Adaptive Code Learning for Spark Configuration Tuning

Chen Lin*, Junqing Zhuang*, Jiadong Feng*, Hui Li*, Xuanhe Zhou†, Guoliang Li †

*School of Informatics, Xiamen University, China. chenlin@xmu.edu.cn

†Department of Computer Science, Tsinghua University, China. liguoliang@tsinghua.edu.cn

Abstract—Configuration tuning is vital to optimize the performance of big data analysis platforms like Spark. Existing methods (e.g. auto-tuning relational databases) are not effective for tuning Spark, because the unique characteristics of Spark pose new challenges to configuration tuning. (C1) The Spark applications own various code structures and semantics, and the code features significantly affect Spark performance and configuration selection; (C2) Spark applications are extremely time-consuming on big data. It is infeasible for approaches such as Bayesian Optimization and Reinforcement Learning to collect sufficient training instances or repeatedly execute the applications; (C3) Spark supports various analytical applications and the tuning system needs to adapt to different applications.

To address these challenges, we propose a LIghtweighT knob rEcommender system (LITE) for auto-tuning Spark configurations on various analytical applications and large-scale datasets. We first propose a code learning framework that can utilize code features to learn complex correlations between application performance and knob values (addressing C1). We then propose a lightweight auto-tuning method that migrates the knowledge learned from small-scale datasets to large-scale datasets (addressing C2). Next, to generalize to different Spark applications, we propose an adaptive model update approach to fine-tune the model via adversarial learning with newly collected feedback (addressing C3). Extensive experiments showed that LITE achieves much better performance compared with state-of-the-art auto-tuning methods.

Index Terms—spark, automatic configuration tuning, code understanding

I. INTRODUCTION

With the incredible volume of data generated each day, **B**ig **D**ata **A**nalytics (BDA) platforms have become indispensable to adequately manage, process and analyze large-scale data. However, it is remarkably difficult to fulfill various configuration requirements for different BDA applications. Figure 1 illustrates the execution time of applications with respect to configuration knobs in Spark, one of the most widely-used BDA platforms [19]. We can see that, the optimal setting of knob "executor.cores" must be tailored for each application (e.g., 6 for TriangleCount and 4 for PageRank). Furthermore, it becomes much harder when multiple knobs are involved. For example, the combination "executor.cores=4, executor.memory=3" produces significantly less execution time than other configurations. Traditionally, experts are hired to carefully set the knobs by repeated trials. This manual tuning method is incomplete (i.e., empirically testing a small percentage of knobs), inefficient (i.e., several days are spent) and sub-optimal (i.e., settings are not optimal) [17].
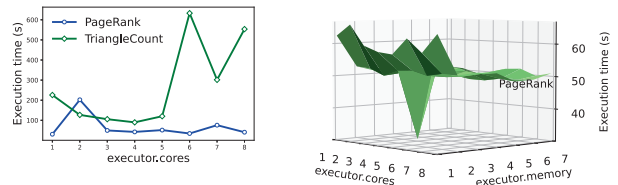

Fig. 1: Spark applications PageRank and TriangleCount execution time on 160MB input data, with different configurations.

To reduce the labor and time cost, existing studies for Spark tuning either select configurations based on the predicted performance by supervised learning models [4], [9], [13], [29], or repeatedly sample and execute promising configurations based on feedback from previous trials [3], [27], [34]. On one hand, to the best of our knowledge, existing Spark tuning, which is trained on small datasets and ignores the code semantics and structures, cannot scale to diverse and large jobs. On the other hand, although there are some database tuning approaches in production and at scale [7], [8], [36], like Reinforcement Learning [17], [31] and Bayesian Optimization [15], [23], [30], those methods are not effective for tuning Spark, because the following unique characteristics of Spark pose new challenges to Spark configuration tuning.

(C1) Complex Spark code semantics. Unlike DB queries and SQL style scripts, Spark applications, especially the widely used ML and graph algorithms, have richer semantic, with a large vocabulary of code tokens. The code structures are more complex, with long-range dependencies upon layers of iterative computations, than SQL queries where the most complex SQL structures like UDFs just include some simple logical judgments. The various code structures and semantics significantly affect Spark performance and configuration selection. Hence, it is important and challenging to capture the complex correlations between code semantics and configuration knobs.

(C2) Expensive training cost. Spark applications, including iterative ML algorithms performed on big data, usually require longer execution time. Compared with DB queries, ML algorithms take hours to finish one job, and thus it is rather expensive to perform repeatedly trials. This limits the usage of online training methods like RL, which explores optimal configurations by repeatedly executing the application, due to the high tuning overhead (i.e., time to decide the best configurations). It is also a bottleneck to collect training instances of large jobs. Hence, there is a need to efficiently migrate the tuning model learned on small datasets to large
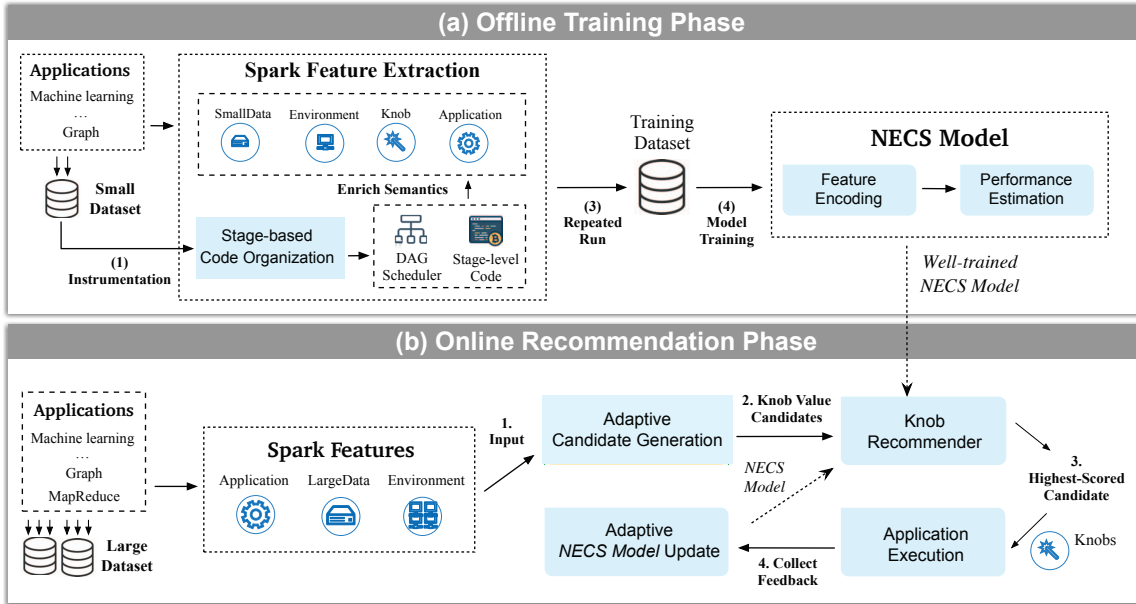
Fig. 2: LITE framework overview

datasets.

(C3) Various Spark applications. Different from databases, Spark powers various applications with libraries like SQL, machine learning, and other data analytics. Different applications have various impacts on the tuning performance, raising the difficulty to represent the application characteristics for knob tuning. Thus the tuning model needs to adapt to different applications.

To address the above challenges, as depicted in Figure 2, we propose the LITE system (**LI**ghtweigh**T** knob r**E**commender) that develops novel machine learning approaches for auto-tuning Spark configurations to optimize application execution. LITE first extracts stage-level code and Directed Acyclic Graph (DAG) Scheduler as training features and proposes a code learning framework NECS model (**N**eural **E**stimator via **C**ode and **S**cheduler representation) that can utilize code features to learn complex correlations between application performance and knob values (addressing C1). We then propose a lightweight auto-tuning method that migrates the knowledge learned from small-scale datasets to large-scale datasets (addressing C2). Next, in order to generalize to different Spark applications, we propose an adaptive model update approach to fine-tune the model via adversarial learning with newly collected feedback (addressing C3), which can provide fast and near-optimal *online* recommendation.

Contributions: We make the following contributions.

1) The lightweight auto-tuning system (i.e. LITE) that migrates the knowledge learned from small-scale datasets to large datasets and greatly improves BDA tuning performance with small training expense (see Section II).
2) The code encoding modules in NECS that capture complex correlations among knob values and BDA program codes with rich semantics and structures (see Section III).

3) The `Adaptive Model Update` method transfers the knowledge learned from the training instances on small data to big data applications, and the `Adaptive Candidate Generation` method reduces tuning overhead and enhances tuning performance (see Section IV).
4) Extensive experimental study to verify that LITE significantly outperformed existing approaches in both efficiency and effectiveness (see Section V).

II. LITE SYSTEM OVERVIEW

Architecture. Figure 2 shows the architecture of LITE (lightweight knob recommender), which is composed of an offline training phase and an online recommendation phase. The offline training phase trains a NECS model (**N**eural **E**stimator via **C**ode and **S**cheduler representation), which is used to estimate the performance of a Spark application for a given knob configuration. Then for online knob recommendation, given a Spark application, it uses the NECS model to recommend appropriate knob values.

Offline Training Phase. The NECS model first analyzes the Spark codes and generates stage-based code organization in order to (1) better capture the code semantics and (2) generate more code combinations via permuting different code stages. Then it collects the code features from the stage-based code and DAG scheduler features from the Spark scheduling. This step aims to enrich the code semantics. Next it generates a set of training data, where each training sample includes the Spark application, Spark code, data, Spark environment, the knob values, and running performance. Based on the training data, the NECS model extracts the features and trains an estimation model that can predict the running performance of a given Spark application. Note that the training data will be run on small datasets and the NECS model can predict the performance on large datasets by migrating the knowledge

learned from small-scale datasets to large-scale datasets. The details of the `NECS` model is discussed in Section III.

Online Recommendation Phase. The online model first identifies a set of candidate knob configurations via `Adaptive Candidate Generation`. This step aims to identify knob regions which are likely to contain good configurations for the given application, and thus the tuning overhead is drastically decreased by filtering out a large number of knob candidates. It then estimates the performance of each candidate configuration via the `NECS` model, and `Knob Recommender` recommends the knob values with the best estimated performance. Next, the user will execute the Spark application with the recommended knob values and collect the feedback. Based on the feedback, we can fine-tune the `NECS` model via `Adaptive Model Update`. Fine-tuning learns indistinguishable representations for training and testing instances via adversarial learning, so that the predictions on large input data will be more accurate. The fine-tuning can be conducted periodically, i.e., updating `LITE` when we collect a predefined batch of testing workloads. The details of online recommendation is discussed in Section IV.

III. NECS: NEURAL ESTIMATOR VIA CODE AND SCHEDULER REPRESENTATION

A. NECS Overview

By repeatedly sampling knob values and running applications, `LITE` collects a set of application instances. Each application instance is a specific application implemented with a set of selected knob values on a certain input data and a particular computing environment. Directly using these application samples to train a conventional machine learning model (e.g., linear regression) is not practical. Because machine learning models are largely reliant on sufficient amount and diversity of training data, which we can hardly reach with limited time and resource. As shown in Figure 3, instead of naively using application samples, `NECS` adopts `Stage-based Code Organization` to build the set of training instances. Each training instance is constructed as a set of input features, paired with a performance metric (i.e., execution time). Then, `NECS` encodes the input features to hidden representation vectors. Finally, `NECS` estimates the performance, given the feature encodings.

B. Stage-based Code Organization

`Stage-based Code Organization` is motivated to enhance the amount and diversity of training data. We illustrate our motivation in Figure 4, with *Terasort* in spark-bench [1]. (1) Firstly, running the application once generates only one application instance, and it takes several minutes. (2) Secondly, its main-body codes are extremely brief, i.e., only three lines are functional (line $3 \sim 5$). Specifically, only line 4 distinguishes the function of this application (i.e., sorting). (3) Thirdly, the tokens are very sparsely distributed. For example, in line 4, the important tokens 'TeraSortPartitioner" and "sortByKey", which are with strong distinguishing power, almost never appear in other applications. Thus, the machine

learning models can not learn the correlations among rare tokens and generalize to other applications. Our solution is to segment each application into stages, expand the stage-level codes and process the stage-level codes to obtain stage features. `Stage-based Code Organization` implements the following three steps.

Step1: Instrumentation. We use instrumentation techniques to segment an application instance to several stage-level instances. Instrumentation is an automatic step. For each application, we prepare a Java agent jar file, which modifies bytecode of the Spark core packages, currently including org/apache/spark/rdd, api, mllib and graphix. It monitors when classes in these packages are loaded at each stage, and stores the codes to a hashing table. After the application is finished, we parse the application logs to extract *stage-level codes*. As shown in Figure 5, after instrumentation, codes of stage 3, which is associated with line 4 in Figure 4, have been greatly expanded. We can see that, the stage-level codes of stage 3 contain a larger set of tokens, where the important tokens "map" are more densely distributed among different instances. Thus, instrumentation captures the operations in each stage and obtains more informative features.

Step2: Code features extraction. After we obtain the stage-level codes, we use a token embedding matrix to represent the stage-level codes. Figure 3 shows that, the token embedding matrix is obtained by concatenating token embeddings, i.e., $\mathbf{C}_i \in \mathbb{R}^{D \times N}$, where D is the embedding size for each code token, $N = 1000$ is the maximal number of tokens in each stage. We use padding (i.e., adding zeros) for short codes.

Step3: Scheduler features extraction. We also extract *stage-level scheduler DAGs* by parsing the event log files. The stage-level DAG scheduler consists of a set of labeled nodes and directed edges, where nodes represent the Resilient Distributed Datasets (RDDs) or DataFrames and the edges represent an operation to be applied. As shown in Figure 3, each node in the DAG is labeled with a word that reveals the atomic operation (e.g., "MapPartition", "ZipPartition") on RDDs. Thus, to fully capture the semantics of the DAG and strengthen prediction power, we use a *node embedding matrix* to represent the nodes, and an *adjacency matrix* to represent the edges, i.e., $\mathcal{G}_i = \{\mathbf{V}_i, \mathbf{A}_i\}$. The node embedding matrix is denoted by $\mathbf{V}_i \in \mathbb{R}^{|V| \times (S+1)}$, where $|V|$ is the number of nodes in the DAG, and S is the embedding size for each node. We use one-hot encoding for each node, hence, S is the number of atomic operations in the training set. To increase generalizability, we add an out-of-vocabulary token (denoted as ovv) to handle unseen atomic operations in the test application.

C. Inputs of NECS

After `Stage-based Code Organization`, we construct the training instances. Formally, the $i-$th training instance $x_i \in \mathcal{DS}$ in training set $\mathcal{DS}$ is represented as a six-tuple, i.e., $x_i =< \mathbf{o}_i, \mathbf{C}_i, \mathcal{G}_i, \mathbf{d}_i, \mathbf{e}_i, y_i >$, where $\mathbf{o}_i$ is an array of knob values, $\mathbf{C}_i, \mathcal{G}_i$ are the code features and scheduler features extracted in Section III-B, $\mathbf{d}_i$ represents the data feature, $\mathbf{e}_i$ denotes the computing environment that this
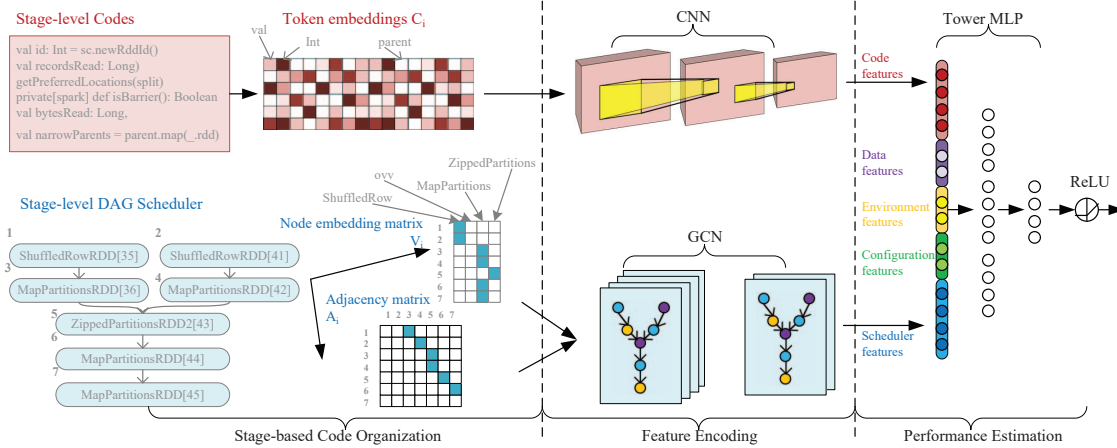
Fig. 3: NECS framework

```
1  object terasortApp{
2    def main(args: Array[String]) {
3      val dataset = sc.newAPIHadoopFile[Array[Byte], Array[Byte],
       ↪ TeraInputFormat](inputFile)
4      val sorted = dataset.partitionBy(new
       ↪ TeraSortPartitioner(dataset.partitions.size)).sortByKey()
5      sorted.saveAsNewAPIHadoopFile[TeraOutputFormat](outputFile) }}
```

Fig. 4: Program codes of application Terasort. Important tokens never appear in other applications.

```
1  @volatile @transient private var partitions_ : Array[Partition]
   ↪ = _
2  val id: Int = sc.newRddId()
3  val recordsRead: Long)
4  getPreferredLocations(split)
5  private[spark] def isBarrier(): Boolean = isBarrier_
6  val bytesRead: Long,
7  val narrowParents = narrowDependencies.map(_.rdd)
8  checkpointRDD.map(_.getPreferredLocations(split)).getOrElse {
9  isFromBarrier || dependencies.exists(_.rdd.isBarrier())
10 checkpointRDD.map(_.partitions).getOrElse {
11 val narrowDependencies =
   ↪ rdd.dependencies.filter(_.isInstanceOf[NarrowDependency[_]])
12 ("Scope" -> rddInfo.scope.map(_.toJson)) ~
13 private[spark] var checkpointData: Option[RDDCheckpointData[T]]
   ↪ = None
14 ("Scope" -> rddInfo.scope.map(_.toJson)) ~
15 checkpointRDD.map(_.partitions).getOrElse {
16 ancestors.add(parent)
17 visit(parent)
```

Fig. 5: Stage-level codes (stage 3) of application Terasort after instrumentation. Important "map" operations are highlighted.

application is implemented on, y_i is the stage-level execution time. Note that, since many stage-level training instances are extracted from the same application instance, we use $w(x_i)$ to denote the application instance where instance x_i is extracted from. For two instances extracted from the same application instance, they share the same knob, data and environment features, i.e., $\mathbf{o}_i = \mathbf{o}_j, \mathbf{d}_i = \mathbf{d}_j, \mathbf{e}_i, \mathbf{e}_j$, if $w(x_i) = w(x_j)$. Only the stage-level code features and scheduler features are different. For example, if we run *Terasort* in Figure 4 once on an input dataset, then we will have four stage-level instances with the same configuration features and data features.

Configuration features. Suppose there are D configuration knobs which affect Spark performance, then we have $\mathbf{o}_i \in \mathbb{R}^D$ (more in Section V).

Data features. We use four entries that can be obtained through the application's data generation phase before exe-

TABLE I: Entries in the data feature

Entry	Brief Description
#rows	Number of rows in the input data
#columns	Number of columns in the input data
#iteration	Number of iterations (optional)
#partitions	Number of partitions (optional)

TABLE II: Entries in the environment feature

Entry	Brief Description
#nodes	Number of nodes (computers) in the cluster
#cores	Number of cores in each node
Frequency	CPU frequency (GHZ)
Memory size	Memory size (GB)
Memory speed	Memory speeds (MT/s)
Bandwidth	Bandwidth in connecting the cluster

cuting the application to describe data characteristics. That is, each data feature vector for instance x_i is a four-dimensional vector, i.e., $\mathbf{d}_i \in \mathbb{R}^4$. As shown in Table I, not all entries are useful. For example, some machine learning libraries, e.g., k-means, define the number of iterations required to be conducted in the data generation phase. For applications which do not have "#iteration" or "#partitions", the corresponding entries in $\mathbf{d}_i$ will be set to zero.

Environment features. For clusters with more nodes and memories, higher CPU frequency and memory speeds, Spark applications are expected to complete faster. However, configuration knobs have complex dependencies with cluster environment. For example, *executor.cores* and *executor.memory* must be optimized according to the computing resource in the cluster. As shown in Table II, we use a six-dimensional vector, i.e., $\mathbf{e}_i \in \mathbb{R}^6$, to represent the environment information.

D. Encoding Code Features

We apply feature encoding upon the code features $\mathbf{C}_i$ to capture rich semantics of the codes. In the literature, sequence models such as LSTM (**L**ong **S**hort **T**erm **M**emory network) [12], [35] and Transformers [18] have been adopted to learn representations for program codes. However, we use CNN (**C**onvolutional **N**eural **N**etwork) blocks, because we observe that the stage-level codes are a set of lines, where

each line is generally very short. Sequence models which emphasize on long-term dependency might introduce noise. CNNs have shown superior performance on short texts such as reviews [33]. We empirically find that CNNs outperform sequence models, by utilizing the proximity structure in codes.

CNN applies convolutional operation with max pooling on multiple filters to obtain feature maps, resulting $\mathbf{Q} = flat\{q_1, \ldots, q_I\}$, where I denotes the number of kernels in the convolutional layer and $flat$ is the flatten operation. For space limit, we omit the calculation of CNN.

The code representation $\mathbf{h}_{code,i}$ can be obtained by a ReLu transformation with $\mathbf{W}^{CNN}$, the learnable weight matrix:

$$\mathbf{h}_{code,i} = ReLU(\mathbf{W}^{CNN}\mathbf{Q}). \tag{1}$$

E. Encoding Scheduler Features

In addition of featuring "what" operations are conducted (i.e., encoding code features), understanding "how" a series of operations are conducted is also important to estimate stage performance. However, it is non-trivial to fully utilize the semantic and structural information in the DAG scheduler features. We adopt GCN (**G**raph **C**onvolutional **N**etwork) [21], which can fully capture the semantic and structural information in DAGs by integrating the node embedding matrix and adjacency matrix in graph convolution at each iteration. GCNs have shown promising results in code summarization applications [16]. We initialize the input of GCN as embedding matrix of DAG features, i.e., $\mathbf{H}_i^0 = \mathbf{V}_i$ and apply the operations $\mathbf{H}^{l+1} = ReLU(\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}_i^l\mathbf{W}^{GCN,l})$, where the $l+1$-th layer of GCN outputs $\mathbf{H}^{l+1}$. $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, where $\mathbf{A}_i \in \mathbb{R}^{|V|\times|V|}$ is the adjacency matrix, $\mathbf{I} \in \mathbb{R}^{|V|\times|V|}$ is the identity matrix. $\hat{\mathbf{D}}$ is the degree matrix, where its diagonal elements represent the degree of each node, its rest elements are set to be zero. $\mathbf{H}_i^l$ is the output of last layer. $\mathbf{W}^{GCN,l}$ is the learnable weight matrix. Finally, the scheduler representation $\mathbf{h}_{DAG,i}$ can be obtained by a max pooling operation:

$$\mathbf{h}_{DAG,i} = \max \mathbf{H}^L. \tag{2}$$

F. Performance Estimation

Performance is affected by multiple factors with complex correlations. The estimation needs to be robust enough to adapt to new applications and large input data after well trained on small datasets. To address these challenges, we design the performance estimation module. NECS concatenates all the representations and transforms the concatenation to predicted execution time by an MLP (**M**ulti-**L**ayer **P**erceptron) module. The procedure can be formally defined in Equation 3:

$$\hat{y}_i = f_L\Big(\cdots f_2\big(f_1(concate(\mathbf{d}_i, \mathbf{e}_i, \mathbf{o}_i, \mathbf{h}_{code,i}, \mathbf{h}_{DAG,i}))\big)\cdots\Big), \tag{3}$$

where L is the number of layers in MLP, $f_l(\cdot)$ with $l = 1, 2, \cdots, L$ denotes the mapping function for the l-th hidden layer. $f_l(\mathbf{x}) = ReLU(\mathbf{W}^{MLP,l}\mathbf{x} + \mathbf{b}^{MLP,l})$, where $\mathbf{W}^{MLP,l}$ and $\mathbf{b}^{MLP,l}$ are learnable weight matrix and bias vector for layer l. The activation function for each layer is $ReLU$. We use a tower MLP structure, where the size of layers (i.e.,

dimensionality of $\mathbf{x}$) is half of the previous layers. As MLP captures high-level interactions among features, the tower structure learns more abstractive features of data by using less hidden units at higher layers.

G. Training

Suppose y_i is the actual execution time of instance x_i, NECS outputs a predicted execution time $\hat{y}_i = M_{\Theta}(x^i)$, where M_{Θ} is the set of parameters including the learnable weights in CNN, GCN and MLP modules. The offline training phase optimizes M_{Θ}:

$$\Theta^* = \arg\min_{\Theta}\sum_{x_i\in\mathcal{DS}}(y_i - \hat{y}_i)^2. \tag{4}$$

IV. ONLINE RECOMMENDATION: ADAPTIVE TUNING

In the online phase, LITE implements the following steps to tune each big data analytic application.

Step 1: Collect application features. For any application to be tuned by LITE, it is convenient to collect program codes of the application, and other information, such as the environment and the data to be processed (e.g., data features in Section III). If the application exists in the training set, i.e., the application has been executed at least once on a different input data and computing environment, we can re-use the segmented stages and obtain the stage-level codes and DAG scheduler. If the application is unseen (i.e., cold-start application), then we run the application on the smallest dataset possible and perform instrumentation (Section III-B) to quickly obtain stage-level codes and DAG sheduler.

Step 2: Generate knob value candidates. LITE selects a few knob value candidates. Due to the huge space for configurations, LITE is not able to enumerate all possible knob values. We present Adaptive Candidate Generation in Section IV-A, which predicts a small, promising search space, based on the given application. Then we randomly sample a small number of candidates in the search space.

Step3: Estimate performance and make recommendations. LITE estimates the performance for each candidate. Suppose the real-production computing environment is $\mathbf{e}_w$, the application w is to be implemented on $\mathbf{d}_w$. A set of stages are associated with the application, with code and DAG scheduler features (i.e., $\mathbf{C}_i, \mathcal{G}_i$). Then, LITE aggregated over predicted performance (i.e., execution time) of all stages. Finally, LITE ranks the configurations. $\mathbf{o}_w^*$, which produces least aggregated execution time among all candidates, will be ranked highest and delivered as the suggested configuration:

$$\mathbf{o}_w^* = \arg\min_{\mathbf{o}_j\sim\mathcal{S}_w}\sum_{w(\mathbf{C}_i)=w(\mathcal{G}_i)=w} M_{\Theta^*}(\mathbf{o}_j, \mathbf{C}_i, \mathcal{G}_i, \mathbf{d}_w, \mathbf{e}_w), \tag{5}$$

where $\mathbf{o}_j\sim\mathcal{S}_w$ is the process of selecting knob candidates in the search space $\mathcal{S}_w$. $M_{\Theta^*}(\mathbf{o}_j, \mathbf{C}_i, \mathcal{G}_i, \mathbf{d}_w, \mathbf{e}_w)$ is NECS's prediction for stage-level execution time, $w(\mathbf{C}_i) = w(\mathcal{G}_i) = w$ indicates that the stages i are associated with application w.

Step4: Model update. Once NECS is trained, it is static and fits to applications on small-scale input data. We present Adaptive Model Update in Section IV-B, to periodically update NECS, when a predefined batch of new instances are collected (i.e., more testing applications are tuned), so

that the tuning performance can be improved over time. The `Adaptive Model Update` adopts adversarial learning to fine-tune `NECS`.

A. Adaptive Candidate Generation

It is possible that sampling from a huge search space will miss good configurations, while selecting too many candidates will increase tuning overhead. We present `Adaptive Candidate Generation`, which adapts and shrinks the search space $\mathcal{S}_w$ for each testing application w, so that good candidates are more likely to be sampled, given limited candidates. To efficiently generate candidates for application w, the region of interest $\mathcal{S}_w \in \mathbb{R}^D$ for D knobs is a subset of the original configuration space. Thus, to identify the region of interest $\mathcal{S}_w \in \mathbb{R}^D$, we first roughly predict for each knob $d = 1 \sim D$ where an appropriate "mean value" should lie. Intuitively, the "mean value" should be related to the input datasize and the application. We build a Random Forest Regression (RFR) model $RFR^d(\mathbf{d}_w, \mathbf{a}_w)$, to map the input datasize and the application to the knob value. To simplify notations, we omit the superscript d if no ambiguity. We use the input data feature $\mathbf{d}_w$ as described in Section III. We use a vector $\mathbf{a}_w$ to represent the application, where each entry of $\mathbf{a}_w$ denotes a code token. We use tf-idf to assign a weight to each token. RFR is trained over a training set $\mathcal{B}$, where each training instance is a triple, i.e., $\mathcal{B} = \{< \mathbf{d}_j, \mathbf{a}_j, \mathbf{o}_j >\}$. The parameters $\boldsymbol{\Phi}$ are optimized by Equation 7.

$$\boldsymbol{\Phi}^* = \arg\min_{\boldsymbol{\Phi}} \sum_{j \in \mathcal{B}} \big(RFR(\mathbf{d}_j, \mathbf{a}_j) - \mathbf{o}_j \big)^2. \qquad (6)$$

The training set $\mathcal{B}$ is constructed as follows. As `LITE` repeatedly runs each application with various configuration settings on different datasets, we collect the execution time of each application instance. We want to reduce training expense, and exclude noisy examples with bad configurations and let RFR to predict a range of good knob values. Thus, for each application and input datasize in $\mathcal{DS}$, we select instance with the least execution time and include it in $\mathcal{B}$. For example, suppose for application "Terasort" on 10MB input dataset, we have two instances, with different configurations, and "executor.cores=4" yields less execution time than "executor.cores=8". We will include the instance (10MB,"Terasot", "executor.cores=4") in $\mathcal{B}$. Using the training set $\mathcal{B}$ constructed as above transfers the knowledge in the "best" training instances.

We next determine the boundary of $\mathcal{S}_w^d$, by extending from the "mean value" $RFR^d(\mathbf{d}_w, \mathbf{a}_w)$, as Equation 7:

$$\mathcal{S}_w^d = [RFR^d(\mathbf{d}_w, \mathbf{a}_w) - \sigma^d, RFR^d(\mathbf{d}_w, \mathbf{a}_w) + \sigma^d], \qquad (7)$$

where σ^d is the span of the search space, from the center to the boundary. To derive σ^d, we use the standard deviation of the knob values in $\mathcal{DS}$. To be specific, we choose top 40% instances with lowest application instance execution time, compute the mean configuration value for these samples on the $d-$th knob, and consequently obtain its sandard deviation σ^d.

B. Adaptive Model Update

Suppose running `LITE` for a period of time, we have obtained $\mathcal{DT}$, which collects our tuning feedback in real-production systems, i.e., applications implemented on large input datasize, and the actual execution time of recommended
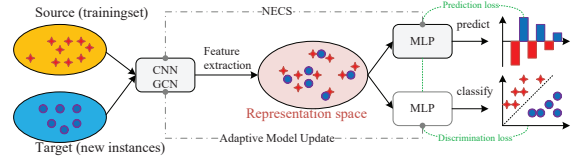

Fig. 6: Illustration of the `Adaptive Model Update` configurations. We can utilize $\mathcal{DT}$ to update `NECS`. The intuition is shown in Figure 6, we denote the input of training instances in $\mathcal{DS}$ as source domain and $\mathcal{DT}$ as target domain. Source domain and target domain have different distributions (e.g., different datasizes, computing environments and applications). Since the parameters of `NECS` are trained using $\mathcal{DS}$, the extracted feature embeddings by `NECS` from $\mathcal{DS}$ and $\mathcal{DT}$ may be non-overlapping, leading to degraded prediction performance for $\mathcal{DT}$.

The solution is to fine-tune `NECS` to obtain domain-invariant feature representations, so that the model parameters are updated and will be proper on target domain $\mathcal{DT}$. As shown in Figure 6, in `NECS`, the input instance flows through the feature encoding module (i.e., CNN and GCN) and the performance estimation module (i.e., MLP). Thus, we define the hidden layer embeddings from MLP in Equation 3 as the extracted feature embeddings, i.e., $\mathbf{h}_i = \big(f^1(x_i) \| \cdots \| f^L(f^{L-1}) \big)$, where $\|$ is the concatenation operation, L is the number of layers in the predictor MLP in `NECS`. Inspired by adversarial learning, we add an additional discriminator, in `Adaptive Model Update`. We label each instance as being from the source (i.e., $l(\mathbf{h}) = 1$ if $\mathbf{h} \in \mathcal{DS}$) or the target domain (i.e., $l(\mathbf{h}) = 0$ if $\mathbf{h} \in \mathcal{DT}$). The discriminator D attempts to distinguish source domain instance from target domain instance, given the feature embeddings extracted by `NECS`, i.e., $D_{\boldsymbol{\Omega}}(\mathbf{h}_i) \in (0, 1)$ is the probability of $\mathbf{h}_i$ being from the source domain, where $\boldsymbol{\Omega}$ are learnable discriminator parameters. We use MLP to implement our discriminator, the activation function in each layer is ReLU, the output layer is sigmoid.

Finally, we update the parameters of `NECS` and discriminator, i.e., $\boldsymbol{\Theta}, \boldsymbol{\Omega}$ by optimizing the *adaptive loss* in Equation 8:

$$L = \min_{\boldsymbol{\Theta}} \max_{\boldsymbol{\Omega}} \big(L_p + L_D \big), \qquad (8)$$

where $L_p = \sum_{i \in \mathcal{DS} \bigcup \mathcal{DT}} \big(y_i - M_{\boldsymbol{\Theta}}(x_i) \big)^2$ is the accumulated *prediction loss* on both source and target domain, $M_{\boldsymbol{\Theta}}(x_i)$ is the predicted execution time by `NECS`, y_i is the actual execution time of instance i. $L_D = \sum_{i \in \mathcal{DS} \bigcup \mathcal{DT}} \big(- l(\mathbf{h}_i) \log D_{\boldsymbol{\Omega}}(\mathbf{h}_i) - \big(1 - l(\mathbf{h}_i) \big) \log \big(1 - D_{\boldsymbol{\Omega}}(\mathbf{h}_i) \big) \big)$ is the *discrimination loss*, which is a cross entropy loss to evaluate the binary classification, $D_{\boldsymbol{\Omega}}(\mathbf{h}_i)$ is the discriminator's output probability of instance i's feature embedding $\mathbf{h}_i$. The updating process plays a *minimax game*. By incorporating the prediction loss in $\mathcal{DT}$, the estimation performance on target domain is naturally improved. By maximizing the discriminator loss with respect to $\boldsymbol{\Theta}$, we encourage `NECS` to learn feature representations for target domain which are hard to distinguish from the source domain. By minimizing the discriminator loss

TABLE III: Evaluation clusters

Cluster	#Nodes	#Cores	CPU	Memory	Network
Cluster A	1	16	3.2GHZ	64GB - 2400	1Gbps
Cluster B	3	16	3.2GHZ	64GB - 2400	1Gbps
Cluster C	8	16	2.9GHZ	16GB - 2666	10Gbps

TABLE IV: 16 key performance-aware configuration knobs

Parameter Name	Brief Description
"spark.default.parallelism"	Number of RDD partitions
"spark.driver.cores"	Number of cores by driver process
"spark.driver.maxResultSize"	Max size of serialized results per Spark action
"spark.driver.memory"	Memory size for driver process
"spark.driver.memoryOverhead"	Off-heap memory size per driver
"spark.executor.cores"	Number of cores per executor
"spark.executor.memory"	Memory size per executor process
"spark.executor.memoryOverhead"	Off-heap memory size per executor
"spark.executor.instances"	initial number of executors
"spark.efiles.maxPartitionBytes"	Max size per partition during file reading
"spark.memory.fraction"	Fraction for execution and storage memory
"spark.memory.storageFraction"	Storage memory percent exempt from eviction
"spark.reducer.maxSizeInFlight"	Max map outputs to collect concurrently per reduce task
"spark.shuffle.compress"	Compress map output (Boolean)
"spark.shuffle.file.buffer"	In-memory buffer size per output stream
"spark.shuffle.spill.compress"	Compress data spilled during shuffles (Boolean)

the discriminator tries to accurately identify instances. The goal is to reach an equilibrium in which the training instance and the test instance are projected to the same representation space, while retaining satisfying performance estimation. This makes NECS adapt to the target domain more easily.

V. EXPERIMENTS

In this section, we conducted extensive experiments to evaluate the proposed techniques and studied the following aspects of our approaches[1].

RQ1: *Did* LITE *gain high quality tuning results?* We focus on the **tuning performance** (i.e., How did LITE's recommended configurations speed up various Spark applications on large-scale input data?) and **tuning overhead** (i.e., How much time did it take to produce good tuning results?).

RQ2: *How well did each component in* LITE *perform?* This question is split into four sub-questions. **RQ2.1:** Can NECS properly rank the candidate configurations, i.e., placing better performing configurations at higher positions? **RQ2.2:** Can Stage-based Code Organization increase training set size and sample complexity? **RQ2.3:** Can Adaptive Candidate Generation improve tuning effectiveness? **RQ2.4:** Can Adaptive Model Update improve LITE's prediction accuracy?

RQ3: *How well did* LITE *generalize to new scenarios?* In particular, we ask two sub-questions. **RQ3.1:** Can LITE generalize to never-seen applications? **RQ3.2:** Can LITE generalize to different computing environments?

A. Experimental setup

Environment. As listed in Table III, we provisioned three typical Spark clusters, separately with 1, 3, 8 nodes. Each node was configured with 16-core CPUs, 64GB or 16GB memory, and 1Gbps or 10Gbps network connections.

Configurations. The knobs used in LITE are listed in Table IV, involving the important aspects like MapReduce, RDD, data compression, and storage management [11].

Applications. Table V lists the applications implemented in a standard benchmark spark-bench [1]. We covered a wide range of machine learning (ML), graph and MapReduce algorithms, which have rich semantic and code structures.

Training data of small sizes. For each application, we used four different input datasizes on every cluster (Table V). The datasizes are as small as possible so that each application can be finished in about one minute. Note that "LabelPropagation" is a graph application and we recorded the number of nodes instead of MegaBytes or GigaBytes.

Validation data of middle sizes. To validate the accuracy of configuration ranking, we used mid-scale input datasize for each application on every cluster (Table V). Note that the validation datasizes are still significantly larger than the input datasizes in each cluster's training sets.

Testing data of large sizes. To testify the tuning results, we also ran each application on large datasets[2] in cluster C to simulate big data analytic tasks in real-world systems (Table V). Note that for each application, we used the same seed to randomly sample from the same distribution to synthesize training, validation and testing data and ensure they have similar characteristics.

B. Performance Comparison

To answer **RQ1**, we evaluate tuning results on testing data of large sizes in Table V. We used the highest ranked $\mathbf{o}_w^*$ in Section IV to set the configuration.

Competitors. (1) Default: applications were implemented using default Spark configurations. (2) Manual: we hired experts to tune the applications based on online tuning guides for maximally 12 hours. (3) MLP: we used a machine learning baseline which feeds a Multi-Layer Perceptron with the application name, data features, environment features, and stage-level data statistics obtained in the Spark monitor UI. Thus, this baseline uses the same prediction module (i.e., MLP) with LITE but without code features. (4) BO(2h): we used Bayesian Optimization, where Gaussian Process was the surrogate model and Expected Improvement was the acquisition function. Furthermore, inspired by Ottertune [23], we used 5 most similar instances in the training set to initialize Gaussian Process. (5) DDPG(2h): following [15], we built a reinforcement learning framework which used Deep Deterministic Policy Gradient, where the action space was the configurations, and the state was the inner status summary of Spark. (6) DDPG-C(2h): similar as QTune [17], an additional module is incorporated in DDPG, which predicts the change of outer metric based on code features and inner status. BO, DDPG, and DDPG-C tuned each application for at least 2 hours.

Metrics. To make tuning performance comparable across applications, we computed *Execution Time Reduction*, defined as

---

[1]The source code, data, and/or other artifacts have been made available at https://github.com/cheyennelin/LITE.

[2]The input datasize significantly differs for each application. We use repeated trials to determine the largest possible datasize.

TABLE V: Application statistics: application name, type, input datasize in training set, validation set and test set.

Application name (Abbreviation)	Type	Input Datasize					
		Training		Testing	Validation		
		Cluster A/B	Cluster C	Cluster C	Cluster A	Cluster B	Cluster C
PrincipalComponentAnalysis(PCA)	ML	$\{0.9, 2.3, 3.5, 4.6\}$MB	$\{943.7, 1887.4, 2831.1, 3772.5\}$MB	102GB	46MB	23MB	9.21GB
ConnectedComponent(CC)	Graph	$\{63.5, 95.2, 127, 158.7\}$MB	$\{464.8, 547, 741, 938.5\}$MB	5.6GB	1.6GB	1.6GB	1.8GB
DecisionTree(DT)	ML	$\{38.4, 192.4, 384.8, 769.6\}$MB	$\{192.3, 384.7, 461.6, 577\}$MB	98GB	1.9GB	7.5GB	3.4GB
KMeans(KM)	ML	$\{186.6, 373.2, 559.8, 746.4\}$MB	$\{745.7, 1188.5, 1305, 1491.4\}$MB	92GB	3.6GB	3.6GB	4.4GB
LabelPropagation(LP)	Graph	$\{100, 200, 300, 400\}$ nodes	$\{500, 600, 650, 700\}$ nodes	2500	1000	1000	1000
LinearRegression(LR)	ML	$\{98, 196, 294, 392\}$MB	$\{1, 2, 3, 3.6\}$GB	137GB	3.83GB	3.83GB	12GB
Logisticregression(Logit)	ML	$\{190, 380, 570, 760\}$MB	$\{1.9, 3.8, 5.7, 6.84\}$GB	134GB	7.4GB	2.5GB	22.8GB
PageRank(PR)	Graph	$\{66.8, 94.9, 128.5, 164.2\}$MB	$\{161.7, 250.6, 323.4, 404.3\}$MB	9.2GB	328MB	328MB	812.4MB
PregelOperation(PO)	Graph	$\{161, 322, 483, 644\}$MB	$\{169.5, 381.4, 339, 423.8\}$MB	5.5GB	6.3GB	3.2GB	836.2MB
ShortestPaths(SP)	Graph	$\{61.9, 92.8, 123.8, 154.7\}$MB	$\{171.8, 229.1, 286.3, 343.6\}$MB	9.3GB	773MB	464MB	572.1MB
StronglyConnectedComponent(SCC)	Graph	$\{30.7, 61.4, 92.1, 122.8\}$KB	$\{6.3, 10.08, 12.6, 18.9\}$MB	260.9MB	1.2MB	0.6MB	31.5MB
SingularVectorDecomposition(SVD^{++})	ML	$\{6.7, 13.4, 20.1, 26.8\}$MB	$\{76.6, 153.2, 230, 306.4\}$MB	1GB	268MB	268MB	514.2MB
SupportVectorMachine(SVM)	ML	$\{20, 39, 59, 79\}$MB	$\{1.5, 3.0, 4.5, 6.0\}$GB	102GB	1.5GB	395MB	13.1GB
TeraSort(TS)	MapReduce	$\{95.4, 190.8, 286.2, 381.6\}$MB	$\{1.4, 1.9, 2.4, 2.8\}$GB	74.5GB	7.4GB	3.7GB	4.8GB
TriangleCount(TC)	MapReduce	$\{39.2, 118.2, 446.3, 1002.3\}$KB	$\{61.8, 92.7, 123.6, 154.5\}$MB	359.2MB	16MB	5MB	229.1MB

TABLE VI: Actual execution time t and execution time reduction ETR by different tuning methods

Method	Default	Manual	MLP	BO(2h)	DDPG(2h)	DDPG-C(2h)	LITE
t (seconds)	7314.93	1329.73	944.4	1125.13	2236.27	2244.93	**588.59**
ETR	0.07	0.63	0.62	0.69	0.44	0.48	**0.99**

$ETR = t_{min}/t$, for each application. For LITE, t is the actual execution time (in seconds) of the application configured as the first recommendation by LITE. For each competitor, t is the least execution time of the application during the tuning period. For any method, if the actual execution time was longer than two hours, or if the application failed, we recorded as the upper execution time, i.e., $t = 7200$. t_{min} is the minimal execution time of the application by all tuning methods.

Analysis of tuning performance. As shown in Table VI, *LITE can make a significant improvement in application performance*. Applications tuned by LITE were averagely 356 seconds faster than MLP, 537 seconds faster than BO, 1648 seconds faster than DDPG, 1656 seconds faster than DDPG-C, and 741 seconds faster than Manual tuning by experts. It gained 99% execution time reduction averagely. As shown in Figure 7, *LITE was very robust in tuning different applications*. LITE achieved $ETR = 1$ (i.e., least execution time) in 13 out of 15 applications. Furthermore, LITE was among the best two tuning methods in the rest two applications. On the contrary, the competitors produced poor tuning results for some applications. For example, MLP's performance was particularly bad on DT, PR and TC, BO was incompetent on TS and PR, and DDPG failed on LR, SP, SVD++, and SVM. Finally, *LITE achieved good tuning results within a very small tuning overhead*. LITE took less than 2 seconds to make recommendations. Its recommendations were better than time consuming competitors such as BO and DDPG that were trained for at least two hours.

Case study of tuning overhead. We warm up BO and DDPG using similar training time and training set as trining LITE. We show two testing applications, i.e., DecisionTree (DT) and LinearRegression (LR) in Figure 8, where x represents the tuning overhead (i.e., the timestamp starting each training epoch), and y denotes the least actual execution time of the application until the current epoch. We also plotted the timestamp when LITE made configuration recommendations and the actual execution time of the recommended configuration. We have the following observations. (1) *LITE took the minimal tuning overhead*. This is because, after a fixed-time training process, LITE directly made predictions within 2 seconds. On the contrary, after warm-up, BO and DDPG still implemented iterative epochs, while each epoch contained model building (i.e., to collect data and update the model), predicting (i.e., to make predictions) and model probe (i.e., to attempt a new trial). Thus, each epoch was very time consuming, given the large input datasize. (2) *LITE gained near optimal tuning performance*. We can see that LITE's recommended configurations are very close to the best configuration that BO and DDPG can possibly achieve in a time-consuming manner. (3) BO and DDPG could not improve the execution time of sampled configuration at each epoch. For example, for DT, DDPG did not improve the initially sampled configurations. Thus the tuning process is inefficient.

C. Evaluating Performance Ranking

Next we evaluated the accuracy and performance of execution time estimator (**RQ2.1**). NECS provided the core function to deliver a ranking list of configuration candidates.

Metrics. We utilized two standard ranking evaluation metrics, which are frequently used in the information retrieval community [22], namely HR@K and NDCG@K. They are computed by comparing each method's topK ranking result with the gold-standard list. Due to the time-consuming procedure in deriving a gold-standard list, the evaluation was conducted on validation data with mid-scale input data.

Competitors. We compared NECS with a variety of machine learning models with different feature encoding modules and performance estimation modules. Performance estimation modules include LightGBM [14] and MLP. We fed these models with three types of features. The first type of features contains only application name without codes. (1) "W": application instance features (i.e., data features and environment features); (2) "S": stage-level features including the data features, environment features, key stage-level data statistics obtained in the Spark monitor UI such as stage input. Note that the stage-level data statistics were not used in our proposed model NECS, because they are only accessible when the application has been actually executed on the real input data and it will be problematic to handle large-scale input data. The second type of features includes codes. (3)"WC": the application instance features and bag-of-words (BOW) representation of program codes of the application. (4) "SC": stage-level features and BOW representation of stage-level
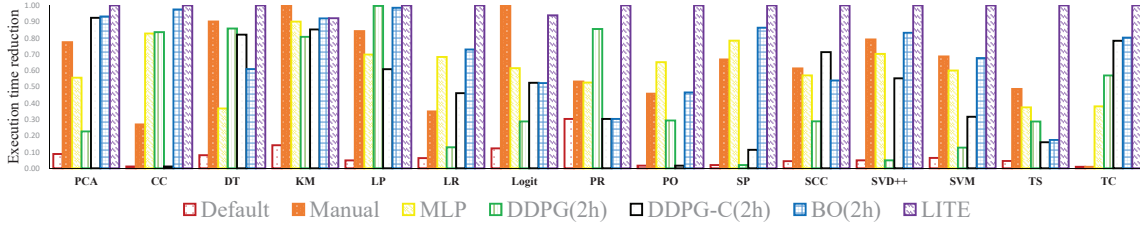
Fig. 7: Each application's ETR (execution time reduction) by different methods. Higher ETR suggests better tuning performance.
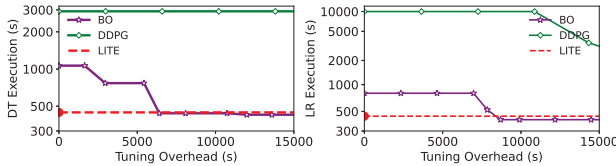

Fig. 8: Tuning DecisionTree and LinearRegression: tuning performance v.s. tuning overhead

Fig. 9: Training set size and complexity increased after data augmentation

codes. The third type of features adopt both codes and DAGs. (5) "SCG": stage-level code features, and pretrained scheduler features using LSTM (i.e., scheduler DAGs are trained to predict "next" DAG). (6) "LSTM": a Long-Short-Term-Memory network was used to encode the stage-level codes. (7) "Transformer": a transformer network based on multi-head self attention was used to encode the stage-level codes. (8) "GCN": a Graph Convolutional Neural network was used to encode the stage-level scheduler. The last three features (i.e., LSTM, Transformer and NECS) are only adopted with MLP due to the computational convenience of gradient back propagation.

Ablation analysis. In Table VII, we reported the average HR@5 and NDCG@5 results over 15 applications in the three clusters and large jobs. We have the following remarks. (1) *NECS consistently outperformed all competitors in terms of both HR@5 and NDCG@5.* NECS did not only retrieve good configurations (i.e., high HR@5), but also distinguished good and better configurations (i.e., high NDCG@5). *NECS produced robust ranking performance over different applications and computing environments.* (2) NECS outperformed very strong competitors. For example, data statistics ("S+") are well regarded as having great impacts on execution time. However, without stage-level data statistic features, NECS was still able to make accurate predictions. This shows that the code and scheduler encoding components are superior in extracting hidden feature representations. (3) Furthermore, NECS was trained using instances on small input data, and it was able to make precise predictions for applications on large jobs. For large jobs, the HR@5 by NECS is 0.4175 and the NDCG@5 is 0.5669, which is nearly "10%" higher than the best competitor. (4) Isolating each module, we observe that using *code features generally outperformed non-code features on different prediction modules*, (i.e., "WC" outperformed "W", "SC" outperformed "S"); *stage-level codes using data augmentation outperformed ("SC") outperformed application codes without data augmentation ("WC").*
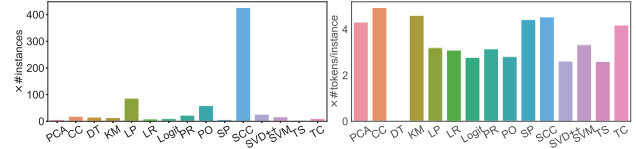
D. Evaluating `Stage-based Code Organization`

To answer **RQ2.2**, we first observed in Table VII, using the original application code without data augmentation ("WC") performs worse than stage-level codes ("SC") on all testing settings (i.e., cluster A,B,C and large jobs). Furthermore, the advantage of using stage-level codes is universally observed on different prediction modules (i.e., MLP and LightGBM).

To further study the effect of `Stage-based Code Organization`, we reported the number of training instances $|\mathcal{DS}|$ after data augmentation in Figure 9. The number of training instances were significantly increased, resulting in $4\times$ (e.g., TS) to $427\times$ (e.g., SCC) more instances. Furthermore, the number of tokens per instance for most applications was increased. The length of codes per training instance was averagely tripled. The increased number of code tokens in each training instance, along with the increased number of training instances, demonstrated that `Stage-based Code Organization` greatly enhanced sample complexity in the training set.

E. Evaluating `Adaptive Candidate Generation`

To answer **RQ2.3**, we first compared LITE with RFR, which is described in Section IV-A. RFR used a random forest regression model to map the input datasize and the application to an appropriate knob value. Since the prediction of random forest regression is numerical, for discrete valued knobs, we round the prediction to the nearest integer. We reported the average execution time reduction ETR and actural execution time t by RFR and LITE in Table VIIIa. We can see that, *using `Adaptive Candidate Generation` boosts tuning performance.* The underlying reason is that `Adaptive Candidate Generation` filtered a region of appropriate knob values, while RFR can only roughly identified one point which was risky.

We further testify whether `Adaptive Candidate Generation` can improve ranking performance over other sampling techniques. The comparative study was implemented for validation applications in cluster C. We used random

Prediction Module	Features		Cluster A		Cluster B		Cluster C		Large	
	Type	Feature	HR@5	NDCG@5	HR@5	NDCG@5	HR@5	NDCG@5	HR@5	NDCG@5
LightGBM	NoCode	W	0.4462	0.5586	0.2693	0.3746	0.3107	0.4190	0.2292	0.3194
		S	0.3825	0.5293	0.3973	0.5238	0.3784	0.5095	0.3090	0.4506
	Code	WC	0.4525	0.5755	0.3333	0.4490	0.3630	0.5181	0.2323	0.3144
		SC	0.4575	0.6127	0.4173	0.5470	0.3846	0.5466	0.3127	0.4348
	Code+DAG	SCG	0.4137	0.5524	0.3786	0.4867	0.3938	0.5134	0.2763	0.3969
MLP	NoCode	W	0.2100	0.3257	0.2080	0.3089	0.2015	0.2888	0.2276	0.2940
		S	0.3896	0.5466	0.3745	0.5015	0.3503	0.4832	0.3091	0.4424
	Code	WC	0.2945	0.4142	0.2538	0.3614	0.2561	0.3225	0.2384	0.3090
		SC	0.4137	0.6008	0.3806	0.5204	0.3621	0.4953	0.2789	0.4032
	Code+DAG	SCG	0.4100	0.6055	0.4026	0.5452	0.3598	0.5013	0.2818	0.4068
		LSTM+GCN	0.4253	0.6072	0.4050	0.5600	0.4053	0.5663	0.3875	0.5523
		Transformer+GCN	0.4250	0.5971	0.4016	0.5444	0.3906	0.5483	0.3600	0.5128
		NECS	**0.4706**	**0.6192**	**0.4440**	**0.5702**	**0.4283**	**0.5818**	**0.4175**	**0.5669**

sampling and `Adaptive Candidate Generation` to select candidates and estimated execution time for these candidates respectively. We reported the ranking performance in Table VIIIb. We can see that, with `Adaptive Candidate Generation`, NECS's recommendations were further improved. The underlying reason is that `Adaptive Candidate Generation` filtered bad configuration regions and thus avoided mistakes in pushing bad configurations to the top positions in the ranking results.

TABLE VIII: `Adaptive Candidate Generation` improved tuning performance and ranking performance.

(a) Tuning performance			(b) Ranking performance		
Method	RFR	LITE	Method	random	LITE
ETR	0.41	**0.99**	HR@5	0.417	**0.440**
t	3055.07	**588.59**	NDCG@5	0.562	**0.573**

F. Evaluating `Adaptive Model Update`

To study **RQ2.4**, we first trained NECS with the training instances in each cluster. Then, we used the static NECS to make recommendations on the cluster's validation data, denoted as "NECS". Next, we randomly split the validation data in each cluster into three folds, where each fold contained five applications. To simulate the scenario of online update, we updated NECS, using `Adaptive Model Update`, with one fold of the validation data in each cluster. We used the updated NECS to make recommendations on the rest two folds in the cluster's validation data. We reported the recommendations made by the updated NECS as "NECS_u". We repeated four runs and reported the average ranking performance by NECS and NECS_u in Table IX.

We have the following observations. (1) Although the static NECS already made satisfying results, `Adaptive Model Update` could further fine-tune NECS and significantly enhanced ranking performance. P-values of Wilcoxon Signed Rank Test for HR@5 and NDCG@5 on three clusters are all less than 0.5. (2) Improvements in prediction performance were observed across different clusters, which shows that the updating via adversarial learning based on newly collected feedback helps to adapt the model to the target domain (i.e., larger input datasize in validation set).

TABLE IX: Ranking performance HR@5 and NDCG@5 for NECS with and without `Adaptive Model Update` in different clusters. Wilcoxon Signed Rank Test was conducted to compute p-value of the increase.

	HR@5			NDCG@5		
	NECS	NECS_u	p-value	NECS	NECS_u	p-value
Cluster A	0.4586	**0.4706**	0.0494	0.6192	**0.6199**	0.0462
Cluster B	0.444	**0.4533**	0.0325	0.5702	**0.5888**	0.0339
Cluster C	0.4283	**0.4438**	0.0263	0.5818	**0.5870**	0.0413

TABLE X: Average (Avg.) and each never-seen application's ETR (execution time reduction) under cold-start setting

Application	PCA	CC	DT	KM	LP	LR	Logit	PR
ETR	0.98	0.95	0.94	0.75	0.98	1.00	0.95	0.95
Application	PO	SP	SCC	SVD ++	SVM	TS	TC	Avg.
ETR	0.94	0.93	0.95	0.98	1.00	1.00	0.97	0.95

G. Generalizing to Never-seen Applications

To answer **RQ3.1**, we first testify whether the proposed method can recommend good configurations for never-seen applications.

Evaluating protocols. We consider two settings. (1) Warm-start setting: predict execution time and deliver tuning recommendations for an existing application on a different dataset. Results in Section V-B are tuning performance under warm-start settings. (2) Cold-start setting: for each application (Table V), we excluded all training instances associated with it in the training set. The rest of the training instances was used to train the model and recommend configuration for this application on the large testing data in cluster C. We repeated the experiments for 15 times, each time removing one application, and reported the ETR (execution time reduction) results.

Analysis. As shown in Table X, the cold-start tuning for most (i.e., 11 out of 15) applications obtains $ETR \geq 0.95$. The average $ETR = 0.95$. Note that the best competitor in Table VI, i.e., BO, obtains an average $ETR = 0.69$ under warm-start settings. It shows that LITE can make near optimal tuning advice for never-seen applications.

H. Stability for Never-seen applications

Estimating performance for never-seen applications. We also testify whether NECS can make accurate estimation for never-seen applications. Results in Section V-C are obtained under the warm-start setting. We train NECS under the cold-

TABLE XI: Average ranking performance under warm-start and cold-start settings

Method	SCG+LightGBM		NECS		
	Cold-start	Warm-start	Cold-start	Cold-UNK	Warm-start
HR@5	0.3570	0.3938	**0.4030**	0.3960	**0.4283**
NDCG@5	0.5050	0.5134	**0.5543**	0.5480	**0.5818**

start setting for validation applications and compare NECS with SCG+LightGBM, which was the best competitor in one cold-start applications. We also provide a version of NECS, where we do not use an "out-of-vocabulary" token to represent unseen operations (Cold-UNK).

Analysis. As shown in Table XI, we observe that SCG+LightGBM showed significant performance decline under cold-start settings, compared with warm-start applications. This is reasonable, because cold-start applications lack information in historical logs. Nonetheless, NECS achieved satisfying ranking performance, in terms of HR@5 and NDCG@5, under cold-start settings. The underlying reason is that the proposed code and DAG scheduler encoding successfully captured fine-grained correlations for never-seen applications. The ablation study of oov token shows that, without the oov token, estimation for never-seen applications are less robust and the ranking accuracy has been decreased.

Performance stability. We next studied whether NECS can provide stable performance estimation for many never-seen applications. For each run, we trained NECS with $15-n, n = 1 \sim 14$ randomly chosen applications and made predictions for n never-seen applications. We conducted five runs and show the average ranking performance with respect to the percentage of never-seen applications, i.e., $x = n/15$, in Figure 10. We have the following observations. (1) The ranking performances, including HR@5 and NDCG@5, decreased as the percentage of never-seen applications increased. This is reasonable, because a larger portion of never-seen applications raised larger difficulty to generalize. (2) However, NECS obtained a strong ranking performance for $x \leq 0.4$. To demonstrate NECS's superiority, we plotted the ranking performance of the best competitor in Section V-C under warm-start settings (i.e., the "Best warm" dotted line in Figure 10). Comparing NECS's performance curve to the "Best warm" line, we see that, *even with* 40% *never-seen applications, NECS could still yield higher performance than the best competitor under warm-start settings*. This shows that NECS was very robust for never-seen applications. (3) Finally, NECS's performance generally degraded smoothly for $x \leq 0.7$. We plotted the average ranking performance of the competitors in Section V-C with all training instances (i.e., the "Avg warm" dotted line in Figure 10). Comparing NECS's performance curve to the "Avg warm" line, we see that, *with up to* 70% *never-seen applications, NECS could still generate satisfying performance, i.e., comparable to the average competitor under warm-start settings*.

I. Tuning Overhead for Never-Seen applications

For cold-start applications, LITE needed to perform instrumentation (more details in Section III-B) to quickly obtain stage-level codes and DAG scheduler. The instrumentation step causes an additional tuning overhead. However, since we
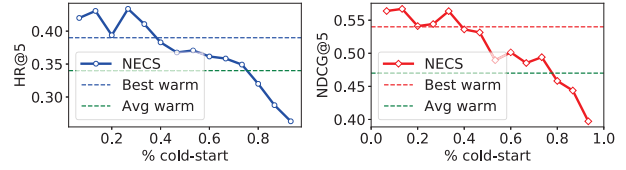

Fig. 10: NECS's ranking performance with respect to the number of never-seen applications
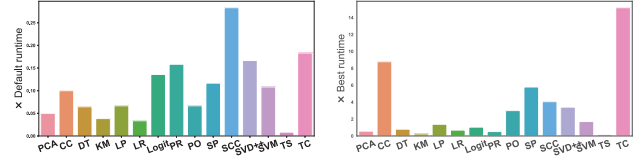

Fig. 11: Comparison between instrumentation time and execution time (by default configurations and best configurations tuned) for cold-start applications

did not use any stage-level data statistics (e.g., stage shuffle read/write), we implemented the application on the smallest dataset possible to reduce the instrumentation time. In Figure 11, the instrumentation time cost for cold-start applications fell in to the range of $[1, 30]$ minutes, which was quite small compared with running the default configurations on large-scale input data. For most applications, the instrumentation cost was even less than the least execution time of best configurations. We believe there is still room for optimizing the instrumentation step and we leave it as our future work. Since the logs are small (e.g., several MBs), parsing event logs for the other two steps took no more than 0.01 second.

J. Generalizing to Different Environments

To answer **RQ3.2**, we use different training sets to train the NECS model. For example, NECS trained with only instances in cluster C is denoted as "NECS_C"; NECS trained with instances in cluster A and B is denoted as "NECS_AB"; NECS trained with all instances in cluster A, B, and C is denoted as "NECS_all". Then we used these models to deliver predictions for validation applications in cluster C. From Table XII, we can see that, using all training instances, NECS produced better NDCG@5 performance than using only cluster C instances. This observation suggests that NECS can better transfer knowledges learned from different computing environments. The variety of computing environments in training set is beneficial to learn correlations among applications and computing environments.

VI. RELATED WORK

Knob Tuning for Analytical Platforms. Existing knob tuning methods for analytical platforms fall into four categories.
(1) *Rule based approaches* are tuning guidance based on expert experience and domain knowledge (e.g., cloudera, databricks). For example, they provide empirical equations for computing the partition numbers. However, they separately give hints on single aspects of knobs, and cannot consider more complex

TABLE XII: Ranking performance trained in different clusters

Method	NECS_AB	NECS_C	NECS_all
HR@5	0.2661	**0.4440**	0.4323
NDCG@5	0.377	0.5702	**0.5834**

multiple aspects of knobs. It is still laborious to tune knobs based on the limited rules. Besides, they only give guidance for typical scenarios and cannot cover different applications.

(2) *Experimental approaches* repeatedly execute applications with different configurations with search algorithms like recursive bound-and-search [34]. AutoTune [3] first constructs a testbed which samples different configurations for the given application, and then it adopts Latin Hypercube Sampling (LHS) to search for more promising configurations using both testbed and production system samples. DAC [27] proposes a hierarchical ensemble model to integrate individual models, each predicts the execution time based on configurations and input data sizes, and utilizes genetic algorithm to search for the optimal configuration. However, the constrained configuration space can still be very large and random sampling may not find high quality configurations within limited time.

(3) *Cost based approaches* design user-defined analytical cost functions, which take into account data statistics and configurations to predict application performance (e.g, execution time). And then they can use the predicted performance to guide configuration tuning. Ernest [24] builds cost models based on the behavior of sampled jobs and then predicts their performance on large datasets and cluster sizes. Dione [28] use a graph edit distance to detect similar spark jobs and reuse their prediction models to predict execution time. DynamiConf [6] assume execution time as a function of the number of nodes, and present a greedy search approach based on dependency graph to configure dynamic partitioning. A simulation based cost model is presented in [5] to optimize vcore and memory configurations. However, cost based approaches can only extract user-defined, simple interactions between a small part of factors. For example, Ernest [24] only models the interaction between the data scale and the inverse of the number of machines and cannot easily support other factors, which may lead to bad performance on complex applications.

(4) *Machine learning approaches* build performance prediction models by learning from history logs. Machine learning approaches are able to extract complex and high-dimension correlations between configurations and the performance, they usually achieve higher prediction accuracy. Various learning models have been adopted, like binary and multi-class classifiers [25], Gradient Boosting Regressors [9], logistic regression [13], Delaunay Triangulation [4], deep neural network [29]. However, those learning based methods only support limited scenarios by encoding the configuration features, and cannot adapt to large scale datasets and new applications.

Knob Tuning for Relational Databases. Fruitful research efforts have been devoted to auto-tuning cloud databases [8], [10], [37]–[43]. For the machine learning approaches, the line of OtterTune works [2], [23] combines machine learning models like Gaussian Process and Neural Networks to optimize the configuration sampling. CDBTune [31] and QTune [17] both adopt deep reinforcement learning to improve tuning performance. QTune generalizes to different workloads by encoding SQL features, but it cannot support complex code

structures in Spark applications. Restune [32] also utilizes Gaussian Process to select configurations based on resource utilization and performance requirements. Restune migrates to different databases by learning the tuning knowledge with meta learning. However, Restune still needs to finetune on real datasets, which may not be available or take relatively long time. Besides, there are works that specifically tune memory related knobs [20]. iBTune [20] designs a pair-wise deep neural network on instance's measurement features to predict the upper bounds of response time. ReLM [15] proposes analytical models to speed up a guided Bayesian optimization.

However, existing methods for relational databases cannot be directly applied to Spark. The reasons are three fold. First, they need numerous training data or repeatedly run on real datasets, which are unaffordable for Spark applications on large scale input data. Second, they only encode simple features like system metrics and SQL features, and cannot encode Spark applications with complex code structures and semantics. Third, many metrics they use in relational databases are unavailable in Spark (e.g, #readblocks).

Finally, due to the time efficiency, the offline-training, online-tuning framework has also been adopted in the system community [26]. The major difference is that in the online phase, we do not generate an explicit fingerprint by running the user-input task on a reference VM to collect resource usage and performance statistics. The online phase does not require to execute any task before giving the tuning advice.

VII. CONCLUSION

In this paper we proposed LITE to automatically tune configurations for big data Spark applications. Our experimental studies showed that LITE significantly reduces application execution time, compared with state-of-the-art tuning methods, for a wide variety of applications. Our contributions are generally applicable to any large-scale distributed data processing framework, including (1) the framework LITE that migrated the knowledge learned from small-scale datasets to large datasets, (2) the code encoding modules in NECS captured complex correlations among the available factors in codes and the scheduler, (3) adaptive model update Adaptive Model Update to fine-tune NECS via adversarial learning, and (4) adaptive candidate generation method Adaptive Candidate Generation that dynamically adjusted search region of interest and made fast and accurate recommendations.

References

[1] D. Agrawal, A. Butt, K. Doshi, J.-L. Larriba-Pey, M. Li, F. R. Reiss, F. Raab, B. Schiefer, T. Suzumura, and Y. Xia. Sparkbench – a spark performance testing suite. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, pages 26–44, Cham, 2016. Springer International Publishing.

[2] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *VLDB*, 2021.

[3] L. Bao, X. Liu, and W. Chen. Learning-based automatic parameter tuning for big data analytics frameworks. In N. Abe, H. Liu, C. Pu, X. Hu, N. K. Ahmed, M. Qiao, Y. Song, D. Kossmann, B. Liu, K. Lee, J. Tang, J. He, and J. S. Saltz, editors, *Big Data 2018*, pages 181–190. IEEE, 2018.

[4] Y. Chen, P. Goetsch, M. A. Hoque, J. Lu, and S. Tarkoma. d-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[5] Y. Chen, J. Lu, C. Chen, M. Hoque, and S. Tarkoma. Cost-effective resource provisioning for spark workloads. In W. Zhu, D. Tao, X. Cheng, P. Cui, E. A. Rundensteiner, D. Carmel, Q. He, and J. X. Yu, editors, *CIKM*, pages 2477–2480. ACM, 2019.

[6] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres. Dynamic configuration of partitioning in spark applications. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1891–1904, 2017.

[7] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866, 2021.

[8] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. In *TKDE*, 2020.

[9] Á. B. Hernández, M. S. Pérez, S. Gupta, and V. Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Gener. Comput. Syst.*, 86:1076–1092, 2018.

[10] X. Zhou, L. Jin, S. Ji, and et al. Dbmind: A self-driving platform in opengauss. *Proc. VLDB Endow.*, 14(12):2743–2746, 2021.

[11] H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv.*, 53(2), Apr. 2020.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*. The Association for Computer Linguistics, 2016.

[13] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee. Auto-tuning spark big data workloads on POWER8: prediction-based dynamic SMT threading. In A. Zaks, B. Mendelson, L. Rauchwerger, and W. W. Hwu, editors, *PACT*, pages 387–400. ACM, 2016.

[14] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *NIPS*, volume 30. Curran Associates, Inc., 2017.

[15] M. Kunjir and S. Babu. Black or white? how to develop an autotuner for memory-based analytics. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *SIGMOD*, pages 1667–1683. ACM, 2020.

[16] A. LeClair, S. Haque, L. Wu, and C. McMillan. Improved code summarization via a graph neural network. In *ICPC*, pages 184–195. ACM, 2020.

[17] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, Aug. 2019.

[18] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu. Improving code summarization with block-wise abstract syntax tree splitting. *CoRR*, abs/2103.07845, 2021.

[19] K. G. Srinivasa and A. K. Muppalla. *Guide to High Performance Distributed Computing - Case Studies with Hadoop, Scalding and Spark*. Computer Communications and Networks. Springer, 2015.

[20] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. Ibtune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow.*, 12(10):1221–1234, June 2019.

[21] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020.

[22] D. Valcarce, A. Bellogín, J. Parapar, and P. Castells. On the robustness and discriminative power of information retrieval metrics for top-n recommendation. In RecSys '18, page 260–268, New York, NY, USA, 2018. Association for Computing Machinery.

[23] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[24] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In K. J. Argyraki and R. Isaacs, editors, *NSDI*, pages 363–378. USENIX Association, 2016.

[25] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of spark based on machine learning. In J. Chen and L. T. Yang, editors, *ICDCS*, pages 586–593. IEEE Computer Society, 2016.

[26] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the ¡i¿best¡/i¿ vm across multiple public clouds: A data-driven performance modeling approach. In *SoCC*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Z. Yu, Z. Bei, and X. Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, editors, *ASPLOS*, pages 564–577. ACM, 2018.

[28] N. Zacheilas, S. Maroulis, and V. Kalogeraki. Dione: Profiling spark applications exploiting graph similarity. In J. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, and M. Toyoda, editors, *BigData*, pages 389–394. IEEE Computer Society, 2017.

[29] K. Zaouk, F. Song, C. Lyu, A. Sinha, Y. Diao, and P. Shenoy. Udao: A next-generation unified data analytics optimizer. *Proc. VLDB Endow.*, 12(12):1934–1937, Aug. 2019.

[30] B. Zhang, D. Van Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proc. VLDB Endow.*, 11(12):1910–1913, Aug. 2018.

[31] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[32] X. Zhang, H. Wu, Z. Chang, S. Jin, J. Tan, F. Li, T. Zhang, and B. Cui. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *SIGMOD*, pages 2102–2114. ACM, 2021.

[33] L. Zheng, V. Noroozi, and P. S. Yu. Joint deep modeling of users and items using reviews for recommendation. In M. de Rijke, M. Shokouhi, A. Tomkins, and M. Zhang, editors, *WSDM*, pages 425–434. ACM, 2017.

[34] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, pages 338–350. ACM, 2017.

[35] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308. IEEE, 2020.

[36] G. Li, X. Zhou, S. Ji, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *VLDB*, 2021.

[37] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *Proc. VLDB Endow.*, 14(12):3190–3193, 2021.

[38] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. In *PVLDB*, 2022.

[39] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*, 2019.

[40] X. Zhou, L. Liu, W. Li, and et al. AutoIndex: An Incremental Index Management System for Dynamic Workloads. In *ICDE*, 2022.

[41] H. Yuan, G. Li, L. Feng, and et al. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, 2020.

[42] J. Sun and G. Li. An end-to-end learning-based cost estimator. *VLDB*, 2019.

[43] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *IEEE Data Eng. Bull.*, 42(2):70–81, 2019.