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, Big Data Analytics (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].

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 datasets.

Fig. 1: Spark applications PageRank and TriangleCount execution time on 160MB input data, with different configurations.
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 (LIghtweigT knob recommen) 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 (Neural Estimator via Code and Scheduler 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 (Neural Estimator via Code and Scheduler 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. Next 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 ~ 5). Specifically, only line 4 distinguishes the function of this application (i.e., sorting).

2. 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., \( C_i \in \mathbb{R}^{D \times N} \), 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., \( G_i = \{ V_i, A_i \} \). The node embedding matrix is denoted by \( V_i \in \mathbb{R}^{1 \times (S+1)} \), 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., \( x_i = \langle o_i, C_i, G_i, d_i, e_i, y_i \rangle \), where \( o_i \) is an array of knob values, \( C_i, G_i \) are the code features and scheduler features extracted in Section III-B, \( d_i \) represents the data feature, \( e_i \) denotes the computing environment that this
Table I: Entries in the data feature

<table>
<thead>
<tr>
<th>Entry</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#rows</td>
<td>Number of rows in the input data</td>
</tr>
<tr>
<td>#columns</td>
<td>Number of columns in the input data</td>
</tr>
<tr>
<td>#iteration</td>
<td>Number of iterations (optional)</td>
</tr>
<tr>
<td>#partitions</td>
<td>Number of partitions (optional)</td>
</tr>
</tbody>
</table>

Table II: Entries in the environment feature

<table>
<thead>
<tr>
<th>Entry</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#nodes</td>
<td>Number of nodes (computers) in the cluster</td>
</tr>
<tr>
<td>#cores</td>
<td>Number of cores in each node</td>
</tr>
<tr>
<td>Frequency</td>
<td>CPU frequency (GHZ)</td>
</tr>
<tr>
<td>Memory size</td>
<td>Memory size (GB)</td>
</tr>
<tr>
<td>Memory speed</td>
<td>Memory speeds (MT/s)</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>Bandwidth in connecting the cluster</td>
</tr>
</tbody>
</table>

Fig. 3: NECS framework

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

Fig. 5: Stage-level codes (stage 3) of application Terasort after instrumentation. Important “map” operations are highlighted. The 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., $o_i = o_j, d_i = d_j, e_i, e_j$, 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 $o_i \in \mathbb{R}^D$ (more in Section V).

Data features. We use four entries that can be obtained through the application’s data generation phase before executing the application to describe data characteristics. That is, each data feature vector for instance $x_i$ is a four-dimensional vector, i.e., $d_i \in \mathbb{R}^4$. 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 $d_i$ 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., $e_i \in \mathbb{R}^6$, to represent the environment information.

D. Encoding Code Features

We apply feature encoding upon the code features $C_i$ to capture rich semantics of the codes. In the literature, sequence models such as LSTM (Long Short Term Memory network) [12], [35] and Transformers [18] have been adopted to learn representations for program codes. However, we use CNN (Convolutional Neural Network) 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 \( Q = \text{flat}(q_1, \ldots, q_l) \), where \( I \) denotes the number of kernels in the convolutional layer and \( \text{flat} \) is the flatten operation. For space limit, we omit the calculation of CNN.

The code representation \( h_{\text{code}, i} \) can be obtained by a ReLU transformation with \( W^{\text{CNN}} \), the learnable weight matrix:

\[
h_{\text{code}, i} = \text{ReLU}(W^{\text{CNN}}Q).
\]  

(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 (Graph Convolutional Network) [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., \( H^0_i = V_i \) and apply the operations \( H^{l+1} = \text{ReLU}(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^lW^{\text{GCN}l}) \), where the \( l + 1 \)-th layer of GCN outputs \( H^{l+1} \), \( A = A + I \), where \( A_i \in \mathbb{R}^{|V| \times |V|} \) is the adjacency matrix, \( I \in \mathbb{R}^{|V| \times |V|} \) is the identity matrix. \( D \) is the degree matrix, where its diagonal elements represent the degree of each node, its rest elements are set to be zero. \( H^l \) is the output of last layer. \( W^{\text{GCN}l} \) is the learnable weight matrix. Finally, the scheduler representation \( h_{\text{DAG}, i} \) can be obtained by a max pooling operation:

\[
h_{\text{DAG}, i} = \max H^L.
\]  

(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 (Multi-Layer Perceptron) module. The procedure can be formally defined in Equation 3:

\[
\hat{y}_i = f_L(\cdots f_1(\text{concat}(d_i, e_i, o_i, h_{\text{code}, i}, h_{\text{DAG}, i})) \cdots),
\]  

(3)

where \( L \) is the number of layers in MLP, \( f_l(\cdot) \) with \( l = 1, 2, \ldots, L \) denotes the mapping function for the \( l \)-th hidden layer. \( f_1(x) = \text{ReLU}(W^{\text{MLP}1}x + b^{\text{MLP}1}) \), where \( W^{\text{MLP}1} \) and \( b^{\text{MLP}1} \) are learnable weight matrix and bias vector for layer \( l \). The activation function for each layer is \( \text{ReLU} \). We use a tower MLP structure, where the size of layers (i.e., dimensionality of \( x \) 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 \( M_\Theta \) is the set of parameters including the learnable weights in CNN, GCN and MLP modules. The offline training phase optimizes \( M_\Theta \):

\[
\Theta^* = \arg\min_{\Theta} \sum_{x_i \in DS} (y_i - \hat{y}_i)^2.
\]  

(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 scheduler.

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.

Step 3: Estimate performance and make recommendations. LITE estimates the performance for each candidate. Suppose the real-production computing environment is \( e_w \), the application \( w \) is to be implemented on \( d_w \). A set of stages are associated with the application, with code and DAG scheduler features (i.e., \( C_i, G_i \)). Then, LITE aggregated over predicted performance (i.e., execution time) of all stages. Finally, LITE ranks the configurations. \( o_{w_i} \), which produces least aggregated execution time among all candidates, will be ranked highest and delivered as the suggested configuration:

\[
o_i^w = \arg\min_{o_j \sim S_w} \sum_{w(C_i) = w(G_i) = w} M_{\Theta^*}(o_j, C_i, G_i, d_w, e_{w_i}),
\]  

(5)

where \( o_j \sim S_w \) is the process of selecting knob candidates in the search space \( S_w \). \( M_{\Theta^*}(o_j, C_i, G_i, d_w, e_{w_i}) \) is NECS’s prediction for stage-level execution time, \( w(C_i) = w(G_i) = w \) indicates that the stages \( i \) are associated with application \( w \).

Step 4: 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 \( S^d_w \) for each testing application \( w \), so that good candidates are more likely to be sampled. To efficiently generate candidates for application \( w \), the region of interest \( S^d_w \) in the configuration space. Thus, to identify the region of interest \( S^d_w \), we first roughly predict for each knob \( d \) 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 \( RFR^d(a_w, a_w) \), to map the input datasize and the application to the knob value. To simplify notations, we omit the superscript \( d \).

We next determine the boundary of \( S^d_w \), by extending from the “mean value” \( RFR^d(a_w, a_w) \), as Equation 7:

\[
S^d_w = [RFR^d(a_w, a_w) - \sigma^d, RFR^d(a_w, a_w) + \sigma^d],
\]

where \( \sigma^d \) is the span of the search space, from the center to the boundary. To derive \( \sigma^d \), we use the standard deviation of the knob values in \( DS \). 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 standard deviation \( \sigma^d \).

B. Adaptive Model Update

Suppose running LITE for a period of time, we have obtained \( DT \), which collects our tuning feedback in real-production systems, i.e., applications implemented on large input datasize, and the actual execution time of recommended configurations. The intuition is shown in Figure 6, we denote the input of training instances in \( DS \) as source domain and \( DT \) 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 \( DS \), the extracted feature embeddings by NECS from \( DS \) and \( DT \) may be non-overlapping, leading to degraded prediction performance for \( DT \).

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 \( DT \). 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., \( h_i = \{f^1(x_i)|...|f^L(f^{L-1})\} \), 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., \( l(h) = 1 \) if \( h \in DS \)) or the target domain (i.e., \( l(h) = 0 \) if \( h \in DT \)). The discriminator \( D \) attempts to distinguish source domain instance from target domain instance, given the feature embeddings extracted by NECS, i.e., \( D(h_i) \in (0,1) \) is the probability of \( h_i \) being from the source domain, where \( \Omega \) 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 \( \text{NECS and discriminator, } \theta, \Theta \) by optimizing the adaptive loss in Equation 8:

\[
L = \min_{\theta} \max_{\Theta} \left( L_p + L_D \right),
\]

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

**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.

**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 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 $o^*_n$ 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

\[
\text{ETR} = \frac{\text{Execution Time}_{\text{Default}} - \text{Execution Time}_{\text{LITE}}}{\text{Execution Time}_{\text{Default}}}
\]

\[
\text{ETR} \times 100\% = \frac{\text{Execution Time}_{\text{Default}} - \text{Execution Time}_{\text{LITE}}}{\text{Execution Time}_{\text{Default}}} \times 100\%.
\]

\[
\text{ETR} = \frac{\text{Execution Time}_{\text{Default}} - \text{Execution Time}_{\text{LITE}}}{\text{Execution Time}_{\text{Default}}} \times 100\%.
\]

**TABLE III: Evaluation clusters**

<table>
<thead>
<tr>
<th>Cluster</th>
<th>#Nodes</th>
<th>#Cores</th>
<th>CPU</th>
<th>Memory</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster A</td>
<td>1</td>
<td>16</td>
<td>3.2GHz</td>
<td>64GB</td>
<td>2400</td>
</tr>
<tr>
<td>Cluster B</td>
<td>3</td>
<td>16</td>
<td>3.2GHz</td>
<td>64GB</td>
<td>2400</td>
</tr>
<tr>
<td>Cluster C</td>
<td>8</td>
<td>16</td>
<td>2.9GHz</td>
<td>16GB</td>
<td>2666</td>
</tr>
</tbody>
</table>

**TABLE IV: 16 key performance-aware configuration knobs**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“spark.default.parallelism”</td>
<td>Number of RDD partitions</td>
</tr>
<tr>
<td>“spark.driver.cores”</td>
<td>Number of cores by driver process</td>
</tr>
<tr>
<td>“spark.driver.maxResultSize”</td>
<td>Size of serialized results per Spark action</td>
</tr>
<tr>
<td>“spark.driver.memory”</td>
<td>Off-heap memory size per driver</td>
</tr>
<tr>
<td>“spark.executor.cores”</td>
<td>Number of cores per executor</td>
</tr>
<tr>
<td>“spark.executor.cores”</td>
<td>Memory size for executor process</td>
</tr>
<tr>
<td>“spark.executor.memory”</td>
<td>Off-heap memory size per executor</td>
</tr>
<tr>
<td>“spark.executor.instances”</td>
<td>Initial number of executors</td>
</tr>
<tr>
<td>“spark.files.maxPartitionBytes”</td>
<td>Max size per partition during file reading</td>
</tr>
<tr>
<td>“spark.memory.fraction”</td>
<td>Fraction for execution and storage memory</td>
</tr>
<tr>
<td>“spark.memory.storageFraction”</td>
<td>Storage memory percent exempt from eviction</td>
</tr>
<tr>
<td>“spark.reducer.maxSizeInFlight”</td>
<td>Max map outputs to collect concurrently per reduce task</td>
</tr>
<tr>
<td>“spark.shuffle.compress”</td>
<td>In-memory buffer size per output stream</td>
</tr>
<tr>
<td>“spark.shuffle.file.buffer”</td>
<td>Compress data spilled during shuffles (Boolean)</td>
</tr>
</tbody>
</table>

1The source code, data, and/or other artifacts have been made available at https://github.com/cheyennelin/LITE.
For each application, 
LITE, t is the actual execution time (in seconds) of the application configured as the first recommendation by LITE. For each competitor, 
LITE 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.,

$$t = 7200.\text{min}$$ 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 tuning 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 dataset. (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 NDCC@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
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”).

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 |D| after data augmentation in Figure 9. The number of training instances were significantly increased, resulting in 4× (e.g., TS) to 427× (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 actual 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
TABLE VII: Ranking performance by various machine learning methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Features</th>
<th>Cluster A</th>
<th>Cluster B</th>
<th>Cluster C</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>LightGBM</td>
<td>W</td>
<td>0.4462</td>
<td>0.3386</td>
<td>0.2695</td>
<td>0.3176</td>
</tr>
<tr>
<td>Code</td>
<td>WC</td>
<td>0.4523</td>
<td>0.3755</td>
<td>0.3333</td>
<td>0.4490</td>
</tr>
<tr>
<td>Code+DAG</td>
<td>SCG</td>
<td>0.4117</td>
<td>0.3554</td>
<td>0.3786</td>
<td>0.4867</td>
</tr>
<tr>
<td>MLP</td>
<td>W</td>
<td>0.2100</td>
<td>0.3257</td>
<td>0.2080</td>
<td>0.3089</td>
</tr>
<tr>
<td>Code</td>
<td>WC</td>
<td>0.2945</td>
<td>0.4142</td>
<td>0.2538</td>
<td>0.3614</td>
</tr>
<tr>
<td>Code+DAG</td>
<td>SCG</td>
<td>0.4177</td>
<td>0.5008</td>
<td>0.3906</td>
<td>0.5304</td>
</tr>
<tr>
<td>Transformer+GCN</td>
<td>SCG</td>
<td>0.4100</td>
<td>0.6055</td>
<td>0.4026</td>
<td>0.5452</td>
</tr>
</tbody>
</table>

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 two 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 dataset 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.

<table>
<thead>
<tr>
<th>Application</th>
<th>NECS</th>
<th>NECS_u</th>
<th>p-value</th>
<th>NECS</th>
<th>NECS_u</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster A</td>
<td>0.4586</td>
<td>0.4706</td>
<td>0.0494</td>
<td>0.6199</td>
<td>0.0462</td>
<td></td>
</tr>
<tr>
<td>Cluster B</td>
<td>0.444</td>
<td>0.4533</td>
<td>0.0025</td>
<td>0.5702</td>
<td>0.0359</td>
<td></td>
</tr>
<tr>
<td>Cluster C</td>
<td>0.4283</td>
<td>0.4383</td>
<td>0.0063</td>
<td>0.3818</td>
<td>0.0413</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Application</th>
<th>NECS</th>
<th>NECS_u</th>
<th>p-value</th>
<th>NECS</th>
<th>NECS_u</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster A</td>
<td>0.98</td>
<td>0.95</td>
<td>0.94</td>
<td>0.75</td>
<td>0.98</td>
<td>0.95</td>
</tr>
<tr>
<td>Cluster B</td>
<td>0.94</td>
<td>0.93</td>
<td>0.95</td>
<td>0.98</td>
<td>1.00</td>
<td>0.97</td>
</tr>
<tr>
<td>Cluster C</td>
<td>0.94</td>
<td>0.93</td>
<td>0.95</td>
<td>0.98</td>
<td>1.00</td>
<td>0.97</td>
</tr>
</tbody>
</table>

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 > 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

<table>
<thead>
<tr>
<th>Method</th>
<th>NECS</th>
<th>SCG+LightGBM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cold-start</td>
<td>Warm-start</td>
</tr>
<tr>
<td>HR@5</td>
<td>0.3570</td>
<td>0.3938</td>
</tr>
<tr>
<td>NDCG@5</td>
<td>0.5050</td>
<td>0.5154</td>
</tr>
</tbody>
</table>

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

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 knowledge 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

<table>
<thead>
<tr>
<th>Method</th>
<th>NECS_AB</th>
<th>NECS_C</th>
<th>NECS_all</th>
</tr>
</thead>
<tbody>
<tr>
<td>HR@5</td>
<td>0.2661</td>
<td>0.4440</td>
<td>0.4323</td>
</tr>
<tr>
<td>NDCG@5</td>
<td>0.377</td>
<td>0.5702</td>
<td>0.5834</td>
</tr>
</tbody>
</table>

Authorized licensed use limited to: Xiamen University. Downloaded on September 20,2022 at 01:40:05 UTC from IEEE Xplore. Restrictions apply.
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.

Experimental approaches repeatedly execute applications with different configurations and 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.

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.

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.

VIII. ACKNOWLEDGEMENTS

Guoliang Li is the corresponding author. Chen Lin is supported by the Natural Science Foundation of China (No. 61972328), Alibaba Group through Alibaba Innovative Research program. Guoliang Li is supported by NSF of China (61925205, 62072261). Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist). Hui Li is supported by the Natural Science Foundation of China (No. 62002303), Natural Science Foundation of Fujian Province China (No. 2020J05001).