samples. Then AdaBoost adjusts the sample distribution such that new weak regressor pays more attention to samples of incorrect predictions in the next round. After a number of rounds, a sequence of weak regressors and their corresponding weights are generated. Finally, combining these weak regressors to form the final performance prediction model. The formal description of modeling process is illustrated by Algorithm 2.

After we obtain the performance prediction model, we can easily know the execution time and cost of each candidate configuration and use them as fitness values to guide the search process in the optimization algorithm.

3.3. Genetic operators

The basic idea of evolutionary algorithm is based on the theory of natural selection, which can screen out more suitable individuals during the evolution process and retain their outstanding characteristics. To evolve the population, two aspects are essential: the mating of the individuals and random changes in the offspring. In AB-MOEA/D, these aspects are respectively translated into crossover and mutation operators.

The crossover operator generates new individuals by combining the genetic material of two or more randomly selected parents. In this work, the one-point crossover is employed. We first randomly select two individuals as parents from the neighbor of a subproblem. Second, selecting a position from 0 to the length of the individual randomly, which divides two parents into two parts. Third, exchanging the segments of the two-parent individuals behind the selected position to produce two offspring. The top half of Fig. 4 shows an example of the crossover operator. After then, we compare the two offspring according to the Pareto dominance and choose the non-dominated individual as the final offspring. If dominance cannot be determined, one is randomly selected. The use of the crossover operator is controlled by a probability p_c.

The mutation operator randomly changes individuals by flipping the bits. Its main goal is to introduce new genetic material to the population and increase the diversity of genes. We apply the one-point mutation to the offspring obtained through the crossover operator. Similarly, we first select a position randomly between 0 and the length of the individual. Then, generating a new value randomly within the specified range to replace the original value. The bottom half of Fig. 4 illustrates an example of the mutation operator. The use of the mutation operator is controlled by a probability p_m.

Generally, a group of offspring individuals will be produced by genetic operation. But in the genetic process, the value of a certain element in the offspring individual may exceed the boundary, which is called an invalid individual or an infeasible solution. How to deal with these invalid individuals is crucial to improving the efficiency of the algorithm. Recently, Wang et al. (2020) proposed a boundary optimization to solve the problem of uneven distribution of individuals caused by improper boundary processing. Its core idea is to utilize the theory of double-sided specular reflection to generate valid individuals uniformly distributed within the boundary. We adopt this method and the specific calculation formula is as follows:

$$x'_j = \begin{cases} x_j^{min} + mod(x_j^{min} - x'_j, x_j^{max} - x_j^{min}) & if\ x'_j < x_j^{min} \\ x_j^{max} - mod(x'_j - x_j^{max}, x_j^{max} - x_j^{min}) & if\ x'_j > x_j^{max} \\ x'_j & otherwise \end{cases} \quad (4)$$

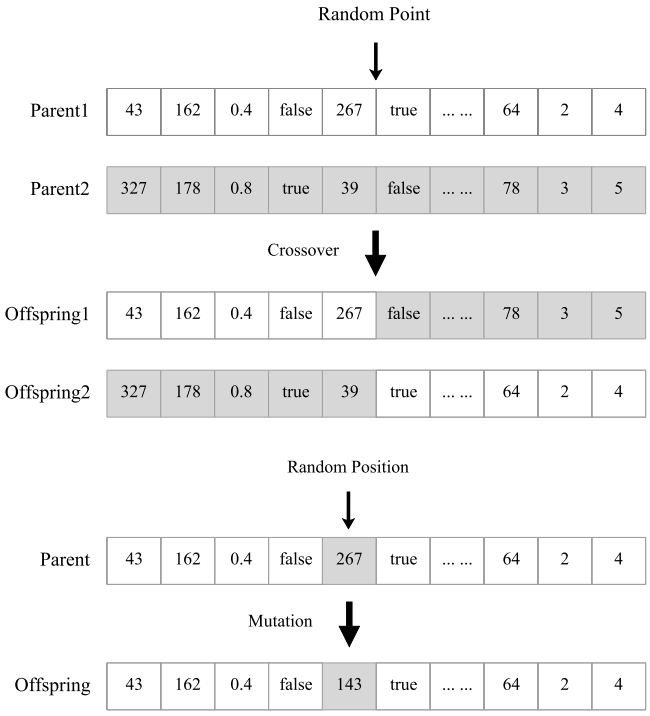


Fig. 4. Example of the genetic operators.

where x'_j represents the new value of the jth element in a evolved individual, x_j^{min} and x_j^{max} represent the corresponding boundary limit of the jth element.

3.4. Population update

The evolution will keep better individuals in the population as much as possible. In each iteration, new individuals generated through genetic operators are compared with original individuals, and the worse individual is replaced with the better individual. AB-MOEA/D achieves the purpose of population update through three steps. The first step is to update Z. Compare the value of the new individual on each objective with the value in the previous Z to ensure that the best value found so far for each objective is stored in Z. The second step is to update the neighbor. For all the neighbors of each subproblem, if the new individual is better than the original individual corresponding to the neighbor subproblem, update the neighbor. The third step is to update EP. If EP is empty, add a new individual to EP. If EP is not empty, remove all individuals dominated by the new individual in EP. If no individual in EP can dominate the new individual, then the new individual is added to EP.

The population is evolved continually until the termination condition is fulfilled, and then EP is output. Note that the termination condition can be freely defined, such as the maximum number of iterations or the specified objective function value.

4. Implementation

In this section, we discuss the implementation details of Spark configuration automatic tuning system with AB-MOEA/D as shown in Fig. 5. It has four modules.

1. **Workload Generator:**Workload Generator generates application workloads. Users can run benchmark workloads commonly used for stress testing like Hibench, or, run actual application workloads on it. To use the tuning system,

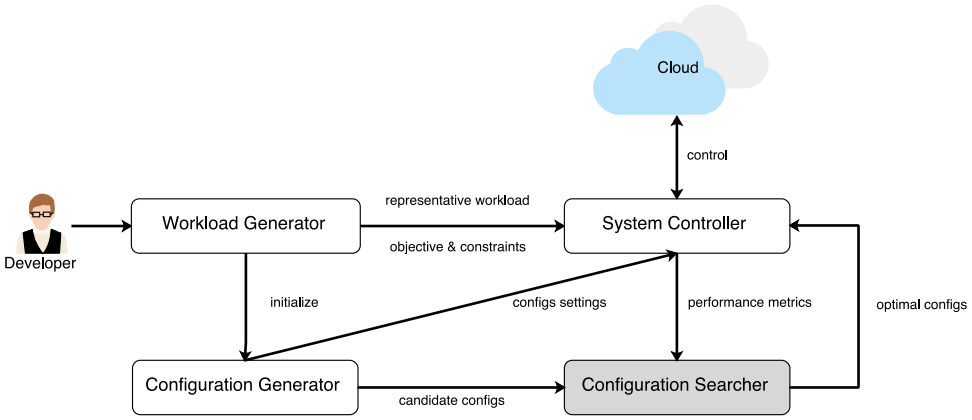


Fig. 5. Architecture of Spark configuration automatic tuning system with AB-MOEA/D.

users need to provide a representative application (see Section 5.1), the optimization objective (e.g. minimizing execution time and cost), and the constraints (e.g. maximum execution time or cost budget).

2. **System Controller:** System Controller is an adaptation layer that handles the heterogeneity to control the clouds. After the user submits the workload, the System Controller creates the operating environment of the application in the cloud and provides the user with a uniform operation API and common configuration mapping. The benefit of standardization is to make the tuning system cope with different clouds. The System Controller also records the status of the workload runtime, including the current configuration and performance metrics.
3. **Configuration Generator:** Configuration Generator produces a set of candidate configurations from the sample space which consists of performance-related parameters. Each parameter is within the specified range to ensure that the generated configuration is valid. These configurations are submitted to the System Controller for repeated benchmarking workloads while being recorded in the Configuration Searcher.
4. **Configuration Searcher:** Configuration Searcher is the core of the tuning system, which is built on the AB-MOEA/D search algorithm. Based on the configuration information of the workload and the corresponding performance metrics from the System Controller, AB-MOEA/D automatically searches for the optimal configuration that meets the user's requirements. Once the optimal configuration is found, Configuration Searcher submits it back to System Controller to configure the workload with optimized performance.

The Spark configuration automatic tuning system has a highly flexible and extensible architecture, in which components are loosely coupled. On the one hand, each component only interacts through configuration information and performance metrics, which minimizes the dependency between components. On the other hand, relying on the uniform operation API, the system allows different optimization algorithms to be applied to the tuning process for handling the actual scenario of users. With such an architecture, we only need to change slightly to the Workload Generator and System Controller when coping with new workloads and optimization objectives in the future.

5. Experimental setup

We build the experimental platform on Ali general purpose Elastic Compute Service (ECS) to deploy the benchmarks, as shown in Table 1. The experimental environment includes Linux Centos7, Spark 2.4.7, and Hadoop 2.9.1. For the distributed computing framework involved in the experiment, we adopt the common one-master, multiple-slave structure. The rest of this section gives details of the experimental setup, including representative applications, configuration spaces, and algorithm implementation details.

Table 1
Characteristics of the VMs.

VM type	Core	Memory	Net bandwidth	Price
g6.large	2	8 GB	1 Gbps	0.5 ¥/h
g6.xlarge	4	16 GB	1.5 Gbps	1 ¥/h

Table 2
Spark benchmarks considered in this study.

Application	Input data size
WordCount(WC)	120, 140, 160, 180, 200 (GB)
TeraSort(TS)	20, 40, 60, 80, 100 (GB)
Kmeans(KM)	60, 70, 80, 90, 100 (million samples)
Bayes(BA)	12, 14, 16, 18, 20 (million pages)
Linear(LR)	0.8, 0.9, 1, 1.1, 1.2 (million examples)
NWeight(NW)	30, 35, 40, 45, 50 (million edges)

5.1. Representative applications

Hibench (Huang et al., 2010) is a widely used benchmark for evaluating Spark's framework. It consists of a series of workloads from which we choose six to evaluate the proposed approach, as shown in Table 2. *WordCount* counts the number of occurrences of each word in a document, which generates the input data using RandomTextWriter. *TeraSort* is a standard sort benchmark for big data analytics, which generates the input data by Hadoop TeraGen example program. *Kmeans* is an iterative clustering analysis algorithm in Spark-MLlib, which generates the input data by GenKMeans Data set based on Uniform Distribution and Gaussian Distribution. *Bayes* is a multi-class classification algorithm that is implemented in Spark-MLlib. This workload uses the automatically generated documents whose words follow the Zipfian distribution. *Linear regression* determines the quantitative relationship between variables, which is implemented in Spark-MLlib with ElasticNet. Its input data is generated by LinearRegressionDataGenerator. *NWeight* is an iterative graph-parallel algorithm implemented by Spark GraphX, which computes associations between two vertices that are n-hop away. These workloads represent a broad enough set of typical Spark workload behaviors, including disk-intensive, memory-intensive, CPU-intensive, and I/O intensive.

Table 3
Description of the 39 application-level configuration parameters.

Parameter	Description	Default	Range
Spark.executor.memory (EM)	Amount of memory to use per executor process, in GB.	1	1-VM.Max
Spark.executor.cores (EC)	The number of cores to use on each executor.	1	1-VM.Max
Spark.executor.memoryOverhead	The amount of off-heap memory to be allocated per executor in cluster mode, in MB.	EM*0.1 (min 384)	EM*0.1–EM*0.5
Spark.driver.memory (DM)	Amount of memory to use for the driver process, in GB.	1	1-VM.Max
Spark.driver.cores (DC)	Number of cores to use for the driver process.	1	1-VM.Max
Spark.driver.maxResultSize	Limit of total size of serialized results of all partitions for each Spark action (e.g. collect) in bytes in MB or GB.	1G	DM*0.5–DM*1.0
Spark.driver.memoryOverhead	The amount of off-heap memory to be allocated per driver in cluster mode, in MB.	DM*0.1 (min 384)	DM*0.1–DM*0.5
Spark.reducer.maxSizeInFlight	Maximum size of map outputs to fetch simultaneously from each reduce task, in MB.	48	1–500
Spark.maxRemoteBlockSizeFetchToMem	The remote block will be fetched to disk when size of the block is above this threshold in bytes.	512	256–1024
Spark.broadcast.blockSize	Size of each piece of a block for TorrentBroadcastFactory, in MB.	4	1–500
Spark.broadcast.compress	Whether to compress broadcast variables before sending them.	True	True/false
Spark.broadcast.checksum	Whether to enable checksum for broadcast.	True	True/false
Spark.default.parallelism	Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user.	Total EC	2–500
Spark.files.maxPartitionBytes	The maximum number of bytes to pack into a single partition when reading files, in MB.	128	1–512
Spark.files.openCostInBytes	The estimated cost to open a file, measured by the number of bytes could be scanned at the same time, in MB.	4	1–128
Spark.storage.memoryMapThreshold	Size in bytes of a block above which Spark memory maps when reading a block from disk, in MB.	2	1–128
Spark.python.worker.memory	Amount of memory to use per python worker process during aggregation, in MB.	256	1–500
Spark.python.worker.reuse	Reuse Python worker or not.	True	True/false
Spark.shuffle.compress	Whether to compress map output files.	True	True/false
Spark.shuffle.file.buffer	Size of the in-memory buffer for each shuffle file output stream, in KB.	32	1–500
Spark.shuffle.io.maxRetries	Fetches that fail due to IO-related exceptions are automatically retried if this is set to a non-zero value.	3	1–5
Spark.shuffle.service.index.cache.size	Cache entries limited to the specified memory footprint, in MB.	100	50–500
Spark.shuffle.spill.compress	Spark.shuffle.spill.compress.	True	True/false
Spark.shuffle.service.enabled	Enables the external shuffle service.	False	True/false
Spark.dynamicAllocation.enabled	Whether to use dynamic resource allocation.	False	True/false
Spark.rdd.compress	Whether to compress serialized RDD partitions.	False	True/false
Spark.serializer	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form.	JavaSL	JavaSL/KryoSL
Spark.memory.fraction	Fraction of used for execution and storage.	0.6	0.5–0.9
Spark.memory.storageFraction	Amount of storage memory immune to eviction.	0.5	0.5–0.9
Spark.memory.offHeap.enabled	If true, Spark will attempt to use off-heap memory for certain operations.	False	True/false
Spark.speculation	If set to true, performs speculative execution of tasks.	False	True/false
Spark.speculation.multiplier	How many times slower a task is than the median to be considered for speculation.	1.5	1.2–2.0
Spark.speculation.quantile	Fraction of tasks which must be complete before speculation is enabled for a particular stage.	0.75	0.5–0.9
Spark.task.cpus	Number of cores to allocate for each task.	1	1-EC.Max
Spark.task.maxFailures	Number of failures of any particular task before giving up on the job.	4	1–5
Spark.rpc.message.maxSize	Maximum message size (in MB) to allow in control plane communication.	128	1–500
Spark.io.compression.codec	The codec used to compress internal data such as RDD partitions, and so on.	Snappy	Snappy, lzf,lz4
Spark.io.compression.lz4.blockSize	Block size used in LZ4 compression, in KB.	32	2–256
Spark.io.compression.snappy.blockSize	Block size used in snappy, in KB	32	2–256

5.2. Configuration spaces

The Spark configuration parameter consists of two disjoint sets: application-level parameters and cloud-level parameters.

Application-level parameters: Spark provides users with 16 classes of configuration parameters, involving application properties, runtime environment, shuffle behavior, etc. Since the goal of this paper is to optimize performance, parameters that irrelevant to performance are not within our consideration. For example, *Spark.app.name* defines the name of the running task, *Spark.ui.retainedJobs* indicates how many tasks are remembered by Spark UI and state APIs before garbage collection. There are other parameters, such as *Spark.driver.extraClassPath*, which are usually set to none because they are left to the user. Listed in Table 3 are a set of Spark-specific parameters we selected that are closely related to application performance. The default column represents the Spark default setting and the range column represents the value range of parameters in the experiment (based on our expertise and the cluster affordability). The reason for choosing these configuration parameters is as follows: These parameters cover the key aspects of the Spark runtime such as compression and serialization, memory management, execution behavior, and these key aspects ultimately determine the performance of the Spark application.

Cloud-level parameters: As discussed earlier, we represent the cloud configuration with the number and type of virtual machines. we used large and xlarge instance sizes each with 2 and 4 cores per machine respectively. For each instance size, the total number of cores ranges from 24 to 72, as shown in Table 4. Considering the cost constraints, we do not pick up more combinations. However, we make sure the configurations we choose are reasonable for our objectives. Because there is no significant improvement in execution time even with more expensive computing resources.

5.3. Algorithm implementation details

The parameters involved in AB-MOEA/D can be divided into two parts: one part is the configuration search, specifically the parameters included in the decomposition-based multi-objective optimization algorithm. The other part is model construction, specifically the parameters included in the Adaboost algorithm.

Table 4
Cloud-level configuration parameters.

VM type	Number of VMs				
g6.large	12	18	24	30	36
g6.xlarge	6	9	12	15	18
Number of cores	24	36	48	60	72

The method to determine these parameters is to combine the recommendations in the relevant research with our attempts during the experiment to ensure that AB-MOEA/D can achieve the best results in all benchmarks. Next, we will introduce these two parts of parameters.

In AB-MOEA/D, the following parameters are used: The termination criterion is the predefined maximum number of iterations, which is set to 300. The size of the population is set to 150. The crossover probability $p_c = 0.9$ and the mutation probability $p_m = 0.1$. T is set to 10 for all the test applications. The setting of N and $\lambda^1, \lambda^2, \ldots, \lambda^N$ is controlled by a parameter H. More precisely, $\lambda^1, \lambda^2, \ldots, \lambda^N$ are all the weight vectors in which each individual weight takes a value from

$$\{\frac{0}{H}, \frac{1}{H}, \ldots, \frac{H}{H}\}$$

Therefore, the number of such vectors is

$$N = C_{H+m-1}^{m+1}$$

Following the suggestions in Zhang and Li (2007), for optimization problem with two objectives, $H = 149$, and therefore $N = 150$.

For the performance prediction model, we set the CART decision tree as the base learner of the Adaboost algorithm. The number of base learners and learning rate are set to be 100 and 0.01 respectively. As high performance is obtained by Adaboost model under this setting, we do not make any adjustments. For each benchmark, we collect 2500 randomly generated samples from the configuration space as the training set and another 500 samples as the test set. Each sample consists of 41 valid parameter values. The way we determine the training set size is to add a fixed number of samples to the training set iteratively until the accuracy of the model no longer improves significantly. The training samples are collected separately, that is, no applications are running concurrently so as not to interfere with the measurements.

In our case, the AB-MOEA/D algorithm, Adaboost algorithm and derived tuning system are implemented by Python 3.7.

6. Evaluation

In this section, we start by verifying the effectiveness of the Adaboost performance prediction model. We then compare AB-MOEA/D with the state-of-the-art BestConfig. Next, we report the overhead of AB-MOEA/D. Finally, we discuss some insights about AB-MOEA/D through specific benchmarks.

6.1. Model accuracy

One of the main concerns about AB-MOEA/D is whether the performance model can accurately predict the execution time of the Spark applications with a given configuration, which is crucial for searching for the optimal configuration. To evaluate the accuracy of the performance model, we use relative error in statistics as the evaluation metric. Compared with absolute error, it can make more reasonable judgments on the reliability and credibility of the prediction results. The formula is as follows:

$$err = \frac{|pre - mea|}{mea} \times 100\% \quad (5)$$

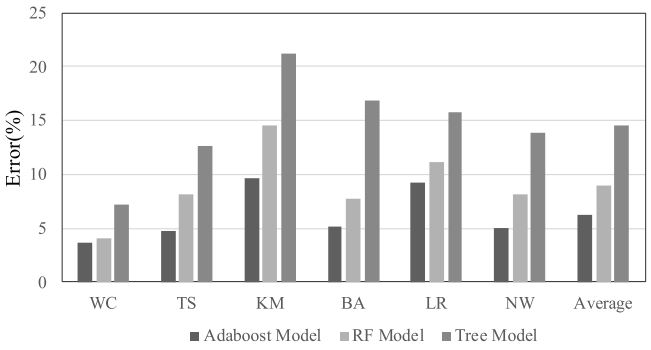


Fig. 6. The average prediction errors of models constructed by Adaboost, random forest, and two-stage tree.

where *pre* is the execution time predicted by Adaboost model, *mea* is the measured execution time. We predict and measure the execution time of the Spark application under a configuration 10 times (to eliminate bias caused by computer hardware, networks, etc.).

We compare performance prediction models constructed by Adaboost algorithm with two models that have been well applied in the Spark single-objective optimization approach. One is a random forest model (Zhibi Yu and Qian, 2018; Bei et al., 2018) and the other is a two-stage tree model (Wang et al., 2016). The settings of these two models, including the size of samples and algorithm parameters, are fully according to their developers. Since the accuracy will be verified on a set of test samples, we calculate the average of the relative errors of all test samples to reflect the overall error of the model on each benchmark.

Fig. 6 shows the prediction error of the models constructed by three algorithms on six benchmarks. As seen, Adaboost model shows significant improvements over the other two models on all benchmarks. Specifically, the average errors of the Adaboost model are 9.7% and 3.6% in the worst and best cases, respectively. On average of six benchmarks, the error is only 6.23%, and the other two models are 8.97%, 14.57%, respectively. This phenomenon can be explained as follows. When dealing with high-latitude and feature-related issues, it is usually hard for decision tree models to learn complex mapping relationships. Besides, it is more sensitive to the distribution of training samples. Even if there is a little disturbance, the structure of the decision tree will change, which will affect the final prediction result. Both Adaboost and random forest can integrate the results of multiple decision trees to produce predictions. The difference lies in ensemble methods. Adaboost pays more attention to the samples that are not correctly predicted and gives more weight to learners with high prediction accuracy, and then generates prediction results through weighted average, which can effectively reduce the deviation in prediction. Random forest introduces random attribute selection on the basis of Bagging algorithm, and then generates prediction results through majority voting or average. This ensemble method determines that random forest is more concerned about reducing the variance of the prediction results. Therefore, in the case of accuracy priority, the Adaboost model performs better. This is also evident from the error distribution. Fig. 7 shows six scatter plots produced by 150 measurements and 150 predictions generated for six applications with 150 randomly selected Spark configurations. As we can see, the predictions and the measurements are scattered near the trend line without the obvious outliers, which means the Adaboost model is fairly accurate across the entire configuration space. The results demonstrate Adaboost model is accurate enough to apply for searching the optimal configuration with performance improvement.

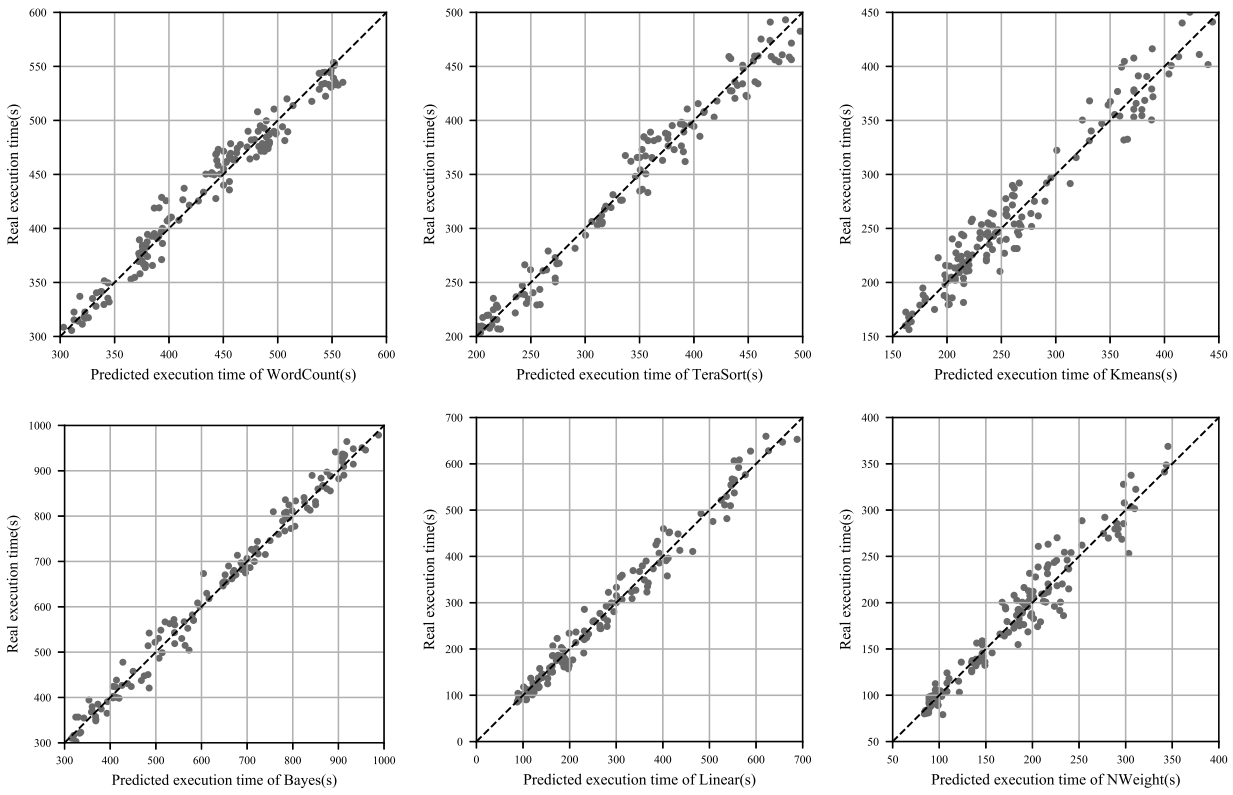


Fig. 7. Error distribution illustrating prediction versus measurement for 150 randomly selected Spark configurations for six applications.

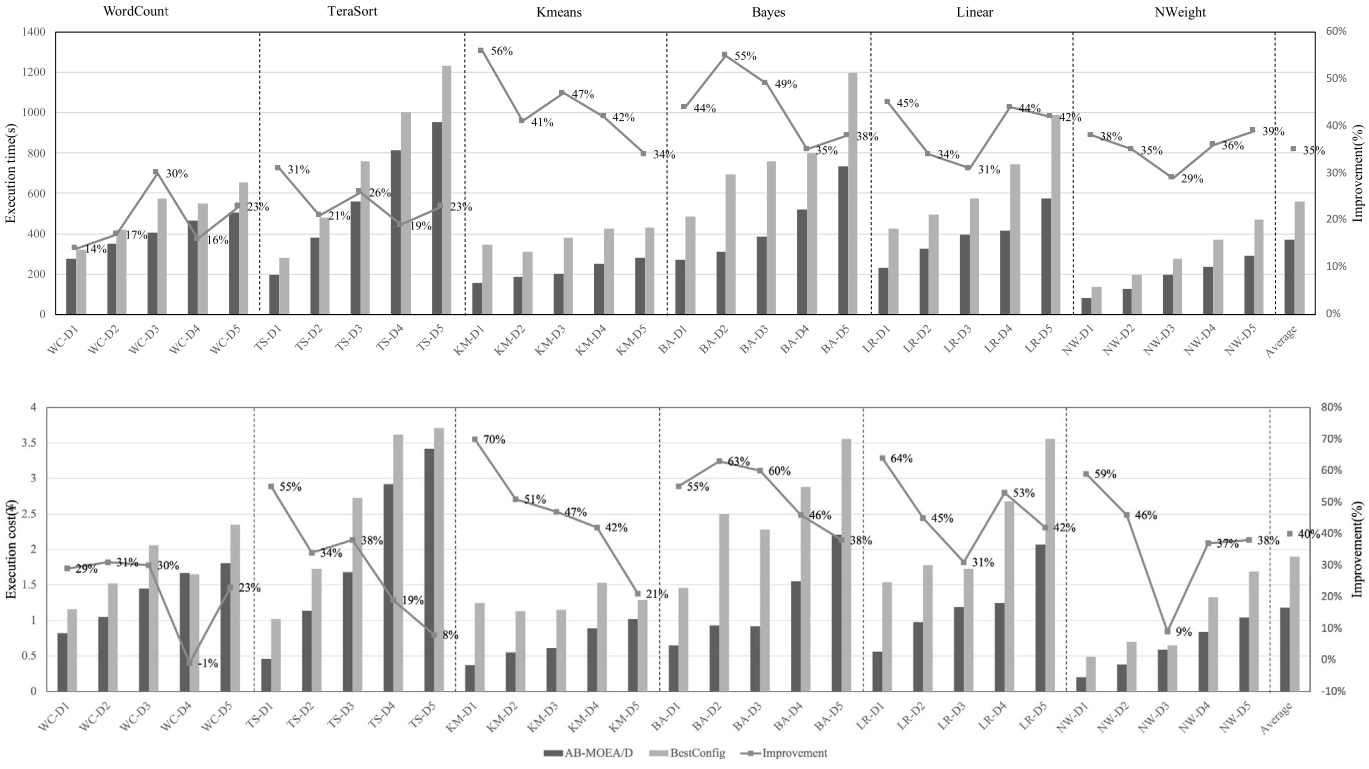


Fig. 8. Objective comparison and improvement percentage of AB-MOEA/D with BestConfig.

6.2. Comparing AB-MOEA/D to the state-of-the-art BestConfig

In this section, the proposed AB-MOEA/D is compared to the state-of-the-art optimization framework BestConfig (Zhu et al., 2017). BestConfig is an automatic optimization configuration system, which can determine the best configuration for one or more performance objectives. The core idea of BestConfig includes two

parts: one is the Divide & Diverge Sampling that selects the effective information point from the sample space, and the other is the Recursive Bound & Search that selects the best configuration from the configuration space. To compare fairly, we re-implement BestConfig in our experimental environment and perform the same experiments as what we do.

Like other multi-objective optimization algorithms, the output of AB-MOEA/D is a set of valid solutions, also known as the Pareto front. The solutions contained in the Pareto front can be non-dominated by any other solution, that is to say, it is the optimal solution on at least one objective. The necessary thing of analyzing the experimental results is to determine the solution that meets our requirements from this set of non-dominated solutions. There are many methods (Zio and Bazzo, 2012) that have been proposed on how to choose the preferred solution in the previous work. In this paper, we use the weighted sum of the objectives (Marler and Arora, 2010) to compare the results of the optimization algorithms. Since we believe that the two objectives under consideration are equally important, we set them the same weight:

$$ws = \frac{T - T_{min}}{(T_{max} - T_{min}) \times 2} + \frac{C - C_{min}}{(C_{max} - C_{min}) \times 2} \tag{6}$$

where T and C are the objective values for a solution, and T_{max}, T_{min}, C_{max}, C_{min} are minimal and maximum values of each objective in the Pareto front. We choose the smallest ws as the optimal solution.

Fig. 8 shows the experimental results of AB-MOEA/D and BestConfig on all benchmarks. The horizontal axis represents six applications with five input data sets, e.g., *WC-D1* corresponds to the *WordCount* with 120G data. The optimized result is plotted by the bar graphs, while the line graphs present the improvement percentages of AB-MOEA/D compared to BestConfig. As seen, AB-MOEA/D shows significant improvements over BestConfig on $6 \times 5 = 30$ cases. Specifically, the improvement of AB-MOEA/D in terms of execution time ranges from 14% to 56%, with an average of 35%. The improvement in terms of execution cost ranges from -1% to 70%, with an average of 40%.

Now we are going to analyze the reasons BestConfig performs poorly. BestConfig merges multiple conflict objectives into a single objective in a weighted summation, which leads to a large distance from the searched solution to the Pareto front. This is interpretable because the core of BestConfig's search engine is still a scalar objective evolution algorithm (Zhang and Li, 2007), which finds one optimal solution by comparing the objective function values between solutions. But in MOPs, the domination does not define the complete order between solutions in the objective space. Benefits from the multi-objective search framework based on decomposition, AB-MOEA/D can find a set of non-dominated solutions as diverse as possible that are closer to the Pareto front for MOPs, allowing stakeholders to choose according to their requirements. Besides, in actual operation, the weight assignment of the scalar optimization algorithm is very important to evaluate the quality of the solution. There is no good way in BestConfig to adjust the weight between the objectives except the trial method. We try a variety of combination methods in the experiment, including linear combination or adding symbolic functions, which are difficult to achieve good performance. Moreover, due to the non-commensurable among multiple objectives, a larger weight is often needed to balance the dimensions between each objective. The objective with greater weight will inevitably have a greater impact on the entire objective function. This does not meet the purpose of the multi-objective optimization procedure.

Table 5
Time cost.

Benchmark	Collection (h)	Modeling (s)	Searching (m)
WordCount	174	1.5	14
TeraSort	189	1.6	14
Kmeans	145	1.5	13
Bayes	156	1.2	17
Linear	154	1.3	12
NWeight	115	1.2	12

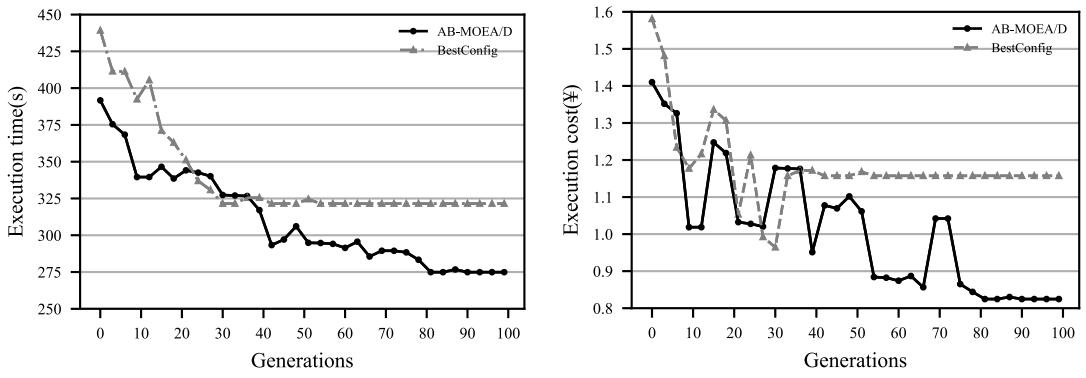
6.3. Overhead

In AB-MOEA/D, the number of iterations required for algorithm convergence will affect the speed of the entire search process. To provide a clearer analysis, we present a comparison between AB-MOEA/D and BestConfig. Fig. 9 shows the evaluation of the objective changes with the number of iterations. As we can see, the maximum number of iterations required for AB-MOEA/D to achieve the minimum value is around 80. For BestConfig, it generally has a stable objective value of around less than 50 generations. This phenomenon may be attributed to the fact that BestConfig gets stuck in sub-optimal areas because of the huge search space. The figure shows only the worst and best cases, but we can observe similar results in all other cases.

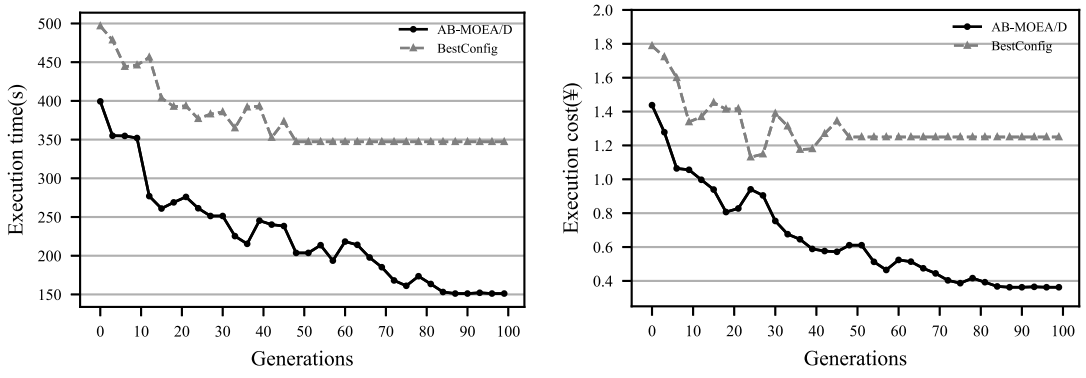
Now we illustrate the overhead required for AB-MOEA/D to complete the entire optimization process. It consists of three parts: collecting historical data, establishing the performance model, and searching for the optimal configuration. The time unit of each part is marked in the first of each column in Table 5. We can see that the main overhead of AB-MOEA/D occurs in the collection of historical data, with a range of 115 h to 189 h. In comparison, the overhead of building a performance model and searching for the optimal configuration is much smaller than the collecting, only a few minutes or even a few seconds. There may be a question: Is it worth spending a long time optimizing the configuration? We believe that although this overhead is relatively large, they are only one-time expenses. In terms of long-term benefits, it is worth paying these expenses to optimize the configuration of periodic Spark jobs, which can make us save more time and monetary cost. This is because periodic jobs are usually repeated for several weeks or months. The one-time expenses are also required in BestConfig. While one-time expenses are inevitable in the optimization process, AB-MOEA/D is still attractive compared to empirical tuning. One reason is that in the face of a huge search space, empirical tuning may fail because the optimal configuration is application-oriented. Another reason is that in many user scenarios, long-running applications can be represented by a large number of periodic jobs, thus makes the utility gains of AB-MOEA/D further expanded.

6.4. Detailed analysis

In this section, we will study the experimental results in depth. We performed a detailed analysis on *Bayes*. The preceding benchmark is chosen mainly because AB-MOEA/D achieved the best optimization. Fig. 10 shows the execution time of each stage of *Bayes*. We can see that both AB-MOEA/D and BestConfig can achieve significant performance improvements compared to the empirical tuning, while AB-MOEA/D has a higher gain. As we mentioned in Section 5.1, *Bayes* is iterative-sensitive and memory-sensitive. Take stage 10 as an example, which is iteratively counting data. When the input data set sizes are D1, D3, D5, the data sizes to be processed by stage 10 are 90G, 120G, 150G, respectively. At this time, the memory becomes a bottleneck to affect the execution time of *Bayes*. AB-MOEA/D can choose the optimal configuration to allocate appropriate execution memory for



(a) WC-D1



(b) KM-D1

Fig. 9. Evolution of the objective values for the *WC-D1* and *KM-D1*.

Bayes to ensure that each stage can be executed with maximum efficiency. This can not only minimize the execution time but also avoid the waste of resources. Moreover, the optimization result of AB-MOEA/D is also consistent with the tuning guide given by the official (Team, 2018), the difference is that AB-MOEA/D can recommend quantitative suggestions instead of qualitative ones. Although BestConfig can also achieve performance improvements, its gain is not satisfactory when considering the execution time and cost simultaneously, which we mentioned in Section 6.2.

To investigate whether AB-MOEA/D can arrive at an optimal trade-off between execution time and cost, we select a new solution with the smallest execution time in Pareto front as a comparison, which is represented by AB-MOEA/D-T. Fig. 11 shows the execution time and cost of AB-MOEA/D and AB-MOEA/D-T. We can see that for all data sets of *Bayes*, the execution time of AB-MOEA/D compared to AB-MOEA/D-T only increased by an average of 5.26% while the maximum of 6.3%. The execution cost is reduced by an average of 17.1% while the maximum of 30.1%. This means the performance improvement may not meet expectations even if we paid a higher price, and similar results can be observed on other benchmarks. In the performance-first scenario, users can certainly choose AB-MOEA/D-T. But in scenarios where cost-effectiveness needs to be considered, it is also feasible to sacrifice a small amount of execution time to save a lot of costs. Benefits from a set of valid solutions provided by AB-MOEA/D, the users can flexibly choose configurations according to their requirements to balance between execution time and cost optimally.

7. Discussion

Choice of search framework: Compared with other Pareto dominance-based multi-objective optimization algorithms, such as NSGA-II (Deb et al., 2000) and SPEA-II (Kim et al., 2004), the decomposition-based multi-objective optimization algorithm (MOEA/D) not only has advantages in computational complexity but also performs better in maintaining population diversity. This is undoubtedly attractive for the optimal configuration selection problem because diverse solutions can provide us with more choices. There are some more complex multi-objective optimization algorithms that we did not consider, such as NSGA-III (Deb and Jain, 2014a). We think it is more suitable for solving many-objective problems (MaOP) (Deb and Jain, 2014b), while our research problem only involves two objectives.

Performance model: Although the performance model can help us avoid the high computational cost of evaluating candidate configurations, it is still necessary to maintain a certain scale of samples as the training set. If we can reduce the overhead of building a performance model, the utility of AB-MOEA/D will be further improved. Recent studies (Sarkar et al., 2015) have shown that heuristic sampling methods can be used to collect as few training samples as possible while meeting the accuracy requirements. We consider integrating similar sampling methods into AB-MOEA/D in future work.

Uncertainties in clouds: So far, we have assumed that the relationship between Spark application performance and configuration is fixed. But when the Spark application runs in the public cloud, it may break this assumption due to uncertainties in the resource-sharing environment. For example, when a user's operation causes overload or failure, it may affect the performance of the Spark application. Considering that this situation rarely occurs in large-scale public cloud environments, we believe that the AB-MOEA/D should remain relevant across relatively long periods.

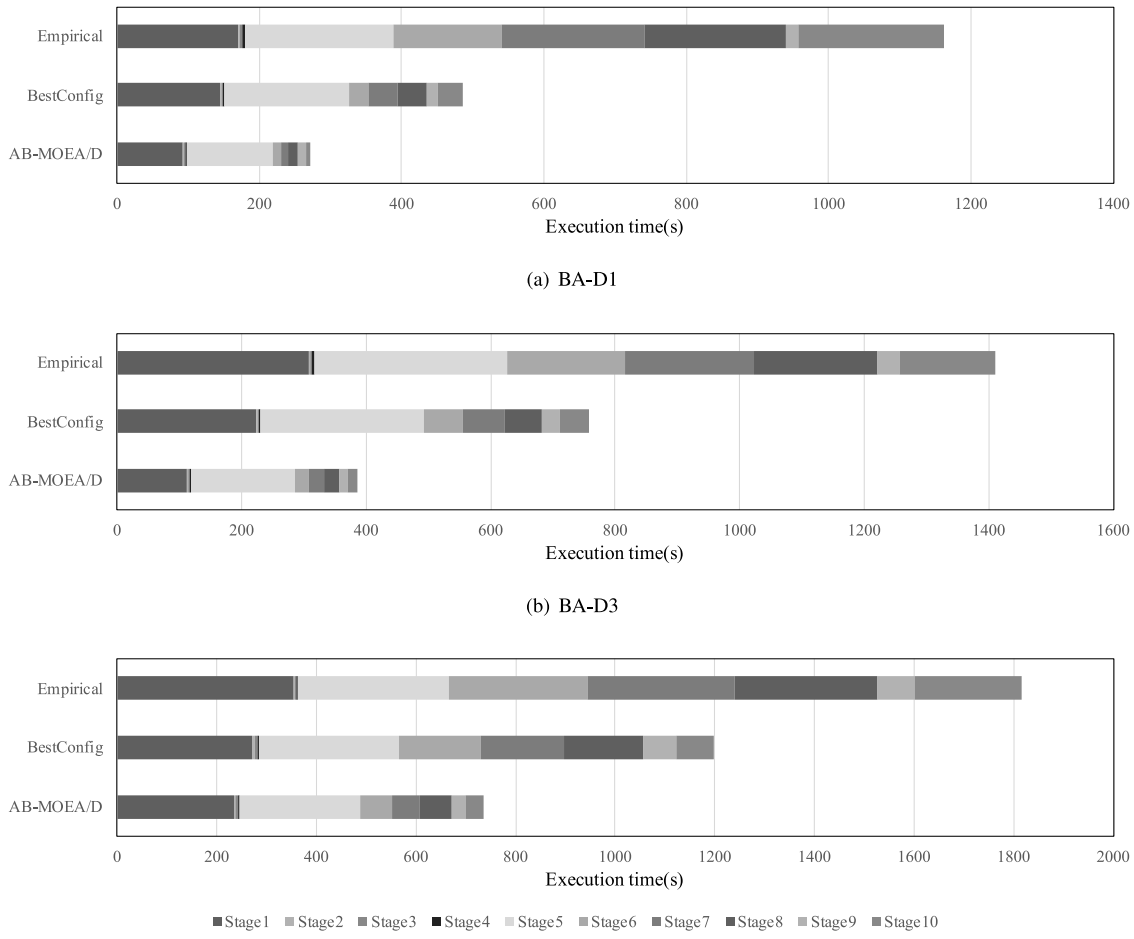


Fig. 10. Execution time of each stage for *Bayes* with empirical tuning, BestConfig, and AB-MOEA/D.

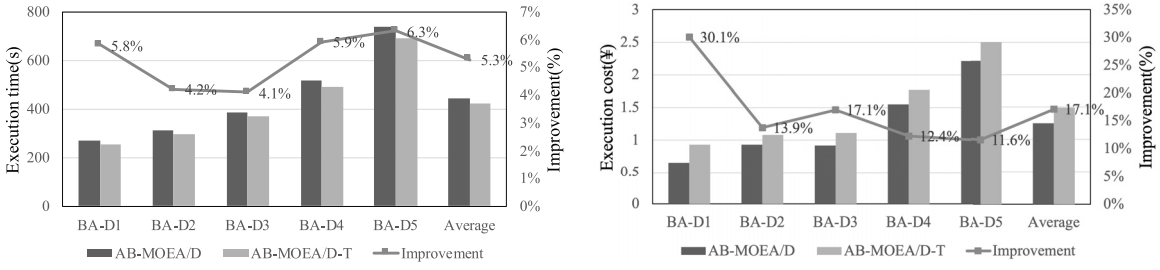


Fig. 11. Performance comparison for *Bayes* between AB-MOEA/D and AB-MOEA/D-T.

8. Related work

Spark configuration optimization can be modeled as a single-objective or a multi-objective optimal parameter selection problem. We start by describing approaches that used single-objective optimization.

Single-objective optimization: Gounaris and Torres (2018) investigated the impact of the most important tunable Spark parameters on the application performance. They proposed a more systematic graph algorithm on the basis of trial-and-error methodology (Petridis et al., 2016) to generate candidate configurations, and then tested them to select the optimal with the best performance. Zhibi Yu and Qian (2018) first suggested that the input data set has a significant impact on Spark performance. In their works, they proposed to combine the performance prediction model with genetic algorithm (GA) to search configuration space. This idea is similar to Bei et al. (2018), the difference lies in the modeling process. The models used in the former takes the input data set size into account, whereas the latter does not. Wang et al. (2017) discussed which searching algorithm is better in the case of Spark configuration parameter tuning, including the genetic algorithm (GA) and the particle swarm optimization (PSO). Gu et al. (2018) proposed a neural network based search algorithm for configuration tuning, where the performance model based on random forest is applied to improve search efficiency. Nguyen et al. (2018) first used Latin hypercube design strategy to collect training data and then determined the most effective performance model through experiments. Finally, they used Recursive Random Search algorithm to tune the configuration settings for Spark applications. Different from the above-mentioned solutions, Perez et al. (2018) proposed a new approach that tunes the configuration according to the resource bottleneck,

which was stable for clusters varying. Although these approaches have been well applied in optimizing Spark performance, they all only consider one optimization objective, that is, the execution time of the Spark applications. Our approach adds execution cost objective which is equally important in actual situations. In other words, we treat the Spark configuration tuning problem as multi-objective optimization and provide a set of Pareto-optimal solutions, which is convenient for the stakeholders to choose a more comprehensive optimization plan.

Another part of the single-objective optimization work focuses on the MapReduce-based computing framework. Wang et al. (2009), Hua et al. (2018), Liu et al. (2015), Khan et al. (2017), Bei et al. (2016) and Kumar et al. (2017) proposed MapReduce optimization approaches based on configuration tuning, respectively. Wasi-ur Rahman et al. (2018) and Wasi-ur Rahman et al. (2014) gave new ideas on how to deal with RDMA-enhanced MapReduce. Han et al. (2018) introduced a runtime configurator for cluster schedulers that automatically adapts to the changing workload and resource status. A good idea is to directly apply the approaches that have been successful in the MapReduce-based computing framework to Spark. However, Spark is different from MapReduce-based computing framework in the bottom implementation, which makes simple transfer difficult to achieve the desired effect. Therefore, we need to design a new optimization approach for Spark.

Multi-objective optimization: At present, there are relatively few works on multi-objective optimization of Spark, but some multi-objective optimization work in MapReduce or cloud has inspired us to think. Zhu et al. (2017) proposed an automatic configuration tuning system BestConfig for general systems, including Spark. To tune system configurations within a resource limit, they introduced the Divide & Diverge Sampling method and the Recursive Bound & Search algorithm. However, as we mentioned in Section 6.2, BestConfig is mainly designed for the single optimization objective. Therefore, when faced with the multi-objective optimization problem, the difficulty of weighing between objectives limits its optimization effect. Herodotou et al. (2011) provided an automatic query system that determines the cluster size required for MapReduce jobs. Alipourfard et al. (2017) proposed a cloud configuration automatic adjustment system, which can identify the best configuration for big data analytics with low search cost based on Bayesian Optimization. Guerrero et al. (2018) leveraged the non-dominated sorting genetic algorithm-II to solve the virtual machine management problem with three minimization objectives. These studies are more focused on cloud configuration and did not analyze the impact of configuration parameters included in Spark on performance, while our approach comprehensively considers two parts of parameters to achieve better performance improvement.

There are also some studies closely related to multi-objective optimization that has also given us a lot of help. Gong et al. (2013) first tried to use interaction theory to solve MOP with interval parameters and proposed an interactive evolutionary algorithm based on the theory of preference polyhedrons, which got feedback information from decision makers for guiding the optimization process. They also proposed a hybrid multi-objective discrete artificial bee colony algorithm for the scheduling problem of the bi-objective batch flow workshop (Gong et al., 2018), in which the newly developed genetic operator and local search were used respectively for enhancing the performance of the algorithm in exploitation and exploration. Zhang et al. (2018) proposed a decomposition-based archiving method to improve the traditional multi-objective optimization algorithm, which can effectively reduce the computational cost. Gong et al. (2021) and Sun et al. (2020) combined PSO with the bagging-ensemble surrogate model based on multiple radial basis function networks (RBFN) to generate path coverage test data for message passing interface (MPI) programs, which was also an expensive optimization problem. All these works provide us with valuable experience for solving the Spark configuration optimization problem.

Besides, many other studies optimize Spark performance from resource utilization (Ruan et al., 2015; Gounaris et al., 2017; Sidhanta et al., 2016; Islam et al., 2020; Rehman et al., 2019). Our work does not optimize performance from these aspects, but instead focuses on configuration and can be used as a complement to these approaches.

9. Conclusion

In this paper, we have investigated the issue of choosing the configuration to optimize Spark applications on public clouds. The choice of configuration depends on the users' requirements which typically involves several conflicting objectives, such as minimizing the execution time and minimizing the execution cost. How to automatically pick the right configuration to trade-off these objectives is a challenge.

To address this challenge, we propose a new optimal configuration search algorithm named AB-MOEA/D by combining multi-objective optimization algorithm with performance prediction model. When facing the configuration space composed of application-level and cloud-level parameters, AB-MOEA/D can provide a set of Pareto-optimal solutions for users to select. Besides, we also present the configuration automatic tuning system with AB-MOEA/D as the optimization engine, which is a highly flexible and extensible architecture. We evaluate our approach using six typical Spark benchmarks, each with five different input data sets. The results show that AB-MOEA/D significantly outperforms the previous work in terms of execution time and cost, with average improvements of approximately 35 and 40 percent.

CRediT authorship contribution statement

Guoli Cheng: Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing. **Shi Ying:** Supervision, Project administration, Funding acquisition. **Bingming Wang:** Validation, Visualization, Data curation.

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 work is supported in part by the grants of National Natural Science Foundation of China (62072342, 61672392).

References

Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M., 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017, pp. 469–482.

Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M., 2015. Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, pp. 1383–1394.

Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W.L., Shalf, J., Williams, S., 2006. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report Uc Berkeley UCB/EECS-2006-183.

Assefi, M., Behravesh, E., Liu, G., Tafti, A.P., 2017. Big data machine learning using apache spark mllib. In: 2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017, pp. 3492–3498.

Bei, Z., Yu, Z., Luo, N., Jiang, C., Xu, C., Feng, S., 2018. Configuring in-memory cluster computing using random forest. Future Gener. Comput. Syst. 79, 1–15.

Bei, Z., Yu, Z., Zhang, H., Xiong, W., Xu, C., Eeckhout, L., Feng, S., 2016. RFHOC: A random-forest approach to auto-tuning hadoop's configuration. IEEE Trans. Parallel Distrib. Syst. 27, 1470–1483.

Bonner, S., Brennan, J., Theodoropoulos, G., Kureshi, I., McGough, A.S., 2016. GFP-X: A parallel approach to massive graph comparison using spark. In: 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016, pp. 3298–3307.

Deb, K., Agrawal, S., Pratap, A., Meyarivan, T., 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In: Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Guervós, J.J.M., Schwefel, H. (Eds.), Parallel Problem Solving from Nature - PPSN VI, 6th International Conference, Paris, France, September 18-20, 2000, Proceedings. In: Lecture Notes in Computer Science, vol. 1917, Springer, pp. 849–858.

Deb, K., Jain, H., 2014a. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. IEEE Trans. Evol. Comput. 18, 577–601.

Deb, K., Jain, H., 2014b. An evolutionary many-objective optimization algorithm using reference-point based non-dominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. IEEE Trans. Evol. Comput. 18, 602–622.

Ferguson, A.D., Bodík, P., Kandula, S., Boutin, E., Fonseca, R., 2012. Jockey: guaranteed job latency in data parallel clusters. In: European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012. ACM, pp. 99–112.

Gong, D., Han, Y., Sun, J., 2018. A novel hybrid multi-objective artificial bee colony algorithm for blocking lot-streaming flow shop scheduling problems. Knowl. Based Syst. 148, 115–130.

Gong, D., Sun, J., Ji, X., 2013. Evolutionary algorithms with preference polyhedron for interval multi-objective optimization problems. Inform. Sci. 233, 141–161.

Gong, D., Sun, B., Yao, X., Tian, T., 2021. Test data generation for path coverage of MPI programs using SAEO. ACM Trans. Softw. Eng. Methodol. 30, 17:1–17:37. http://dx.doi.org/10.1145/3423132.

Gounaris, A., Kougka, G., Tous, R., Montes, C.T., Torres, J., 2017. Dynamic configuration of partitioning in spark applications. IEEE Trans. Parallel Distrib. Syst. 28, 1891–1904.

Gounaris, A., Torres, J., 2018. A methodology for spark parameter tuning. Big Data Res. 41186, 22–32.

Gu, J., Li, Y., Tang, H., Wu, Z., 2018. Auto-tuning spark configurations based on neural network. In: 2018 IEEE International Conference on Communications, ICC 2018, Kansas City, MO, USA, May 20-24, 2018, pp. 1–6.

Guerrero, C., Lera, I., Bermejo, B., Juiz, C., 2018. Multi-objective optimization for virtual machine allocation and replica placement in virtualized hadoop. IEEE Trans. Parallel Distrib. Syst. 29, 2568–2581.

Hai, A.A., Forouraghi, B., 2018. On scalability of distributed machine learning with big data on apache spark. In: Big Data - BigData 2018 - 7th International Congress, Held As Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings, Vol. 10968. pp. 209–219.

Han, R., Zong, Z., Chen, L.Y., Wang, S., Zhan, J., 2018. Adaptiveconfig: Run-time configuration of cluster schedulers for cloud short-running jobs. In: 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018, pp. 1519–1526.

Herodotou, H., Dong, F., Babu, S., 2011. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In: ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011. ACM, p. 18.

Hua, X., Huang, M.C., Liu, P., 2018. Hadoop configuration tuning with ensemble modeling and metaheuristic optimization. IEEE Access 6, 44161–44174.

Huang, S., Huang, J., Dai, J., Xie, T., Huang, B., 2010. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010. IEEE Computer Society, Long Beach, California, USA, pp. 41–51.

Islam, M.T., Srirama, S.N., Karunasekera, S., Buyya, R., 2020. Cost-efficient dynamic scheduling of big data applications in apache spark on cloud. J. Syst. Softw. 162.

Khan, M., Huang, Z., Li, M., Taylor, G.A., Khan, M., 2017. Optimizing hadoop parameter settings with gene expression programming guided PSO. Concurr. Comput. Pract. Exp. 29.

Kim, M., Hiroyasu, T., Miki, M., Watanabe, S., 2004. SPEA2+: improving the performance of the strength pareto evolutionary algorithm 2. In: Parallel Problem Solving from Nature - PPSN VIII, 8th International Conference, Birmingham, UK, September 18-22, 2004, Proceedings. In: Lecture Notes in Computer Science, vol. 3242, Springer, pp. 742–751.

Kumar, S., Padakandla, S., Lakshminarayanan, C., Parihar, P., Gopinath, K., Bhatnagar, S., 2017. Scalable performance tuning of hadoop mapreduce: A noisy gradient approach. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017, pp. 375–382.

Li, T., Shi, S., Luo, J., Wang, H., 2018. A method to identify spark important parameters based on machine learning. In: Data Science - 4th International Conference of Pioneering Computer Scientists, Engineers and Educators, ICPCSEE 2018, Zhengzhou, China, September 21-23, 2018, Proceedings, Part I. In: Communications in Computer and Information Science, vol. 901, Springer, pp. 525–538.

Liu, C., Zeng, D., Yao, H., Hu, C., Yan, X., Fan, Y., 2015. MR-COF: A genetic mapreduce configuration optimization framework. In: Algorithms and Architectures for Parallel Processing - 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015. Proceedings, Part IV, Vol. 9531. pp. 344–357.

Luo, X., Fu, X., 2019. Configuration optimization method of hadoop system performance based on genetic simulated annealing algorithm. Cluster Comput. 22, 8965–8973.

Marler, R.T., Arora, J.S., 2010. The weighted sum method for multi-objective optimization: new insights. Struct. Multidiplinary Optim. 41, 853–862.

Mavridis, I., Karatza, H.D., 2017. Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark. J. Syst. Softw. 125, 133–151.

Nguyen, N., Khan, M.M.H., Albayram, Y., Wang, K., 2017. Understanding the influence of configuration settings: An execution model-driven framework for apache spark platform. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, June 25-30, 2017. IEEE Computer Society, pp. 802–807.

Nguyen, N., Khan, M.M.H., Wang, K., 2018. Towards automatic tuning of apache spark configuration. In: 11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018, pp. 417–425.

Perez, T.B.G., Chen, W., Ji, R., Liu, L., Zhou, X., 2018. PETS: bottleneck-aware spark tuning with parameter ensembles, in: 27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018, pp. 1–9.

Petridis, P., Gounaris, A., Torres, J., 2016. Spark parameter tuning via trial-and-error. In: Advances in Big Data - Proceedings of the 2nd INNS Conference on Big Data, Vol. 529, October 23-25, 2016, Thessaloniki, Greece, pp. 226–237.

Wasi-ur Rahman, M., Islam, N.S., Lu, X., Shankar, D., Panda, D.K., 2018. Mr-advisor: A comprehensive tuning, profiling, and prediction tool for mapreduce execution frameworks on HPC clusters. J. Parallel Distrib. Comput. 120, 237–250.

Wasi-ur Rahman, M., Lu, X., Islam, N.S., Panda, D.K., 2014. Performance modeling for rdma-enhanced hadoop mapreduce. In: 43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014, pp. 50–59.

Rehman, A., Hussain, S.S., u. Rehman, Z., Zia, S., Shamshirband, S., 2019. Multi-objective approach of energy efficient workflow scheduling in cloud environments. Concurr. Comput. Pract. Exp. 31.

Ruan, J., Zheng, Q., Dong, B., 2015. Optimal resource provisioning approach based on cost modeling for spark applications in public clouds. In: I. Beschastnikh, W. Joosen (Eds.), Proceedings of the Doctoral Symposium of the 16th International Middleware Conference, Middleware Doct Symposium 2015, Vancouver, BC, Canada, December 7-11, 2015, pp. 6:1–6:4.

Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z., 2016. Big data analytics on apache spark. Int. J. Data Sci. Anal. 1, 145–164.

Sarkar, A., Guo, J., Siegmund, N., Apel, S., Czarnecki, K., 2015. Cost-efficient sampling for performance prediction of configurable systems (t). In: Cohen, M.B., Grunske, L., Whalen, M. (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. IEEE Computer Society, pp. 342–352.

Sidhanta, S., Golab, W.M., Mukhopadhyay, S., 2016. Optex: A deadline-aware cost optimization model for spark. In: IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016, pp. 193–202.

Sun, B., Gong, D., Yao, X., Tian, T., 2020. Integrating an ensemble surrogate model's estimation into test data generation. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2020.3019406.

Team, A.S., 2018. Spark configuration. http://spark.apache.org/docs/latest/configuration.html.

Wang, G., Butt, A.R., Pandey, P., Gupta, K., 2009. A simulation approach to evaluating design decisions in mapreduce setups. In: 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2009, September 21-23, 2009. South Kensington Campus, Imperial College, London, UK, pp. 1–11.

Wang, W., Li, K., Tao, X., Gu, F., 2020. An improved MOEA/D algorithm with an adaptive evolutionary strategy. Inform. Sci. 539, 1–15.

Wang, Y., Liu, Q., Yu, J., Yu, Z., 2017. An experimental comparison between genetic algorithm and particle swarm optimization in spark performance tuning. In: Proceedings of the First Workshop on Emerging Technologies for Software-Defined and Reconfigurable Hardware-Accelerated Cloud Datacenters, ETCD@ASPLOS 2017, Xi'an, China, April 8, 2017, pp. 1:1–1:6.

Wang, G., Xu, J., He, B., 2016. A novel method for tuning configuration parameters of spark based on machine learning. In: 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016, Sydney, Australia, December 12-14, 2016, pp. 586–593.

Zhang, Y., Gong, D., Sun, J., Qu, B., 2018. A decomposition-based archiving approach for multi-objective evolutionary optimization. Inform. Sci. 430, 397–413.

Zhang, Q., Li, H., 2007. MOEA/D: a multiobjective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. 11, 712–731.

Zhibi Yu, Z.B., Qian, X., 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018, pp. 564–577.

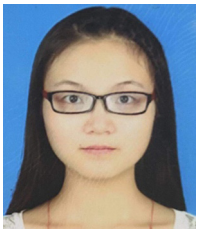
Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., Yang, Y., 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In: Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017, pp. 338–350.

Zhu, Y., Zhan, J., Weng, C., Nambiar, R., Zhang, J., Chen, X., Wang, L., 2014. Bigop: Generating comprehensive big data workloads as a benchmarking framework. In: Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part II. In: Lecture Notes in Computer Science, vol. 8422, Springer, pp. 483–492.

Zio, E., Bazzo, R., 2012. A comparison of methods for selecting preferred solutions in multiobjective decision making. In: Computational Intelligence Systems in Industrial Engineering. Springer, pp. 23–43.

Guoli Cheng is working toward the Ph.D. degree in the School of Computer Science, Wuhan University, China. His research interests include performance optimization of big data system and machine learning.

Shi Ying is now a professor with Wuhan University, where he is vice dean in the Computer School and he is also the deputy director of the State Key Laboratory of Software Engineering. He has authored or co-authored more than 100 referred papers in the area of software engineering. His main research interests include service-oriented software engineering, Semantic Web service, and trustworthy software.



Bingming Wang is currently a Ph.D. candidate in College of Computer Science at Wuhan University, China, under the supervision of Prof Shi Ying. She received her master degree from Central South University for nationalities in 2016. Her research interests include data mining, machine learning, and log analysis.