CLIC: An Extensible and Efficient Cross-Platform Data Analytics System

Qixiang Chen, Zhijun Chen, Kai Zhang, and X. Sean Wang, *Senior Member, IEEE*

***Abstract*—With the ever-increasing data volume and application diversity, a modern data analytics job is generally built as a workflow consisting of multiple tasks. For either specific functionalities or higher performance, tasks in a workflow may need to be deployed on different data processing platforms. This article proposes CLIC, a highly extensible system for efficient cross-platform data analytics. To leverage the advantage of diverse platforms while alleviating development efforts, we propose an embedding-based operator encoding scheme and a Graph Convolutional Network model for efficient platform selection. Aiming at flexibly integrating new operators and platforms, CLIC is designed with a highly extensible system architecture that decouples the core functionalities from backend platforms. Experiments show that CLIC can significantly improve the performance of modern data analysis workflows with fast platform selection.**

***Index Terms*—Data analysis, data processing, data systems, systems.**

I. Introduction

RECENT progresses in Big Data and artificial intelligence have significantly enhanced and energized data analytics. With diverse goals in performance and programming efficiency, data analytics platforms targeting different domains are constantly emerging. For instance, tasks such as data cleaning and filtering are generally performed on a DBMS or Big Data platforms like Spark [1]; tasks applied on the graph structure such as relationship discovery are performed by GraphX or Giraph [2]; deep learning models are built and trained by platforms like Tensorflow [3] and Pytorch [4], to name a few. Besides the functionalities, a data processing platform may deliver much higher performance for specific tasks due to the specialized system design. Therefore, an efficient modern data analytics workflow is usually built as a sequence of tasks that are separately executed on multiple platforms to leverage their advantages [5], [6], [7].

Since platforms generally have fundamentally different programming models, API standards, and performance advantages, etc., building such a cross-platform workflow while achieving high performance is complex and intricate. It not only demands expertise for all involved platforms which causes a high learning curve, but also needs to develop ad-hoc programs to orchestrate them which is time-consuming. These issues give rise to a spectrum of cross-platform systems that facilitate the development of cross-platform workflows [5], [6], [8], [9], [10]. These systems support multiple computing paradigms like batch processing and machine learning by integrating platforms from multiple domains as the backends. According to factors including platform efficiency and data movement overhead, a cross-platform system automatically selects platforms for the tasks of a given workflow to maximize its overall performance. These research prototypes have demonstrated the effectiveness of cross-platform computing. However, there are still two key aspects that have not been effectively addressed, which hinders the adoption of cross-platform data analytics in production systems.

1) Low System Extensibility: A system should be highly extensible since it needs to continuously integrate new operators to build workflows. However, due to the operator encoding technique, the machine learning model adopted in state-of-the-art systems, such as Robopt [10], has to be re-trained when integrating a new operator each time. Besides that, being designed as a monolithic architecture, system components and supported computing platforms are highly coupled. Consequently, integrating a new operator or platform needs to build, test, and deploy the entire system. The current system design incurs high costs to add a new operator, which severely limits the system extensibility.

2) Inefficient Platform Selection: Platform selection has a significant impact to the overall performance of a workflow. Therefore, its procedure should be efficient and robust. To derive the optimal execution plan for a workflow, state-of-the-art studies [6], [10] leverage cost models to estimate the execution time of a workflow when running on different platform combinations. With the estimated cost, available execution plans are enumerated and evaluated in a large search space, resulting in nontrivial overheads in platform selection. Moreover, the costs grows with the number of operators and platforms [10]. Consequently, performance is even degraded for small jobs, because the caused overhead can offset the performance advantages of selected platforms.

To address the above issues, we propose CLIC, a highly-extensible cross-platform data analytics system targeting efficient platform selection and flexible operator integration. By abstracting platform selection as a node classification problem, CLIC adopts a Graph Convolutional Network (GCN) model to capture the topological information of a workflow and derive the optimal execution plan. As the input for the GCN model, an embedding-based operator encoding technique is adopted to

Manuscript received 6 June 2022; revised 7 July 2023; accepted 18 July 2023. Date of publication 24 July 2023; date of current version 20 November 2023. Recommended for acceptance by A. Randles. *(Corresponding author: Kai Zhang.)*

The authors are with the School of Computer Science, Technology, Fudan University, Shanghai 200437, China (e-mail: qxchen19@fudan.edu.cn; zjchen20@fudan.edu.cn; kay21s@gmail.com; xywangCS@fudan.edu.cn).

Digital Object Identifier 10.1109/TPDS.2023.3298038

generate a dense and fixed-length feature vector to represent each operator. The graph convolutional process makes the platform selection extremely fast, with no need to enumerate all combinations. Moreover, it effectively enhances system extensibility by avoiding re-training the model when adding a new operator. Besides, CLIC is designed with an extensible architecture where the core system functionalities and backend platforms are fully decoupled as separate components. Each backend platform exists in the form of a container image and is instantiated when the included operators need to be deployed for execution. In this way, an operator implementation can be flexibly integrated or updated at run time, without influencing the core components of the system.

The contributions of this paper are as follows.

- We propose an end-to-end GCN-based approach to provide robust and fast platform selection.
- We propose a novel embedding-based operator encoding method to enhance the system extensibility for integrating new operators.
- We design and implement CLIC, a highly-extensible cross-platform system prototype with a decoupled architecture.

The roadmap of this paper is as follows. Section II introduces the background and motivation. Section III outlines the main approach of CLIC and overview its extensible system architecture. The GCN-based platform selection and the embedding-based operator encoding are described in Section IV. Section V gives the detail of the design and implementation of CLIC. Section VI evaluates the prototype system, and Section VII concludes the paper.

II. Background and Motivation

A. State-of-the-Art Cross-Platform Systems

A modern data analytics workflow usually consists of multiple data processing tasks such as data collection, data Extract-Transform-Load (ETL), machine learning model training, etc. [7]. Since a data processing platform is often developed for certain scenarios, involved tasks are usually beyond the specific functionalities of a single platform. Therefore, a workflow may need to be processed by multiple platforms collaboratively. For example, benefiting from its memory-based execution engine, Apache Spark has powerful distributed data processing capabilities and is commonly used in data ETL. But to model and analyze data with machine learning, data scientists tend to choose specialized frameworks like Tensorflow for the much richer integrated algorithms. In addition to special functionalities, tasks are also deployed on different platforms for higher performance. A platform's internal may have significantly different execution and communication models, making them more suitable for specific tasks [11].

A spectrum of cross-platform systems have been developed to facilitate the development and deployment of cross-platform workflows [1], [5], [6], [9], [10]. Those systems internally federate multiple data processing platforms for handling corresponding tasks. They block the platform details by providing users a set of platform-agnostic interfaces. By automatically selecting platforms for a workflow, users only need to focus on data analytics, without caring about underlying platform details.

Platform selection methods in current systems are mainly divided into three classes, i.e., rule-based, cost-based, and machine learning (ML)-based. 1) *Rule-based* systems like Musketeer typically design a series of decision rules to determine the platform of each task in the workflow. The approach not only requires large amounts of decision rules, but also has to take care of their complex dependencies. 2) *Cost-based* systems like Rheem [6] and RHEEMix [12] use cost models to estimate the costs of a task on different platforms, and then select the best platforms for the workflow. It requires each operator in a task to describe itself with properties like cardinality, computation complexity, or even the average CPU-cycles-per-run. As those properties need to be customized according to different environments, users have to take substantial efforts to fine-tune those properties to make the model work when being deployed [10]. Both rule-based and cost-based approaches demonstrate limited extensibility for the requirement to either add new rules or build new cost models when integrating an operator. 3) *ML-based* methods [10], [13], [14] are proposed to help select platforms to address issues in cost-based approaches. Specifically, by encoding a workflow as a one-hot encoded feature vector, Robopt [10] enumerates possible platform combinations of operators in terms of vectors. Then vectors are fed into the ML model to estimate their costs. For a workflow with n logical operators, each of which has k execution platforms, there would be a total of n^k possible execution plans. With an ML-based pruning mechanism, Robopt alleviates the enumeration overhead by reducing unnecessary calculations. However, ML-based methods have limitations when being applied in production systems.

B. Motivation

1) System Extensibility: Modern data analytics systems generally need to handle new data sources with different formats and train new machine learning models to gain insights. Through analysis and evaluation, we find that the system design of Rheem and the machine learning techniques in Robopt limit the system extensibility of integrating new platforms and operators. First, Robopt uses one-hot encoding to encode workflows makes it hard to extend new operators. As the input to models, each bit in an one-hot encoding vector represents a category, where a bit is set as 1 if the data fits the corresponding category. Therefore, the dimension of the vector space equals to the number of operators, which should be increased with a new operator. Fig. 1 illustrates this situation when integrating a new *SIN* operator(represented by the red dashed line) where all the original vectors' dimensions also grow by one. However, the machine learning model trained in the previous vector space can't take a vector with a different dimension as input. As a result, the model has to be retrained. Second, the monolithic architecture of Rheem (Robopt) binds backend platforms and the system in the same run time environment. In workflow execution, the system executes operators by directly calling its backend platform through the driver and passes data as in-memory objects. The tightly coupled system architecture makes it costly to add new platforms. Consequently,

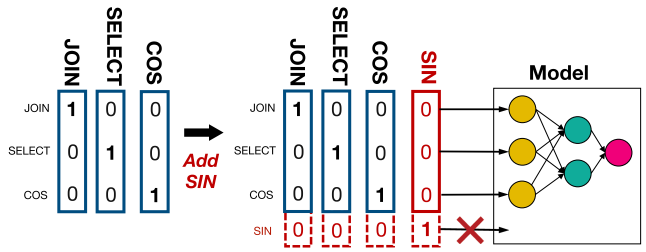


Fig. 1. One-hot encoding limits system extensibility.

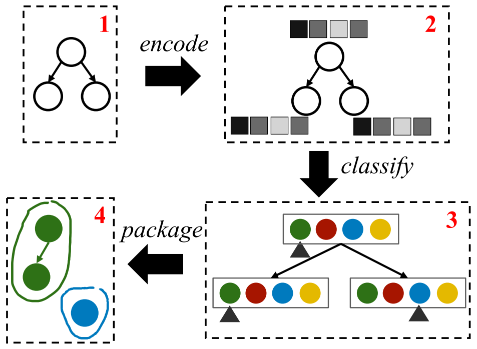


Fig. 2. The procedure of GCN-based platform selection.

when integrating a new platform, the whole system must be compiled, tested, and deployed each time.

2) Efficiency of Platform Selection: Existing approaches enumerate large number of possible platform combinations for a workflow [10]. Then it estimates the execution time to select the deployment plan with the lowest execution time. Even with optimizations such as pruning, this procedure incurs high overhead. In our experiments, for a cross-platform system that has 80 operators and 5 platforms, the average latency with the approach takes more than one second. Specifically, when processing small data sets, the cost of platform selection may become similar to the workflow execution time. To make things worse, the latency grows with an increasing number of operators and platforms [10]. As a result, the incurred overhead may offset the benefits from platform selection, which can even degrade the overall performance. Moreover, limited by the flat vectors, Robopt encodes the workflow by the number of special components, such as loops and junctures, and the one-hot vectors of its operators. However, since workflows usually have complex topological structures and different operator dependencies, this encoding method may fail to the characteristics of a workflow.

In summary, practical cross-platform data analytics demand an extensible system architecture and an efficient platform selection method. This motivates us to rethink the design and implementation of a cross-platform data analytics system.

III. System Overview

To address the above challenges, we propose a highly-extensible cross-platform computing system, namely CLIC. In this section, we introduce the main approach adopted by CLIC and overview its architecture.

A. GCN-Based Platform Selection

Different than previous approaches that estimate the costs of possible execution plans, we model the platform selection process as an end-to-end node classification problem. CLIC takes all platforms as categories and the best platform of an operator as its class. Then the platform selection is abstracted as determining the class of each node in a graph, which is a typical node classification problem [15]. As an effective approach for node classification [14], [16], Graph Convolutional Network (GCN) model is adopted to enhance the accuracy and efficiency in platform selection. GCN is a graph neural network that directly takes the topological structure as input and extracts relationships between nodes, thus it has the advantage on platform selection for a structured workflow than basic ML models. Besides, GCN embeds the search space into an euclidean space where the optimal points, i.e., the best platforms are computed directly in real-time, which is much more efficient than the traditional search.

GCN effectively utilizes the workflow topology information which involves three key factors that influence platform selection. 1) There can be data conversion between platforms, where a data conversion operator needs to be added in the physical plan. This would influence the platform selection between two adjacent operators for the incurred overhead. 2) There are data serialization and deserialization overhead when data being transferred between platforms. For instance, to pass data from Spark to PyTorch, data in RDD of Spark needs to be serialized into a common format before being read by PyTorch. 3) The topology information such as juncture influences platform selection. For a juncture (denoted as A), such as the *Join* operator, the outputs of its previous operators (denoted as B and C) become the inputs of operator A. This relationship is usually important for platform selection because B and C may run faster on different platforms. If B $\rightarrow$ A runs faster on one platform but C $\rightarrow$ A runs faster on another, it is hard to select the optimal platforms for the three operators. Therefore, the topology information is important for improving the accuracy of platform selection. Previous work such as Robopt [10] has also demonstrated the importance of topology information.

Together with the GCN model, CLIC improves the system extensibility with an embedding-based operator encoding technique. It encodes an operator into a dense and low-dimensional vector, called operator embedding, whose dimension is predefined and fixed. Because the length of the vector is not bound with the number of operators, integrating a new operator no longer needs to retrain the model. As the basis for the GCN model, the two main techniques significantly enhance the extensibility in CLIC so that new operators and platforms can be easily integrated.

Fig. 2 shows the major states in platform selection. The first state is the initial workflow where the vertex denotes the operator, and the edge denotes the data dependencies between operators. The vectorization process encodes each operator to their corresponding feature vector and generates the second state

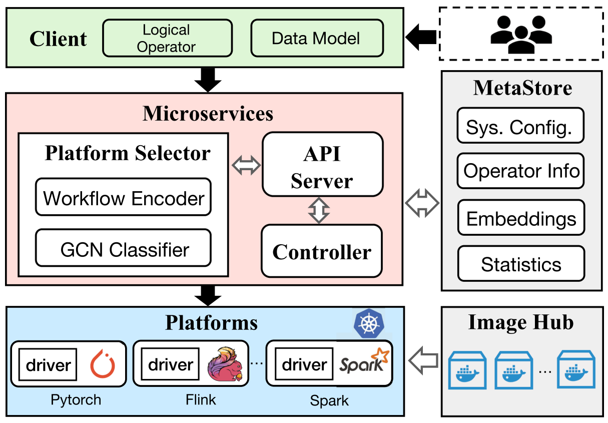


Fig. 3. The system architecture of CLIC.

(see Section IV). After that, GCN is utilized to classify operators. During the classification process, the workflow's intermediate state is shown as the third state where all available choices of an operator are indicated by colored circles, where one color represents a platform. A possible classification result is denoted by the triangles where the top and the bottom left operators are classified to the green platform and the bottom right operator to the blue platform. Finally, adjacent operators that are classified to the same platform are packaged as a task. The example in the figure generates two tasks as the output of the forth state, which runs on two different platforms. Those tasks are then submitted to the corresponding platforms for execution.

B. The System Architecture of CLIC

Fig. 3 demonstrates the system architecture of CLIC. CLIC aims at achieving high extensibility by decoupling the core functionalities such as platform selection and job scheduling from the underlying platforms. In CLIC, operators of each platform are packaged as images and stored in the *Image Hub* while their corresponding information are stored in a database called *MetaStore*. MetaStore records information like available platforms of each operator, operator embedding and runtime statistics of workflows. With this dynamically maintained information, only the platform image and the corresponding information in the MetaStore needs to be updated when integrating new operators or platforms. Since the core of CLIC stays unaffected, the integration can be flexibly performed at run time. Overall, the decoupled architecture of CLIC aims at enhancing its extensibilty so that the system can dynamically incorporate large number of new operators.

CLIC supports the full life cycle of a workflow including building, packaging, executing, and progress monitoring. A *client* provides users with various logical operators and data models for building data processing, machine learning, and graph computing workflows. A workflow is organized in the form of a logical Directed Acyclic Graph (DAG) and submitted for execution. The core functionalities of CLIC are deployed as separate microservices, including *API Server*, *Controller* and *Platform Selector*. API Server receives the workflow submitted from a client and hands it over to Platform Selector to generate the execution plan. In platform selection, operators in the workflow are first encoded as feature vectors, and then the GCN Classifier assigns each operator with a platform. In this process, properties including system configurations, operator's properties and embeddings are retrieved from MetaStore to form the feature vectors. Note that the GCN-based platform selection can be integrated into any cross-platform systems as a separate module. With the generated execution plan, Controller instantiates the platform instances by pulling their images from Image Hub. The platform image is instantiated in the Kubernetes cluster as containers. There is a driver running as a process in a platform instance. The driver receives a task from Controller and interprets the included operators into the platform's native APIs or implementations for execution. After submission, Controller monitors tasks and updates task states and statistics in MetaStore. CLIC can run on both bare-metal machines and a cluster with virtual machines on the cloud. By initializing virtual machines with a Kubernetes environment, CLIC can directly deploy tasks for execution.

IV. GCN-Based Platform Selection

To capture the workflow topology while enhancing its efficiency, we take platform selection as a node classification problem in a graph and use GCN model to solve it. Together with the model, we propose a novel embedding-based operator encoding method to enhance system extensibility. In this section, we first describe the embedding-based operator encoding method and then give details of how CLIC uses the GCN model for platform selection.

A. Embedding-Based Operator Encoding

The feature vector of an operator is provided to the GCN model for classification. The structure of the feature vector needs to cover the key factors that affect platform selection. Besides that, the vector should be as dense as possible so that it is conductive for model training. We mainly consider the following important influential factors, i.e., operator type, data volume, and hardware setups. We append the factors together to form the final feature vector as shown in Fig. 6. Below we narrate the encoding method of each factor separately.

1) Operator Type: Different operators need to be distinguished in a workflow for accurate platform selection [10], [14]. Different from using one-hot encoding whose dimension is strongly related with the number of operators, we introduce an embedding-based method to generate fixed length vectors. The resulting vector is called as operator embedding.

In mathematics, an embedding is a function $f(X) \rightarrow Y$ that maps a data point X in one space to point Y in another space. This is an important preprocessing technique in natural language processing, where word embedding is used for representing words in text analysis. Word embedding is typically a real-valued low-dimensional vector that encodes the meanings and relationships of words. The feature of embedding is that words that are closer (cosine distance) in the vector space are expected to have

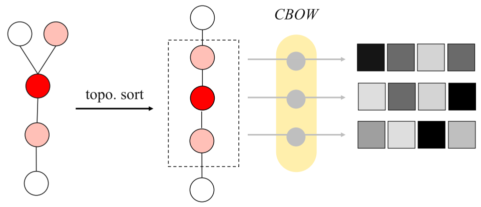


Fig. 4. Generating embeddings for operators.

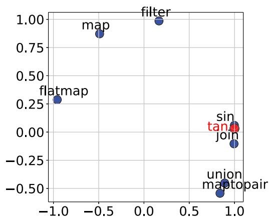


Fig. 5. Visualization of the embedding-based encoding.

similar semantic meanings. Those meanings and relationships can be captured from the corpus and be mapped to the vector space through neural networks. A representative algorithm for this is the CBOW [17], which takes words that appear in the same sentence to have higher relevance. CBOW works to maximize the joint probability of a word and its adjacent words in a sentence. Word embedding not only contains enough information but also has a much lower dimension comparing with one-hot encoding. Moreover, the advantage of embedding-based encoding is that it uses fixed vector length to represent variable number of items. Therefore, existing embeddings are still valid after integrating a new item. Because embedding can encode words in a low-dimensional dense vector, it is gradually being promoted for vectorizing other types of data like image, video, and graph.

In CLIC, we propose to treat operators in a workflow like words in an article, where an operator also has meanings and relationships with adjacent ones. The semantical meanings, for example, include operator's computing paradigm, number and type of input/output, etc. To generate the operator embedding, we share the same ideas in the word embedding that 1) allowing semantically similar operators to have closer (cosine) distance, and 2) maximizing the joint probability of an operator and its neighbour operators in a workflow. Therefore, we utilize the CBOW algorithm to generate operator embedding, which is illustrated in Fig. 4. It first topological sorts a workflow to get the linear operator sequence, and then uses a fixed-size sliding window to traverse the sequence. Each time an operator and a fixed number of its adjacent words are chosen as the input of the CBOW model. The model then outputs a vector for each operator, which is its generated embedding.

Following the above approach, we get an embedding for each operator. Fig. 5 shows the locations of the resulting embeddings where only the top-2 dimensions with the highest eigenvalue are used for visualization. As shown in the figure, the *SIN* operator is closer to *TAN* and far from *UNION*. This can be explained

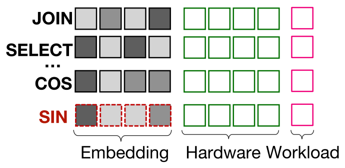


Fig. 6. Operator feature vector.

semantically because both *SIN* and *TAN* belong to the same computing paradigm and have the same number and type of inputs/outputs, etc. Although the real meanings behind are much more complex than that, we can find that the operator embedding in CLIC successfully portrays an operator.

One thing to note is that, one can generate the embedding of any operator as long as the workflows used for training contains it, even if this operator has not been integrated in CLIC yet. In other words, the operator embedding technique is a general encoding method that is independent to computing platforms. This feature allows CLIC to generate a rich operator embedding set in advance. After that, when CLIC integrates a new operator that needs the embedding, it can directly look up to the embedding set. There are two scenarios when getting the new operator's embedding:

1) Its embedding is contained in the embedding set. At this time, CLIC can retrieve the embedding directly.
2) The operator is not contained in the embedding set, i.e., the "Out-Of-Vocabulary (OOV)" problem. In this case, some new workflows that contains this new operator are required by CBOW to learn its embedding. The resulted embedding is still in the same space with the existing ones, which have the same dimensions. Therefore, the GCN model does not need to be re-trained.

As one of its core advantages, the dimension of operator embedding is fixed when integrating new operators. The grey segment in Fig. 6 illustrates the embedding(denoted by dashed line) of the newly integrated *SIN* operator that has a consistent dimension with the old ones, i.e., JOIN, SELECT and COS. Therefore, the encodings for operators no longer need to be re-trained, which significantly improves the extensibility of CLIC.

2) Hardware and Workload: Due to the differences on the design of communication model, execution engine, etc., the suitable platform under different hardware setups may also be different. Take the selection of machine learning frameworks as an example. Tensorflow adopts the parameter server architecture as its communication model, while PyTorch uses the all-reduce model. In general, the parameter server works better on a large number of unreliable and less powerful machines; All-reduce achieves higher performance for a small amount of fast devices being linked with high network bandwidth [18].

When measuring the effects of hardware setups, one should note that factors that can affect the platform performance do not necessarily affect the choice of the platform, because it may bring the same effects to all platforms. For instance, a faster hard drive brings performance improvements for all operators. In CLIC, we mainly consider the following four factors which correspond to the green segments in Fig. 6.

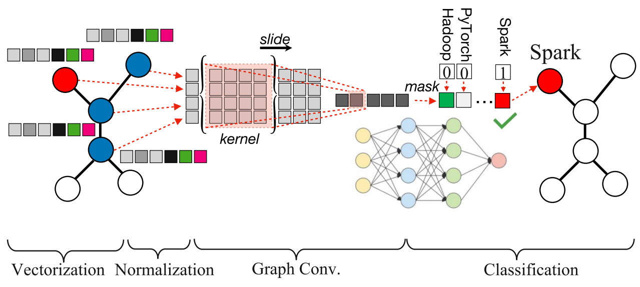


Fig. 7. Selecting platforms with GCN.

- *Cluster size:* This parameter influences the use of a local program or distributed platforms that are designed for different scales.
- *Memory size:* Different platforms have different usage of memory, this may influence the choice of memory-based or disk-based platforms, e.g., Spark versus Hadoop.
- *GPU:* The factor distinguishes CPU-based platforms like Spark from platforms that support GPU acceleration like Tensorflow.
- *Network bandwidth:* This may influence platforms by their communication models.

Workload also has a significant impact on platform selection. Specifically, as the size of a data set grows, the platform that can achieve highest performance for a given task gradually shifts from a local program to a distributed platform [6]. Therefore, we use the size of a data set to indicate the workload, which corresponds to the pink segment in Fig. 6. Moreover, other data set features are also considered in CLIC, such as average input tuple size.

B. GCN-Based Platform Selection

Except graph convolution network, other machine learning methods can't directly take topological data as the input but need to first stack the vectors of all nodes in the graph together to a two-dimensional matrix, i.e., the "design matrix". This prerequisite leaves out the structural relationship between nodes, which is the key for describing the computational process of a workflow. Therefore, those models may have intrinsic deficiencies like being difficult to converge or converged with poor accuracy.

The GCN model adopted by CLIC is a convolutional neural network that can be applied directly to graphs. The GCN model introduces a graph convolutional layer in the network architecture to aggregate topological information to represent nodes, making it suitable for the platform selection. The output of this layer is the vector of each operator that is embedded with structural information, called the graph representation, which can be further used for classification and regression tasks. Fig. 7 shows the classification process with GCN, which is mainly divided into four steps: vectorization, normalization, convolution, and classification.

1) *Vectorization:* As a deep learning model, GCN takes a vector as its input. The first step is vectorization which encodes each operator in the workflow into an embedding-based feature vector.
2) *Normalization:* Normalization is performed to align the irregular topology to a matrix. For each time, an operator is selected as the center operator (the red one) and its $k-1$ neighbor operators (the blue ones). After that, the feature vectors are aligned as a $k \times |V|$ matrix M(as shown in the middle grey matrix) where k is a hyper-parameter and $|V|$ is the dimension of feature vectors that equals to the sum of the dimensions of the operator embedding, hardware, and workload factors. Based on the workflow topology, we get a partial ordering over the nodes to build the matrix. Because large amounts of workflows are generated for model training, the training data cover different orders of operators. Therefore, platforms can be effectively predicted with a partial ordering of the input workflow nodes. This step repeats multiple times, each for an operator.
3) *Convolution:* Convolution utilizes the convolution kernel to extract localized features, which is the core step of GCN. The convolution kernel is a pre-defined $K \times K$ matrix that acts like a sliding window. The window slides and takes a $K \times K$ sub-matrix from M to perform the convolution. The resulted vector in this step is the graph representation of the center operator, which is used for classification.
4) *Classification:* This step classifies operators according to their graph representations. CLIC appends a dense layer in the end of GCN for classification. The output is the platform probability distributions of each operator, with the highest one being selected as the target platform. In case that it incorrectly classifies an operator to an unsupported platform, we place a mask on the probability distribution to filter such platforms. The masks are maintained in CLIC and can be retrieved directly at runtime.

In CLIC, the GCN model takes the initial platform-agnostic workflow as the input and outputs the optimal execution plan in which each operator is assigned with a platform as its label. Comparing with previous approaches that require the enumeration of large number of possible execution plans, this GCN-based inference procedure is not only orders of magnitude faster, but also more robust for a larger system with more operators.

C. Training GCN

The adoption of a machine learning or a deep learning model has the requirement of enough training data. In the GCN training in CLIC, the training data contains tuples in the term of $< W, H, V, L >$, where W is the workflow with a series of operator, H is the hardware setups of the current environment that are retrieved on startup, V is the data volume of the workflow that is dynamically given by users, and L is the label, i.e., the best platform of each operator in W. W is found by running all possible platform combinations and selecting the one with the lowest running time. Take the London crime analysis[1] as an example. W is the workflow consists of {Source, Map, Filter, Group, CountBy, Sort}, H is varied by controlling the cluster

[1]https://www.kaggle.com/LondonDataStore/london-crime

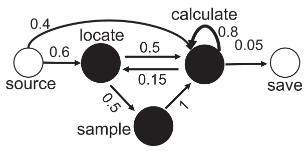


Fig. 8. An example Markov chain for generating training data.

configurations, and V is varied by gradually using 3% of the crime logs to 100%.

Because generating sufficient training data to train an effective GCN model is very time-consuming and almost impractical [19], we construct a Markov chain to mimic the pattern of real-world workflows. The state set and transition probabilities of a linear algebra workflow are shown in Fig. 8. To generate workflows similar to real-world workflows, the Markov chain model is trained with existing Big Data and machine learning data sets on Kaggle [20]. There are lots of user-built data analytics workflows on Kaggle Notebooks. We extract a set of the representative workflows and take them as the input to train the Markov chain model. The Markov chain model learns the relationships of the operators and the possibilities of transitions in the training workflow set. In this way, the model can generate new workflows based on real-world relationships between operators, where adjacent operators are always placed in a legal order.

We construct workflows with different lengths and topologies based on the Markov chain model. First, we adjust the transition probabilities to the end state to get workflows with variable lengths. For instance, with a higher transition probability to the end state, longer workflows can be generated. Second, we generate workflows with complex topology. The output workflow shape of the Markov chain model is a pipeline, i.e., a sequence of connected operators without branches. Workflows with branches or other topological shapes are constructed by connecting multiple pipelines using operators with multiple inputs or outputs as the hub. An operator with multiple input sources is called as an *input-hub*, and an operator with multiple outputs is called as an *output-hub*. For example, a *Join* operator is an input-hub that joins two database tables. The workflow shown in Fig. 7 is constructed by an input-hub operator, an output-hub operator, and four basic operators. We take generated workflows that have the same hub operator, remove the rest operators after an input-hub or before an output-hub of one workflow, and connect the workflows at the hub operator.

With generated workflows, the label, i.e., the platform for each operator, is determined following the approach in [19]. First, pruning the search space by rules, then running the rest combinations with various small workloads to find the best computing platforms for each workflow. When two adjacent operators are assigned with different platforms, the data conversion overhead is taken account if necessary. The platform performance on heavy workloads is generated by interpolation. At last, repeating the above steps on various hardware setups. In this way, workflows with arbitrary topology, workloads, and hardware setups are constructed as the training data for the GCN model.

V. System Design and Implementation

Besides the robust and efficient platform selection with the GCN model, CLIC offers platform-independent interfaces for workflow development and a set of techniques for enhancing system extensibility. In this section, we illustrate the system design and implementation details of CLIC.

A. Building a Workflow

The client of CLIC is provided with a series of platform-irrelevant logical operator and various data models to build the cross-platform workflow. A logical operator is an abstraction of an executable functional unit. It describes all information of the unit such as the function name, required data model, input parameters and return values, while the implementation details are excluded. A data model, including table, list, matrix, and graph, defines how data is organized and what is the basic processing unit. There can be significant differences between data models, and each model has its own suitable scenarios. For example, matrix and table are similar in data arrangement, while the former implicitly orders dimension columns and all its elements must have the same data type; On the other hand, columns in a table are not strictly ordered and can have different data types. Therefore, while matrix suits for machine learning, tables are generally served as the data model in relational data analytics. Graph explicitly links data points therefore is better at tasks with frequent relationship queries than a table. However, since a cross-platform workflow may involve multiple data models at the same time, it is hard to construct an unified data model for all scenarios due to the above differences.

CLIC bridges the gap between data models by implementing a series of model conversion operators and offers them to users as another type of logical operator. Two operators that require different data models are connected by a conversion operator. Take the simplest *matrix2table* conversion as an example. It transforms a matrix to a table by adding schema to the columns of the matrix, where the column name is set to the column serial number, and the data type is consistent with the original data type of cells. Note that there can be more than one conversion method between two models. For example, a matrix can be converted to a graph either by being regarded as an adjacency matrix or by more complex graph construction algorithms like HNSW [21].

We use sentiment classification as a case to demonstrate how CLIC works. The pseudo-code of a simple implementation is shown in Listing 1. It is a common natural language processing task that classifies documents based on the semantics. The data processing tasks mainly include data ETL (Extract, Transforming, Loading) and training of a classification model. As demonstrated in the listing, operators like *read_csv*, *union*, *select* and *train* represent the logical operators in the task. The operators take different types of input, from strings to a data set. Specifically, LSTM is used as the model for sentiment analysis, with SGD being set as the optimizer and cross-entropy as its loss function. At line 7, the operator toMatrix is used as the data model converter that outputs a matrix for machine learning with LSTM.

```
TableModel source1 = read_csv('BBC-News.csv');
TableModel source2 = read_csv('Hamlet.csv');
TableModel union_set = source1.union(source2);
ListModel sentences = union_set.select("sentence")
ListModel labels = union_set.select("label");
MatrixModel result =
    union_set.toMatrix(Conversions.Word2Vec);
result.train(labels, new LSTM(),
        Optimizer.SGD, Loss.CrossEntropy);
```

Listing 1. Pseudo-Code of Sentiment Classification in CLIC.

TABLE I
OPERATOR MAPPING TABLE

Logical Operator	Platform	Physical Operator
PCA	Tensorflow	tft.pca
	Spark	ml.feature.PCA
	PyTorch	torch.pca_lowrank
Map	Spark	map
	JavaStream	map
Word2Vec	PyTorch	nn.Embedding
	Spark ML	mllib.feature.Word2Vec
	Gensim	models.word2vec

B. Executing a Workflow

Operator Mapping: The workflow composed of logical operators is not executable, until each logical operator is mapped to its native platform APIs or implementations. Since a logical operator can have different implementations on multiple platforms, we wrap APIs of a logical operator to physical operators which have the same name, inputs, and outputs. Table I illustrates the mapping rules from logical operators to physical operators. These rules are implemented in the platform's configuration files and packaged into the docker image. The driver of each platform loads its mapping table and interprets the received logical operator to its native APIs or implementations. Besides that, the table is also used to generate the mask for filtering available platforms in the GCN model.

Adding Data Conversion Operator: Data conversion operators may be demanded between two adjacent operators of different platforms. Therefore, before platform selection, data conversion operators are not specified in a logical workflow. After the GCN model predicts platforms for operators in a workflow, two adjacent operators may run on different platforms. Then data conversion operators are added on the fly and deployed for execution if required. As discussed in Section IV-C, since the GCN model already considers the data conversion overhead in its training process, the added operator won't influence the effectiveness of platform selection.

Workflow Execution: CLIC controls the execution of a workflow and launches the included tasks according to their topological dependency. When a task completes execution, Controller records its completion in the MetaStore and launches its subsequent task and passes input variables. During the process, clients can query the execution states via API Server. To support flexible workflows with if-else and while loop clauses, CLIC also implements a series of flow controllers. A flow controller dynamically indicates the executing direction of the workflow that is determined according to the run time value. For a workflow with either loop clauses or if-else statements, a set of controllers is insert into the workflow to direct the execution flow.

Data Movement: In CLIC, the data movement between tasks on two platforms is performed by writing and reading intermediate results from a unified distributed storage engine. After a task writes its output as a file, the file name is passed as the input for the following task in the workflow. In CLIC, we adopt Alluxio [22] as the middle layer for the storage engine. Alluxio is compatible for multiple data access interfaces so that different platforms can access to the same storage engine. Moreover, with a memory caching layer, Alluxio can accelerate data movement by alleviating the overhead of disk operations.

C. Extending With New Operators and Platforms

CLIC is designed for flexibly integrating new operators and new platforms with low overhead. In CLIC, integrating a logical operator only needs to define its description file and register it into MetaStore. The description file is a JSON file that contains information like the operator name, parameter list for building the workflow, and operator properties like computing paradigm for constructing the operator embedding. To implement a logical operator on a specific platform, developers need to first implement the functionality using platform APIs and wrap it as the physical operator following the standard of CLIC. After that, developers need to add a new item in the mapping table and update the GCN mask for the corresponding logical operator that are all maintained in MetaStore. Finally, its docker image of the targeted platform needs to be rebuild and registered into Image Hub. The Image Hub can be a private repository maintained in Docker Hub and the registration is simply submitting the docker image to the repository.

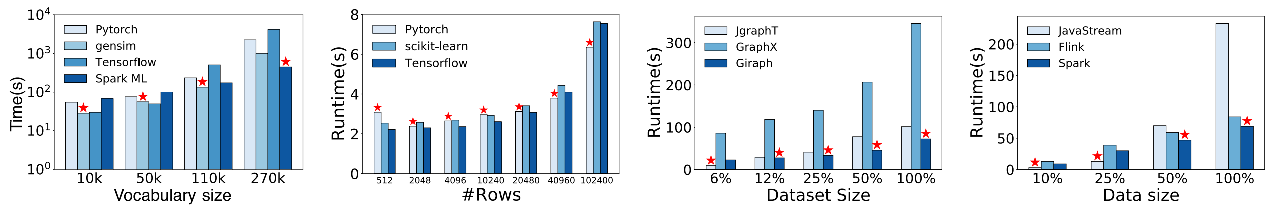
Integrating a new platform should first implement the included physical operators. Besides that, an essential component for a platform image is the driver program for interpreting logical operators and driving task execution. After the driver and operators are packaged along with the platform in a docker image, the related items in MetaStore and Image Hub are updated. Since MetaStore, Image Hub, and platforms are all decoupled and maintained independently, none of the above processes need to modify the core components of CLIC. Overall, operators and platforms in CLIC can be flexibly integrated at run time.

VI. EXPERIMENTAL EVALUATION

In this section, we conduct experiments to evaluate the accuracy and efficiency of the GCN model and validate CLIC's capability of enhancing workflow performance.

A. Experimental Setup

All experiments are conducted on a cluster of 7 nodes, where each node is equipped with two 2.3 GHz Intel Xeon Gold 5218 processors with 16 cores, four 32 GB DDR4



(a) Word2Vec (b) PCA (c) PageRank (d) WordCount

Fig. 9. Evaluation of platform selection for one operator.

RAM, 2 TB SSD and runs on 64-bit Ubuntu 20.0.1. The platforms deployed on the cluster include Java's Streaming library (JavaStream), Spark 2.4.5 (Spark), Spark ML 2.4.5 (Spark ML), GraphX 2.4.5 (GraphX), JgraphT 1.4.0 (JgraphT), GraphChi 0.2.2 (GraphChi), Giraph 1.3.9 (Giraph), PyTorch 1.7.1 (PyTorch), python gensim library (gensim), Tensorflow 2.9.1. The platform images and containers are managed by Kubernetes v1.20. Alluxio 2.4 is used for data transmission and HDFS 3.2.1 works as the file system under Alluxio.

With the markov chain model, we generate 2.5×10^4 workflows as the training data, where workflows have an average of around 40 operators. The data generation takes about 12 hours. Then the generated workflows are used as the training data to train the GCN model. The training process takes around a day on our server.

B. *Performance Evaluation of Basic Workflows*

Fig. 9 shows the performance of four representative operators i.e., Word2Vec, PCA, PageRank, and WordCount. In the figure, the platform chosen by the GCN model in CLIC is indicated by red stars.

In the evaluation of Word2Vec (Fig. 9(a)), we use the NLTK dataset containing the different sizes of corpora to represent different workloads. Among supported platforms, gensim achieves the highest performance when using the corpus of 10,000 vocabularies while Spark ML suffers from the distributed communication overheads. However, Spark ML gains a performance advantage when the corpus contains 270,000 vocabularies. This is because Spark ML is able to utilize multiple nodes for the enlarged dataset, where the communication overhead becomes negligible compared with the computation time. CLIC identifies the difference and selects the best platform in both situations.

Fig. 9(b) shows the accumulated time of 10 PCA runs. The dataset is the Hotel Booking Demand.[2] As shown in the figure, the GCN model may sometimes get suboptimal results for the PCA operator, which selects Pytorch other than Tensorflow for small dataset. Same with other machine learning models, the GCN model cannot always make the optimal prediction. Moreover, since huge amounts of data are demanded for training the GCN model, optimizations are adopted to make platform predictions for achieving optimal or near-optimal performance. In this case, because Pytorch, scikit-learn, and Tensorflow have negligible performance differences for the PCA operator, same feature vectors are used in model training to significantly reduce the amount of training data. Therefore, even GCN makes suboptimal predictions for such operators, the performance of the predicted platform is close to that of the optimal one.

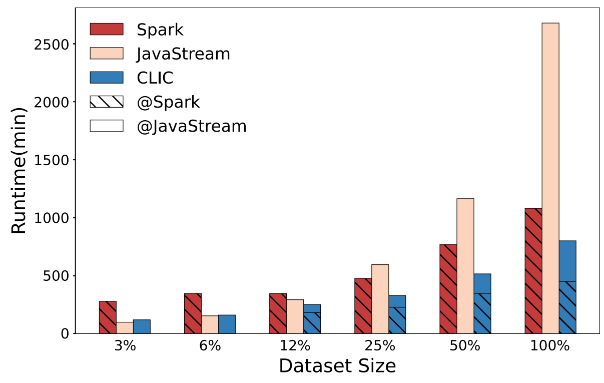


Fig. 10. Performance improvement of London crime analysis.

The data set in evaluating PageRank is the Twitter follower network.[3] As shown in Fig. 9(c), JgraphT achieves the highest performance when the graph contains less than 1.8 million vertexes while Giraph performs the best on the other cases for its higher efficiency in processing large graphs. Comparing with GraphX, the Giraph chosen by CLIC can achieve up to $5.1\times$ performance improvement with the entire dataset.

To evaluate the map-reduce operators, we encapsulate the WordCount as a coarse-grained operator. As shown in Fig. 9(d), the single-machine framework JavaStream achieves the highest when the corpus is small. Instead, Spark is $4.2\times$ faster than JavaStream when using the full dataset because it can be divided and computed locally on each node with little global synchronizing overheads. CLIC selects the optimal platforms for both the PageRank and WordCount operator.

The above experiments demonstrates that CLIC can make optimal or near-optimal predictions and achieve substantial performance enhancements for most cases.

C. *Performance Evaluation of a Big Data Workflow*

Fig. 10 compares the performance of a workflow that analyzes the London Crime dataset. The size of the dataset is 28 GB. The dataset describes the amounts of criminal reports by month,

[2]https://www.kaggle.com/jessemostipak/hotel-booking-demand

[3]https://snap.stanford.edu/data/twitter-2010.html

major/minor category, etc. We use percentages in the figure to denote different sizes of the dataset to evaluate platform prediction. For instance, 50% means the execution time on only 14 GB of data (28×0.5). The workflow mainly consists of four procedures: 1) reading data and transforming it to a specified format; 2) filtering reports to preserve only the ones with certain crime types, which will remove 90% of the data; 3) sorting and transforming reports to a human-readable format, and 4) grouping reports based on quarter.

The execution time for the workflow is shown in Fig. 10. For performance comparison, we also evaluate the performance on JavaStream and Spark. The observation is that when the data size is less than 6%, the computational efficiency of native JavaStream and CLIC is 2.8× higher than Spark. The reason is that Spark is set to use all available nodes in a cluster even when the dataset is small. Consequently, the communication overheads among distributed workers dominate the execution time. With the GCN model, CLIC effectively selects JavaStream for all operators in a small dataset. When the dataset size is larger than 12%, the amount of data to be processed in the first two procedures is too large that moving the computation from JavaStream to Spark can benefit from the acceleration of distributed computing. Therefore, step 1 and 2 are executed on Spark while step 3 and 4 are executed on Javatream for data set size higher than 12%. This is the reason that CLIC starts to outperform the native JavaStream. This benefit is most obvious when the dataset size reaches 100% where native JavaStream's performance is 2.4× lower than Spark and 3.3× lower than CLIC. The workflow demonstrates the effectiveness of cross-platform processing and the significant performance improvement it brought.

For dataset sizes of lower than 6%, CLIC is slightly slower than JavaStream. This is because, comparing with the direct execution of JavaStream, CLIC has three types of extra overhead. 1) *Platform selection:* It needs to lookup the embedding vectors for all operators in a workflow and uses the GCN model to predict platforms. 2) *Interpretation:* As discussed in Section V-B, there is a driver program on each platform that loads a mapping table and interprets the received logical operators to its native APIs or implementations. 3) *Deployment:* CLIC deploys tasks on Kubernetes, where docker containers need to be created and deployed with the images of corresponding platforms. This deployment process causes the main overhead that degrades the performance of CLIC. Therefore, comparing with running JavaStream directly, CLIC is slightly slower for small data sets although it chooses the same platform.

D. Performance Evaluation of Cross-Platform Workflows

We take two real-world workflows, i.e., sentiment classification and PageRank, to validate CLIC's ability to execute a workflow that contains multiple algebras. The topology of the two workflows are shown in Fig. 11, where sentiment classification has a tree topology and PageRank has a linear topology. We record the execution time of each stage when executing the workflow, including start session, data I/O, and computation.

In sentiment classification, it first reads corpora to generate Word2Vec embedding, then it reads and processes paragraphs

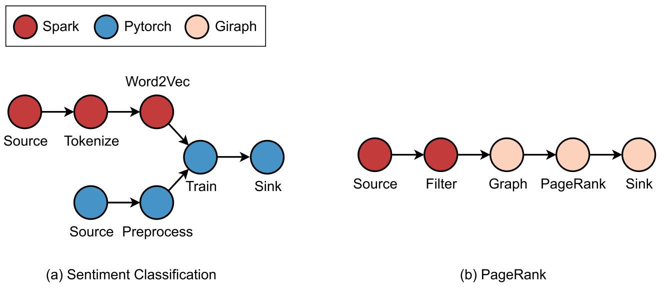
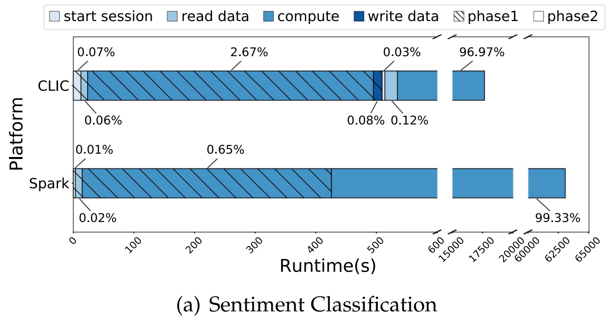


Fig. 11. Two workflows with different topologies.



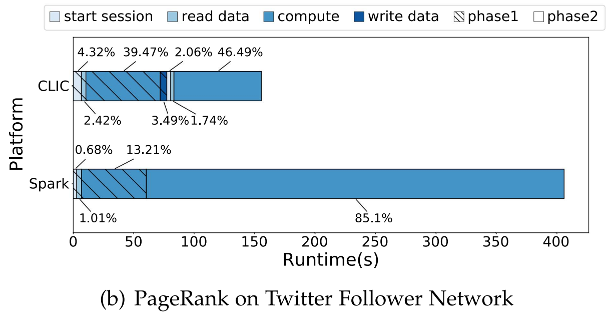


Fig. 12. Performance improvement of cross-platform workflows.

for classification, which is called as *phase1*. After that, it trains the deep learning model, which is called as *phase2*. The training data set is the Amazon Reviews[4] that consists of 3.4 million Amazon customer reviews (input text) and star ratings (output labels). Consider that SparkML currently only supports a few simple machine learning models, we replace the LSTM (Long Short-Term Memory) module with MLP (Multilayer Perceptron) for a fair comparison. As shown in Fig. 12(a), CLIC trains 3.6× faster than Spark. This is because CLIC migrates the model training process to PyTorch which can utilize GPUs for parallel training, while the CPU-based Spark cannot reap this benefit. Although the migration needs to read and write intermediate results across platforms, it can be seen from the figure that its overhead is far less than the benefit from GPU acceleration. Moreover, both platforms achieve 78% training accuracy, while CLIC can achieve an accuracy of 97% with the LSTM model.

The second workflow is to perform PageRank on the Twitter follower network.[5] As shown in Fig. 11(b), it consists of 1) extracting and filtering users that are followed by less than five

[4]https://www.kaggle.com/bittlingmayer/amazonreviews
[5]https://snap.stanford.edu/data/twitter-2010.html

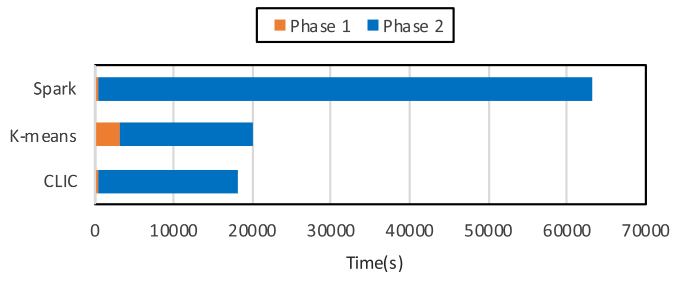


Fig. 13. Performance comparison with K-means-based platform selection.

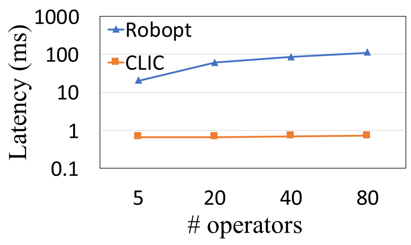
other users (batch processing), 2) constructing a graph, and 3) performing PageRank on the graph (graph computing). We call the first procedure as *phase1* and the last two procedures as *phase2*. With the GCN model, CLIC keeps the *phase1* in Spark and migrates the phase2 to Giraph. The running time comparison between Spark and CLIC is shown in Fig. 12(b), where Giraph significantly reduces the execution time of the second phase. Therefore, CLIC finally demonstrates 4.8× performance improvement.

When processing data on two platforms, intermediate results need to be written from the first one and read by the following one. In this example, Spark writes data into Alluxio and Giraph reads it. As shown in Fig. 12, the read and write overhead is trivial comparing with the execution time reduction, which takes around 0.23% and 7.29% of the overall execution time in the two workflows, respectively. Besides that, launching tasks as containers on the cloud takes 0.07% and 4.32% of the execution time, respectively. Considering the performance improvements from cross-platform computing, these extra overhead is negligible.

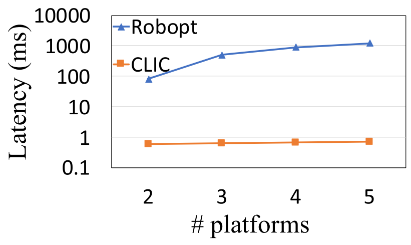
E. Comparison With ML-Based Platform Selection

To evaluate the effectiveness of the GCN model, we develop another machine learning approach that uses K-means to classify the node embeddings into a set of clusters. As shown in Fig. 5, with node embeddings, similar operators have smaller distances. Therefore, operators that perform similar operations are classified into the same cluster(s) with K-means. For each operator in a cluster, we take the most frequently assigned platform as its vote. Then the cluster is marked with the platform that has the highest number of votes. With a workflow of multiple operators, the embedding of each operator is classified in the K-means clusters to assign a platform. In this approach, each operator has fixed platform assignment on different workflows.

Taking sentiment classification as an example, Fig. 13 compares the performance improvement with the K-means-based platform selection and the GCN-based platform selection in CLIC. With K-means-based platform selection, machine learning operators including Word2Vec and model training are all assigned to PyTorch. It enhances the workflow performance by 3.1× because PyTorch can utilize GPUs to accelerate the model training process. With the GCN model for platform selection, CLIC assigns Word2Vec to Spark, which achieves higher performance with distributed batch processing. Evaluation results show that the GCN selected platforms can further enhance the



(a) 2 platforms



(b) 80 operators

Fig. 14. Evaluation of the robustness and efficiency of the GCN model.

performance by 11.7%, comparing with the K-means-based approach.

Approaches with fixed platform selection demonstrate limitations in many cases. As shown in Fig. 9, different dataset size may lead to different platforms. For instance, PageRank runs faster on Giraph on most cases while the JgraphT performs better for small data set. For WordCount, Spark performs better for larger dataset while JavaStream achieves higher performance for small dataset. These knowledges can be learned from workflow running instances with different topology, dataset, and hardware. This is the reason that GCN-based machine learning can perform better than other simple approaches that have fixed platform assignments.

F. The Efficiency of GCN-Based Platform Selection

We evaluate the runtime efficiency of the GCN-based platform selection and make a comparison with that of Robopt. The latency here refers to the model inference time of a workflow. We first evaluate the performance impact from the number of operators, whose results are shown in Fig. 14(a). As shown in the figure, the prediction time of CLIC is only around 1 ms with 5 to 80 operators, and the latency is three orders of magnitude lower than that of Robopt. This is because the GCN embeds the search space to an euclidean space, which makes the optimal platform combination can be directly computed with the model, instead of exhaustively searching in a large space. Moreover, the inference time can be further optimized with GPU acceleration.

In Fig. 14(b), we evaluate the performance impact with different number of platforms. The latency of CLIC is three to four orders of magnitude lower that Robopt. More importantly, it barely increases as the number of platform grows. The reason behind is that when considering more platforms, the search space in Robopt grows exponentially therefore taking more time to search. Differently, CLIC only increases the dimension of the GCN output with more platforms, which has negligible influences on the inference time. Overall, CLIC demonstrates good extensibility for integrating more operators and platforms.

VII. Conclusion

We have made a strong case by designing and implementing CLIC, a highly extensible system for cross-platform data analytics. Optimized for data analysis applications, CLIC supports platform-independent workflow development, agile operator and platform integration, and modular execution environment management. As the key for enhancing overall performance, CLIC efficiently selects platforms for operators in a workflow by adopting a GCN model and embedding-based operator encoding. Experimental results show that CLIC is able to select appropriate platforms for a data analysis job to enhance its overall performance. Overall, CLIC demonstrates a promising holistic solution for effectively leveraging multiple platforms in data analytics. In the future, we plan to extend CLIC to support stream processing platforms such as Flink, which demands a different computing paradigm than the platforms with batch processing.

References

[1] M. Zaharia et al., "Apache spark: A unified engine for Big Data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[2] A. Giraph, "Welcome to Apache Giraph!," 2011. [Online]. Available: https://giraph.apache.org
[3] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/
[4] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[5] V. Gadepally et al., "The BigDAWG polystore system and architecture," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2016, pp. 1–6.
[6] D. Agrawal et al., "Rheem: Enabling multi-platform task execution," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2069–2072.
[7] E. Bisong, "Kubeflow and kubeflow pipelines," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Berlin, Germany: Springer, 2019, pp. 671–685.
[8] A. Beam, "Apache beam: An advanced unified programming model," 2017. [Online]. Available: https://beam.apache.org/
[9] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: All for one, one for all in data processing systems," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.
[10] Z. Kaoudi, J.-A. Quiané-Ruiz, B. Contreras-Rojas, R. Pardo-Meza, A. Troudi, and S. Chawla, "ML-based cross-platform query optimization," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1489–1500.
[11] M. Anderson et al., "Bridging the gap between HPC and Big Data frameworks," in *Proc. VLDB Endowment*, vol. 10, no. 8, pp. 901–912, 2017.
[12] S. Kruse, Z. Kaoudi, B. Contreras, S. Chawla, F. Naumann, and J.-A. Quiane-Ruiz, "Rheemix in the data jungle: A cost-based optimizer for cross-platform systems," *VLDB J.*, vol. 29, pp. 1287–1310, 2020.
[13] R. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," 2019, *arXiv:1902.00132*.
[14] X. Zhou, J. Sun, G. Li, and J. Feng, "Query performance prediction for concurrent queries using graph embedding," in *Proc. VLDB Endowment*, vol. 13, no. 9, pp. 1416–1428, 2020.
[15] S. Bhagat, G. Cormode, and S. Muthukrishnan, "Node classification in social networks," in *Social Network Data Analytics*. Berlin, Germany: Springer, 2011, pp. 115–148.
[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
[18] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *Proc. IEEE/ACM Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2018, pp. 33–44.
[19] F. Ventura, Z. Kaoudi, J. A. Quiané-Ruiz, and V. Markl, "Expand your training limits! Generating training data for ML-based data management," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 1865–1878.
[20] Kaggle: Your machine learning and data science community. [Online]. Available: https://www.kaggle.com/
[21] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, Apr. 2020.
[22] H. Li, "Alluxio: A virtual distributed file system," Ph.D. dissertation, EECS Dept., Univ. California, Berkeley, CA, USA, May 2018. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html

Qixiang Chen received the MS degree in computer science from Fudan University, China, in 2022. Currently, he is working with Alibaba. His research interests include cross-platform computing and high-performance computing.

Zhijun Chen received the BS degree in communication engineering from Shanghai University, Shanghai, China, in 2020. He is currently working toward the graduate degree with Fudan University. His research interests include computer architecture, vector database, and cloud computing.

Kai Zhang received the PhD degree from the University of Science and Technology of China, in 2016. He is an associate professor with the School of Computer Science and Engineering, Fudan University. He was a research fellow with the National University of Singapore and a visiting scholar with The Ohio State University. His research interests are mainly in the fields of parallel and distributed computing and database systems.

X. Sean Wang (Senior Member, IEEE) received the PhD degree in computer science from the University of Southern California, CA, USA, in 1992. He is a distinguished professor with the School of Compute Science, Fudan University, Shanghai, China. Before joining Fudan University, in 2011, he was the Dorothean chair professor in computer science with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.