A Resource-Aware Deep Cost Model for Big Data Query Processing

Yan Li[1], Liwei Wang[1*], Sheng Wang[1], Yuan Sun[3], Zhiyong Peng[1,2*]
[1]School of Computer Science, Wuhan University [2]Big Data Institute, Wuhan University
[3]School of Computing and Information Systems, The University of Melbourne
[1]{liyana, liwei.wang, swangcs, peng}@whu.edu.cn, [3]yuan.sun@unimelb.edu.au

***Abstract*—The efficiency of query processing is highly affected by execution plans and allocated resources in the Spark SQL big data processing engine. However, the cost models for Spark SQL are still based on hand-crafted rules. The learning-based cost models have been proposed for relational databases, but it does not consider the effect of the available resources. To address this, we propose a resource-aware deep learning model that can automatically predict the execution time of query plans based on historical data. To train our model, we embed the query execution plans based on the query plan tree and extract features from the allocated resources. A deep learning model with adaptive attention mechanisms is then trained to predict the execution time of query plans. The experiments show that our deep cost model can achieve higher accuracy in predicting the execution time of query plans compared to traditional rule-based methods and relational database learning-based optimizers.**

I. Introduction

With the rapid development of the Internet, the volume of data is growing explosively, especially in recent years with the broader use of social networks [1], the Internet of Things [2], and cloud computing [3]. Traditional data storage and analysis technologies have become challenging to meet the needs of users. Therefore, various big data processing systems [4]–[6] have emerged to provide new options for current data queries. Most of them support declarative Structured Query Language (SQL) intending to represent queries more concisely. Compared to relational databases (Oracle [7], MySQL [8], etc.), where usage scenarios have largely solidified, the big data processing systems are more flexible in their functionality.

Spark [4] is one of the popular computational engines for large-scale data processing. Spark SQL [9] is the Spark module for structured data processing, and it is a well-known open-source SQL engine.[1] However, Spark SQL still relies on rule-based optimizers, and the research into cost models that can improve the query performance of Spark SQL is relatively rare [10]. The query optimizer of Spark SQL is a critical component in making queries perform well, designed to generate the best query execution plans. Optimized query execution plans can significantly improve the query execution efficiency of systems. For example, Fig. 1 illustrates the impact of cost models on query performance in Spark SQL, where we compare the performance of the default cost model and our optimized cost model (described later) on twenty queries. We can see that our tuned cost model can significantly reduce the execution time of each query and greatly improve the query performance. Therefore, it is essential to provide accurate cost models for Spark SQL.

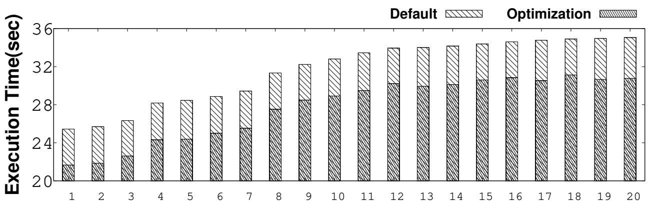


Fig. 1. A performance comparison between the default and our optimized cost models on twenty randomly selected SQL queries over the IMDB dataset on Spark SQL.

The study of cost models in relational databases [11] is well-developed and usually designed based on the experience and knowledge of experts. These methods frequently lead to the generation of sub-optimal plans due to incorrect cardinality and cost estimation [12]. It is because they rely on inaccurate statistics and over-simplified assumptions. Recently, the database community has begun to explore the use of machine learning technologies to improve query performance [13]–[17]. Most of them use learning-based models instead of complex heuristically-driven models. For instance, TLSTM [17] used a tree-structure based learning framework that combines cardinality and costs to improve cost estimation. DRL [16] drew on the experience of cost models to automatically explore the space of possible query plans using deep reinforcement learning.

However, the above cost models are not suitable for Spark SQL. Firstly, in relational databases, it is always assumed that the modified cardinalities will automatically correct the cost estimate, and the cost model is not as crucial as the cardinality estimate [12]. But for big data processing engines, the error of the cost models is still large even with real-time cardinality [18]. Besides, existing cost models consider queries to run on a fixed set of resources. In contrast, Spark SQL runs in a cloud computing environment where multiple applications share resources. Users' queries are running in a scenario where resources are constantly changing and adding certain resources will not necessarily improve the query performance.

[1]In the rest of the paper, we will focus on Spark SQL due to its popularity, but our proposed model is general and can be applied to other big data processing frameworks which support SQL.

*Corresponding authors

For example, we observe in our experiments that allocating more resources to an executor may even increase the execution time of a query (See Sec. III). Therefore, resource is an important factor to consider when estimating the cost of a query execution plan in big data systems. Although Baldacci and Golfarelli [10] proposed the first cost model for Spark SQL, it is still designed based on the knowledge and experience of experts and requires significant human efforts to tune. Furthermore, their hand-crafted rules are too limited to capture the complex patterns of the effects of resources on the performance of query plans. Hence, there is a need to design an automatic and learnable cost model, in order to obtain a proper combination of real-time resources and query execution plans. In contrast to the traditional cost models, learnable cost models can easily be updated regularly and adapted to different clusters.

To address the above issues, we first analyze the impact of resource changes on query performance to show that resources are an essential factor to the cost of a query execution plan. After that, we propose a Resource-Aware Attentional LSTM (RAAL) model to accurately predict the cost of query execution plans to improve the query performance. We develop a novel encoding method that can efficiently extract features of query execution plans. Then, to better extract the semantics of the execution statements, we use LSTM networks to model the query execution plans. Finally, we apply an adaptive attention mechanism that dynamically learns the relationships between nodes in query execution plans and actively tracks the resources available to run these execution plans.

In summary, our key contributions are as follows:

- We conduct a detailed analysis of the impact of resources on the cost of execution plans in Spark SQL. Our analysis shows that resources strongly influence the cost of execution plans, and the optimal query execution plan for a query varies with resources without an obvious pattern. (See Sec. III)
- We propose a novel encoding method to represent the embedding vector of query execution plans. We extract features of execution plan trees using a structure embedding consisting of the out-degree and in-degree information of each node and a node-semantic embedding to represent the operational semantics of each node. (See Sec. IV-C)
- We propose an LSTM-based cost model RAAL. RAAL extracts the ordered connection between nodes in query execution plans, and senses changes in the executing resources in combination with an adaptive attention mechanism. Our RAAL model is built on both the node connection and resource features to predict the execution time of query execution plans. (See Sec. IV-D)
- We conduct extensive experiments using different datasets to verify the impact of resources on the cost of query execution plans in clusters. The results show that our proposed model outperforms the state-of-the-art models in terms of prediction accuracy. (See Sec. V)

II. Related Work

A. *Spark and Spark SQL*

Apache Spark [4] is the dominant big data framework, with more efficient and faster computing power. In contrast to other big data systems such as Hadoop and Storm, Spark provides a comprehensive, unified framework for managing a wide range of datasets and data sources with different properties.

Spark Running Mechanism. Resilient Distributed Dataset (RDD) [19] is an abstraction of distributed memory that provides a highly shared memory model. Spark organizes data in RDDs to logically partition the data. When each application is submitted to run in Spark, it requests a set of executor resources to run the application independently. Spark's resource unit is the executor, a process on the worker node where the application runs. Each application has its own set of executors. Theoretically, the executor's memory, the number of executor cores, and the number of executors determine the resources available to run the application.

Spark has two types of resource allocation mechanisms: static and dynamic. The static resource allocation means that the application, throughout its lifecycle, occupies the allocated resources. This is in contrast to dynamic resource allocation, under which the resources allocated to an application are released back to the cluster when they become free. In either resource allocation approach, the resource features captured by our cost model are the initial resources allocated to the application. If the resource changes during the query execution, we will continue executing the chosen plan, because it may cost more to terminate the current execution plan and switch to another.

Spark SQL. The core component of Spark SQL is *Catalyst*, responsible for converting declarative SQL queries into an optimized query plan. Catalyst processes a SQL query by compiling and parsing, generating a logical plan, optimizing the logical plan, and generating the physical plan. A logical plan usually develops one or more physical plans, from which we need to choose the best one depending on cost models. Catalyst has utilized statistical data and a simple cost model since version 2.3.0, mainly by developing some calculation rules to estimate the cost of each operator to determine the join order [10]. After the physical plan is selected, Spark SQL translates the query plan into a DAG graph for execution on the Spark Common Engine.

B. *Query Optimization*

Traditional Cost Models. Traditional database cost models [20], [21] calculate a weighted sum of CPU cost and I/O cost. These factors are heavily dependent on database statistics information. Additionally, the weighting of each element requires human tuning. Most of the early studies focused on efficient cost functions for database operations. Manegold et al. [20] concerned about the cost of memory access to database operators, developing a generic technique for creating exact cost functions for database operations. Theodoridis et al. [21]

proposed to estimate the cost of join queries in spatial databases based on an R-tree.

Learning-based Query Optimization for Relational Databases. Traditional query optimizers rely on heuristic rules and require much effort to tune databases. The database community starts to build learning-based query optimizers [22]. Several studies [23], [24] have attempted to estimate the cardinality more precisely. Liu et al. [23] proposed to use the neural network architecture to learn a function that could estimate the cardinality of only containing relational predicates. MSCN [24] was designed to capture the connected cross-correlation of data using a multi-layer convolution neural network. These studies argue that an accurate cardinality estimation is more important than a precise cost model.

Deep learning techniques for query optimization can be broadly divided into two categories: 1) generating query execution plans [15], [16], [25], [26]; and 2) predicting the cost of query execution plans to select the best one [17], [27]–[29]. DQ [25] used reinforcement learning to optimize the join queries, guiding the search space in a data-driven manner to learn optimized join search strategies. Ortiz et al. [26] discussed how to use state representation to improve reinforcement learning-based query optimization. Neo [15] employed a deep neural network to generate query execution plans, which learns from existing optimizers to form a new type of learning-based query optimizer. In contrast, the other category of methods applied deep learning techniques to predict the execution cost of queries. Akdere et al. [27] and Marcus et al. [28] focused on deep learning-based query performance prediction for relational databases. TLSTM [17] combined cost and cardinality estimation to improve prediction accuracy, which is currently the state-of-the-art method. However, all of these efforts are improvements and refinements to relational database query optimization problems.

Query Optimization for Big Data Processing. Due to the complexity of Spark SQL running in multiple computing nodes, developing accurate cost models for Spark SQL has become more complex than that for relational databases. RIOS [30] is a runtime integrated optimizer for Spark SQL. RIOS collected all statistics related to a given query at runtime to determine the optimized join order. It focused on the impact of cardinalities on generating query plans. Baldacci and Golfarelli [10] designed a cost estimation function for Spark SQL, which was the first cost model for the Spark computing paradigm.

There are also studies [31], [32] that aim to match the best resources for a given query execution plan. They tend to run some sample data through the model to find the optimal resource allocation. However, even if the optimal resources required to run a query plan are known, the resources allocated to the query plan in a cloud environment do not necessarily match the optimal resources, thus increasing the blocking time of the task. Iorgulescu et al. [33] applied memory elasticity to cluster scheduling to reduce task wait time. C_{LEO} [18] was an initial exploration of learning-based cost models for SCOPE [34], the big data processing system of Microsoft, and it integrated a large number of individual cost models (micro-model). Viswanathan et al. [35] proposed to select both query plan and resource allocation at the same time. Microlearner [36] is a learning query optimizer based on the idea of micro-models to estimate cardinality and cost, etc. In contrast, our model is an end-to-end approach, aiming to predict the cost of execution plans with different resources in Spark SQL.

TABLE I
RESOURCE CONFIGURATIONS

Resources	Description
Node	Number of nodes composing the cluster, which is the number physical CPUs that make up the cluster.
Core	Number of cores on each node, where a node can contain more than one core.
Executor	The number of processes started on the Worker Node for an Application. Each node can have one or more executors.
E-Core	Number of cores for each executor. That is the number of concurrent threads that can be used per executor.
E-Memory	Maximum memory per Executor.
N-throughput	Network throughput between nodes.
D-throughput	Disk read/write throughput.

III. THE IMPACT OF RESOURCES

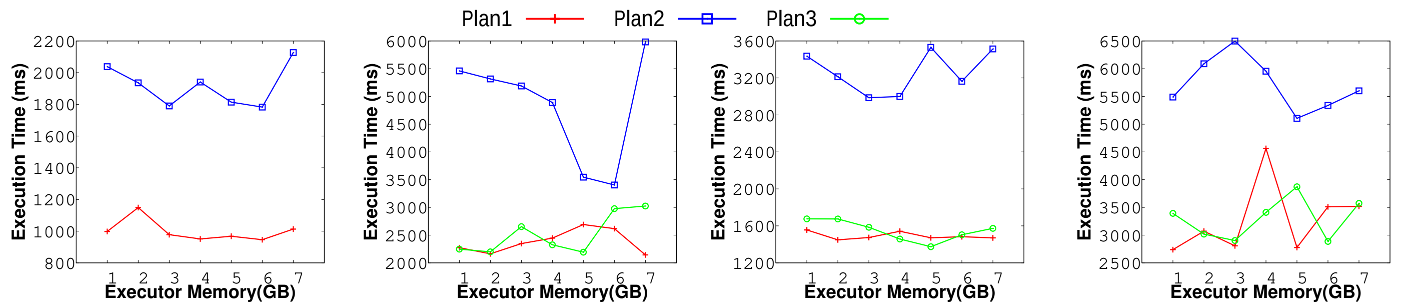
Spark SQL runs in a cloud, where multiple users or applications share the resources of a cluster. Resources affect the execution time of queries and play a critical role in query plan cost estimation. Intuitively, the amount of resources available are negatively correlated with the query cost. However, we observed that this is not always true in experiments. In the following, we investigate the impact of resources on the cost of query execution plans.

We conducted our experiments in a cloud environment, and the cluster is configured as shown in Table III (see Section V-A). Table I shows the resource configurations associated with Spark SQL.[2] We use a real-world dataset IMDB [12], which is the Join Order Benchmark extension. It is a 7.2GB dataset including 22 tables from the Internet Movie Data Base (IMDB). The final cost of each query execution plan is averaged over three runs to alleviate randomness. As memory is usually the performance bottleneck, we will examine the impact of resources on query performance, using executor memory as an example.

Sort Merge Join (SMJ) and Broadcast Hash Join (BHJ) are two typical implementations of joins in Spark SQL. To investigate the impact of resources on different query operations, we analyze the following four representative queries.

1) Single-table query, the query execution plan not involving join operations:
```
SELECT COUNT(*) FROM movie_keyword
     mk WHERE mk.keyword_id<71692;
```

[2]Here we only list representative resources, and our model can be extended with other resources through our flexible embedding method in Sec. IV-C.



(a) Single-table query (b) Two-table join query (SMJ) (c) Two-table join query (BHJ) (d) Three-table join query (SMJ + BHJ)

Fig. 2. Cost of query execution plans with increasing memory sizes.

2) Two-table join query, the query execution plan involving SMJ operations:

```
SELECT COUNT(*) FROM title t,
     movie_companies mc
     WHERE t.id = mc.movie_id
     AND mc.company_id <213849
     AND mc.company_type_id>1;
```

3) Two-table join query, the query execution plan involving BHJ operations:

```
SELECT COUNT(*) FROM title t,
     movie_info_idx mi_idx
     WHERE t.id = mi_idx.movie_id
     AND t.kind_id<7
     AND t.production_year>1961
     AND mi_idx.info_type_id<101;
```

4) Three-table join query, the query execution plan involving SMJ and BHJ operations:

```
SELECT COUNT(*) FROM title t,
     movie_companies mc,
     movie_keyword mk
     WHERE t.id = mc.movie_id
     AND t.id = mk.movie_id
     AND mc.company_id = 43268
     AND mk.keyword_id < 2560;
```

In our experiments, we use the physical plans generated by Catalyst, the query optimizer of Spark SQL. In Catalyst, the optimized logical plan develops multiple physical execution plans. We fetch each physical execution plan of each query and evaluate them. For complex multi-table join queries, we select the first three Catalyst-generated physical execution plans for evaluation. Note that for the queries on one table, normally there are only two physical execution plans. We investigated each of the query plans for the above four queries. The execution plans for the same query differ in operator type, operating conditions, or plan tree structure. In the case of the single-table query, the difference between its two physical plans is the variation in the conditions in the `File Scan` operators.

We set the number of cores in each executor and the number of executors as 2 and then vary the size of executor memory. Fig. 2 presents the impact of executor memory on the cost of different query plans. We can see that the memory of each executor greatly affects the cost of query execution plans. Even for a single-table query, the cost of its execution plans varies with memory. In addition, changing the memory of each executor will also sometimes change the optimal execution plan of a query. Taking the two-table query (Fig. 2(c)) as an example, when the executor memory is 4GB or 5GB, the optimal query execution plan for this query is $plan3$, while the rest is $plan1$.

Increasing the memory of each executor does not necessarily reduce the cost of the execution plan because a cluster has a finite amount of memory. By increasing the memory of each executor, we will lessen the disk I/O, but it may also increase the blocking time of tasks. Also, different operators may apply to different memory sizes. For example, SMJ is more advantageous than BHJ in a certain memory range [35]. Allocating the right resources to the query plan requires a balanced configuration of the memory of each executor and the number of executors. There is a complex non-linear relationship between the memory and number of cores and executors and the cost of execution plans. Even though several studies [31], [32] try to match the optimal resources to query execution plans, the optimal resources are not always satisfied in fact. Therefore, we aim to develop an accurate cost model to predict an optimal execution plan for a query with a limited resource allocation. Our resource-aware model captures the real-time resources allocated by the resource manager for the query and selects the best execution plan for the query given the real-time resources.

IV. Resource-aware Attentional LSTM Model

Relational databases usually rely on statistical information to estimate the cost of query execution plans. Unlike relational databases, the cost of query execution plans in Spark SQL also depends on the available resources. We need to combine real-time resources to estimate the cost of query plans. Therefore, we propose a deep cost model to find the best combination of resources and query execution plans, enabling the learning model to generate optimal run-time plans.

A. Problem Definition

When Spark SQL runs a query, the query optimizer will generate several query execution plans for the query and select the best one. At the same time, the system allocates resources to run the query. The cost model is an essential component of

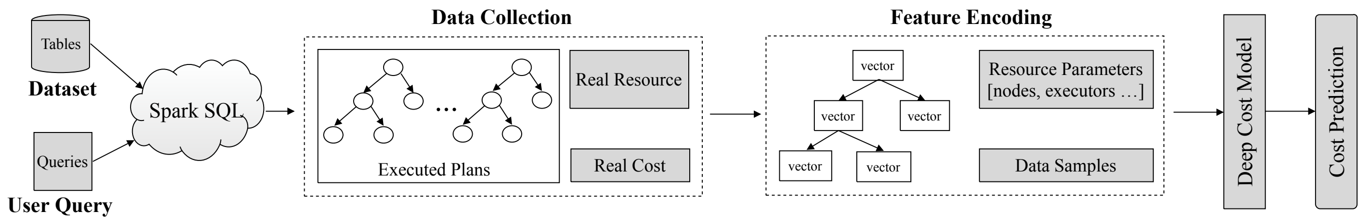


Fig. 3. An overview of our end-to-end resource-aware query optimization framework. There are three main parts: data collection, data feature encoding, and learning with a deep cost model.

the query optimizer used to evaluate the cost of query execution plans. A query execution plan is a tree structure containing multiple operation nodes, where the child nodes affect the operation of the parent node.

We set $Q = \{q_1, q_2, q_3, ..., q_{|Q|}\}$ to be a set of queries submitted by users, where $|Q|$ denotes the total number of queries. $P = \{p_1, p_2, p_3, ..., p_i\}$ includes all query execution plans for each query, where $p_i = \{v_1, v_2, v_3, ..., v_n\}$ is the representation of a query execution plan, and n is the total number of nodes in the execution plan p_i. v_n represents an operation node vector, and can be defined as $v_n = \{E_n, N_n\}$, where E_n denotes the connections between node v_n and other nodes and N_n denotes the operational features of node v_n. Our aim is to predict the cost of a query execution plan by developing a deep learning model based on the real-time resources $Re = \{r_1, r_2, r_3, ..., r_j\}$ and features extracted from query plans.

B. Overview

The overview of our resource-aware query optimization framework is shown in Fig. 3. It mainly consists of three phases:

1) The data collection phase, which aims to obtain many query execution plans with the actual cost and resource consumption.
2) The data feature encoding phase, which aims to encode the query execution plans and other features as embedding vectors.
3) The learning phase, where the deep cost model is trained based on the collected data.

Data Collection. This phase generates data needed for training the deep cost model. When Spark SQL receives multiple queries and the corresponding data for each query, we obtain all its query execution plans, the actual cost and resource consumption of each plan.

Feature Encoding. Feature encoding focuses on extracting key factors that affects the cost and encoding it into feature vectors. Query execution plans are bottom-up tree structures. The feature encoding consists of two parts: one is the information contained in each node, and the other is the relationships between the nodes. Meanwhile, we obtain the real-time resource consumption of each execution plan. The execution plan vectors and the resource vectors are then used as inputs to the deep cost model. In addition, we input statistical information like cardinality into the cost model. An intuitive idea to represent node information, such as operations and table names, is using *one-hot encoding* [37]. However, the one-hot encoding does not perform well to represent complex query conditions. Moreover, it may produce sparse vector representations making it difficult to extract useful features. Instead, we use *word2vec* [38] to overcome these issues. (see Sec. IV-C)

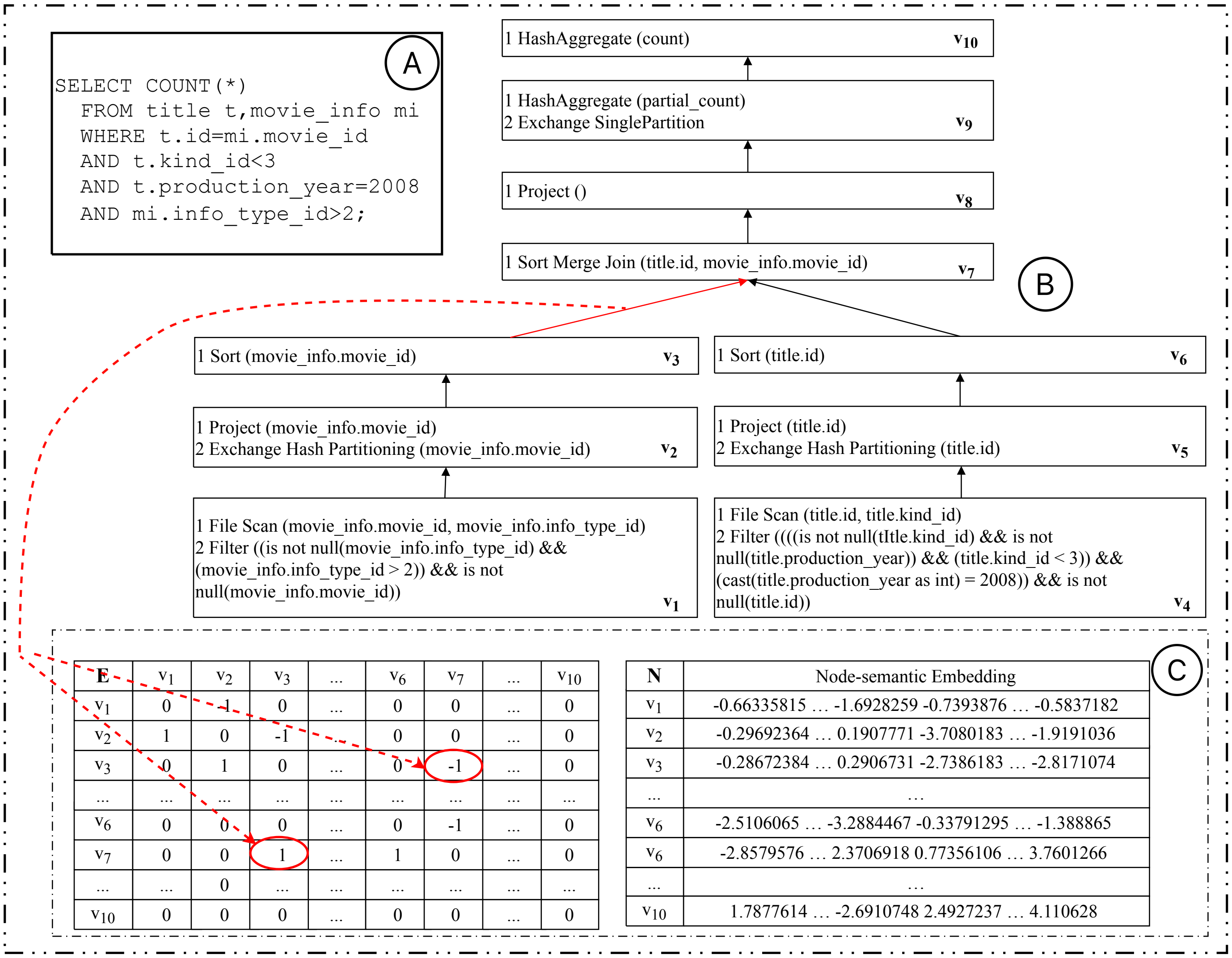
Deep Cost Model. A deep cost model is built based on the collected data. We propose an end-to-end resource-aware attentional model for predicting the execution cost of query plans. We convert node information and correlations between nodes into an embedding matrix to learn the feature representation of query execution plans. (see Sec. IV-D)

Cost Prediction. The trained deep cost model can be used to predict the execution cost of query plans. Each query execution plan is converted into a feature embedding and, with the other features together, input to the deep cost model to predict its cost.

C. Feature Encoding

In Spark SQL, both query execution plans and running resources are important factors that affect the query performance. The main challenge in modeling these factors is how to encode different query execution plans. First, query execution plans are bottom-up tree structures, and the input of parent nodes depends heavily on the output of their child node. It requires to capture the correlation between nodes to preserve the influence of child nodes on their parents. Secondly, we need to extract the information for each node and design a reasonable encoding method to extract the features of different nodes. To address these issues, we propose an embedding model consisting of a node-semantic embedding and a plan-structure embedding.

Node-semantic Embedding. In query execution plans, each node usually consists of execution statements containing information such as operation type, query predicates, and operation data. These nodes are stacked together in the order of execution to form complete query execution plans. Operations in Spark SQL are the physical operations, which include scan operation (e.g., File Scan), join operation (e.g., Sort Merge Join, Broadcast Hash Join, Broadcast Nested Loop Join), sort operation (e.g., Sort by, Order by), aggregation operation (e.g., Aggregate, Hash Aggregate, Sort Aggregate), and partitioning operation (e.g., Exchange Hash Partition, Exchange Single Partition). Predicates are filter/join conditions, combinations of operators, columns, and operating conditions. Fig. 4 shows an example of the physical execution plan, where we can see


A
B
C
1 HashAggregate (count) v10
1 HashAggregate (partial_count)
2 Exchange SinglePartition v9
1 Project () v8
1 Sort Merge Join (title.id, movie_info.movie_id) v7
1 Sort (movie_info.movie_id) v3
1 Sort (title.id) v6
1 Project (movie_info.movie_id)
2 Exchange Hash Partitioning (movie_info.movie_id) v2
1 Project (title.id)
2 Exchange Hash Partitioning (title.id) v5
1 File Scan (movie_info.movie_id, movie_info.info_type_id)
2 Filter ((is not null(movie_info.info_type_id) && (movie_info.info_type_id > 2)) && is not null(movie_info.movie_id)) v1
1 File Scan (title.id, title.kind_id)
2 Filter ((((is not null(tItle.kind_id) && is not null(title.production_year)) && (title.kind_id < 3)) && (cast(title.production_year as int) = 2008)) && is not null(title.id)) v4

```
SELECT COUNT(*)
  FROM title t,movie_info mi
  WHERE t.id=mi.movie_id
  AND t.kind_id<3
  AND t.production_year=2008
  AND mi.info_type_id>2;
```

E	v_1	v_2	v_3	...	v_6	v_7	...	v_{10}
v_1	0	-1	0	...	0	0	...	0
v_2	1	0	-1	...	0	0	...	0
v_3	0	1	0	...	0	-1	...	0
...	...	...	...	...	...	...	...	...
v_6	0	0	0	...	0	-1	...	0
v_7	0	0	1	...	1	0	...	0
...	...	0	...	...	...	...	...	...
v_{10}	0	0	0	...	0	0	...	0

N	Node-semantic Embedding
v_1	-0.66335815 … -1.6928259 -0.7393876 … -0.5837182
v_2	-0.29692364 … 0.1907771 -3.7080183 … -1.9191036
v_3	-0.28672384 … 0.2906731 -2.7386183 … -2.8171074
...	…
v_6	-2.5106065 … -3.2884467 -0.33791295 … -1.388865
v_6	-2.8579576 … 2.3706918 0.77356106 … 3.7601266
...	…
v_{10}	1.7877614 … -2.6910748 2.4927237 … 4.110628

Fig. 4. A query execution plan encoding example. Ⓐ: a SQL query. Ⓑ: an execution plan for this query in Spark SQL. Ⓒ: encoding of this execution plan, where E is the structural feature embedding and N is the node semantic embedding.

that node v_1 contains two execution statements, including a scan statement:

```
File Scan (movie_info.movie_id,
movie_info.info_type_id)
```

and a filter statement:

```
Filter ((is not null
(movie_info.info_type_id) &&
(movie_info.info_type_id>2)) &&
is not null (movie_info.movie_id)),
```

where `>`, `&`, `is not null` are operators; `movie_id`, `info_type_id` are columns; `movie_info` is the table to which the columns belong; and `2` is the operating condition.

We first apply explicit one-hot encoding to represent different nodes. For example, each operation type can be encoded as shown in Table II. We can represent other features such as column and operator in the same way. One-hot encoding is a more intuitive way, but it suffers from two problems. One is that joining each feature encoding will produce high-dimensional and sparse vector, which is not conducive to extract effective information. The other is the difficulty of representing the complex predicate operating conditions. If the operating conditions are numeric, we can use a normalized floating-point encoding. If it is a string value, we cannot use one-hot encoding to represent it. Moreover, one-hot encoding is not conducive to feature extraction between similar nodes.

Word2vec [38] is an embedding method that maps each word in statements to a vector representing the relationship between words. The operations of each node in the query execution plan consist of different execution statements. Therefore, we can use word2vec to extract the operational features of each execution statement. Firstly, it is possible to represent complex predicate conditions, no matter they are numbers or strings. Secondly, it facilitates the capture of semantic information about each operation statement and the correlation between similar nodes.

Word2vec maps words into a new space by embedding them so that semantically similar words are close to each other in that space. Based on the word embedding, we can learn the relationship between words by computing their embeddings' Euclidean distance. We consider operators in

TABLE II
AN EXAMPLE OF ONE-HOT ENCODING

Operation type	One-hot encoding
File scan	0000001
Project	0000010
Sort	0000100
Sort merge join	0001000
Hash aggregate	0010000
Exchange single partition	0100000
Exchange hash partition	1000000

execution statements as words whose vectors are similar in spatial coordinates. Using the word2vec encoding to execute statements, the distances of semantic embedding vectors of similar nodes are similar. In contrast, the vectors generated using the one-hot encoding are independent and thus the one-hot encoding cannot learn the association relationships between similar nodes.

Structure Feature Encoding. Query execution plans are bottom-up execution structures where the output of all child nodes affects the execution cost of their parents. Therefore, we encode each operation node's features and extract the connection relationships between nodes. We treat each query execution plan as a directed acyclic graph and sort the operation nodes according to the execution order. We then extract the out-degree and in-degree of each node to form an edge embedding matrix that reflects the correlation between the nodes in the query plan. As shown in Fig. 4, node v_7 is the parent of v_3 and v_6 and the child of v_8. Therefore, disposing of v_3 and v_6 as 1 and v_8 as -1 is the structure vector of node v_7.

We use the information about operation nodes and the inter-node correlations to model each query execution plan, each of which is composed of node-semantic embedding and structure feature embedding. As shown in Fig. 4, we define the query plan as $p = [E, N]$, where $E = \{E_1, E_2, ..., E_n\}$ is the edge embedding matrix, representing the relationship between each node, and $N = \{N_1, N_2, ..., N_n\}$ is the node embedding matrix, with each row representing the features of a node operation, consisting of the embedding vectors of the operating statements.

Resource Information Embedding. In theory, the system's resource manager allocates to a query the number of executors, the number of the executor cores, and the executors' memory to reveal the resources available to run the query. To extract these resource features r_j easily and quickly, we normalize each feature value into the range of [0,1]:

$$r_j{}^* = \frac{r_j}{\max(r_j)} \tag{1}$$

where $\max(r_j)$ represents the maximum value of the feature r_j. We set $\max(r_j)$ to the maximum available r_j of the system (means the system performs a single query task and resources are not shared).

Other Features. The external features such as cardinality and unique values also significantly impact the cost of execution plans. Similar to the resource information, we normalize each feature value into the range of [0,1].

D. Deep Cost Model

For the query cost estimation, we use a Recurrent Neural Network (RNN), called the Long Short Term Memory (LSTM) network [39], to learn the overall representation of query execution plans. Unlike traditional machine learning models, LSTM has long-term memory capabilities, with a network structure consisting of one or more forgettable and memorable units. However, the results may be less satisfactory if using the basic LSTM network structure here. Firstly, the LSTM network does not handle the relationships between nodes very well. Although we consider the structural features of execution plans when encoding them, this does not highlight strongly relevant nodes (e.g., the nearest neighbor node). In addition, we should capture the impact of resources on each node and give more attention to nodes that are sensitive the change of resources. Therefore, we propose a Resource-Aware Attentional LSTM model (RAAL for short) to address these issues and the overall architecture is shown in Fig. 5.

We first embed query execution plan vectors (both node-semantic embedding and structure feature embedding) into the plan feature layer to learn the potential features of execution plan trees. For the output vectors of the plan feature layer, we input them to the node-aware attention layer and the resource-aware attention layer, respectively, to improve the accuracy of query execution cost prediction. The node-aware attention layer captures the features between the current node and its strongly associated nodes. The resource-aware attention layer captures the impact of resources on the execution of each node. We then use multiple dense layers to obtain a high-quality feature representation and predict the cost of the execution plan.

Embedding Layer. Given a query execution plan containing n nodes, the first step is to transform each node into a real embedding vector e_i. After feature extraction, we obtain the node-semantic embedding N and structure feature vector E for each query execution plan. Then, we join N and E as embedding vectors of the query execution plan. In the embedding layer shown in Fig. 5, the embedding $emb = \{e_1, e_2, ..., e_n\}$ is the input to the RAAL model. The embedding vector of a query execution plan is large and sparse. We need to process it to obtain a high-quality feature representation.

Plan Feature Layer. As LSTM networks can learn long-term dependency information, we use it to model query execution plans better to extract the semantics of each node's operational statements.

LSTM networks are composed of multiple memory cells. In contrast to a normal RNN that uses only one temporal state h, the LSTM has a cell state c for storing important information in addition to the temporal state h. The LSTM basic block can be added and forgotten to previous input information through an internal gate structure. The plan feature layer of Fig. 5 shows the relationship between the basic cells inside an LSTM

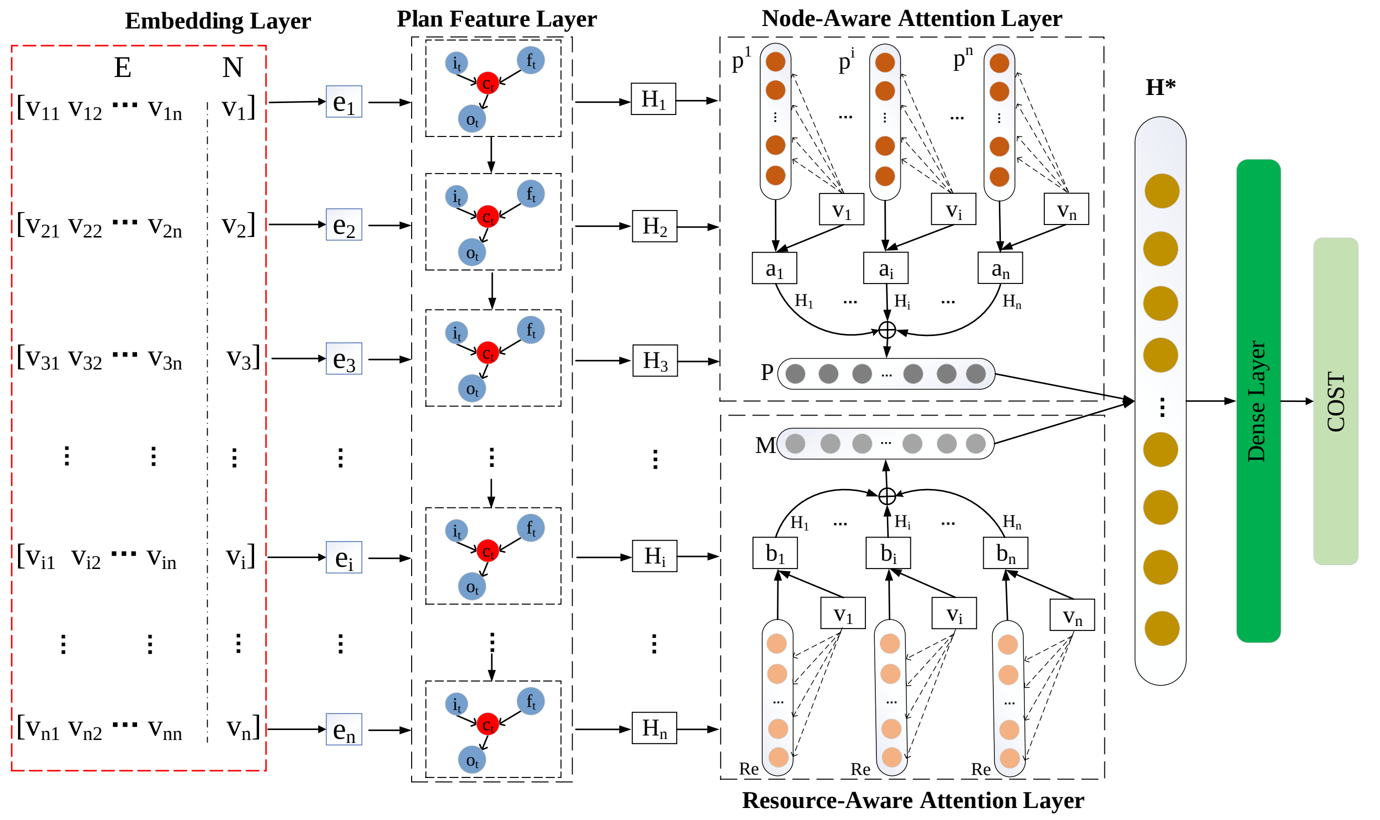


Fig. 5. The architecture of the proposed Resource-Aware Attentional LSTM model. Encoding the query execution plan first and an embedding $emb = \{e_1, e_2, ..., e_n\}$ that input to the model. Then, through the LSTM layer to learn the overall representation H_i of each plan node. Subsequently, through the node-aware attention to learn the representation of relationships between nodes, and through the resource-aware attention to learn the impact of resources on execution nodes. Finally, the predicted cost is output through multiple dense layers.

block. The update of the hidden state h and cell state c is shown as follows:

$$
\begin{aligned}
i_t &= \sigma(W_{hi}h_{t-1} + W_{xi}x_t + b_i) && (2)\\
f_t &= \sigma(W_{hf}h_{t-1} + W_{xf}x_t + b_f) && (3)\\
\tilde{c}_t &= \tanh(W_{hc}h_{t-1} + W_{xc}x_t + b_c) && (4)\\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t && (5)\\
o_t &= \sigma(W_{ho}h_{t-1} + W_{xo}x_t + b_o) && (6)\\
h_t &= o_t \cdot \tanh(c_t) && (7)
\end{aligned}
$$

where σ is the sigmoid activation function that outputs a value between 0 and 1, describing how much of each part can pass, $\tanh$ is the hyperbolic tangent function that outputs a value between -1 and 1, W is the weight to be learned, and b is the bias. Notations i_t, f_t, o_t represent input gate, forgetting gate, and output gate, respectively, and each containing a sigmoid neural network layer and a per-bit multiplication operation. The input gate determines how much of the input data at time t needs to be saved to the cell state. The forgetting gate determines how much of the cell state at time $t-1$ needs to be preserved to time t. The output gate controls how much of the unit state needs to be output at time t. The cell state c_t incorporates information about the past cell state c_{t-1}, the candidate memory cell and new input data, and $\tilde{c}_t$ is the candidate memory cell. The hidden vector h_t is the final output of the LSTM block.

Node-Aware Attention Layer. Query execution plans are bottom-up execution tree structures, where the output of the child nodes is the input to the parent node, so we need to consider the impact of different child nodes on their parent node. Let $p = \{v_1, v_2, v_3, ..., v_i, ..., v_n\}$ denote the set of nodes, and n be the total number of nodes in the query execution plan p. Let $p^i = \{v_1^i, v_2^i, v_3^i, ..., v_m^i\}$ denote the set of child nodes associated with node v_i, and m is the total number of children of node v_i. The number of related children of each node is different, and only a subset of p^i has a strong influence with node v_i. Therefore, our node-aware attention mechanism uses a non-fixed feature representation. The core idea here is to compute the correlation scores of node v_i and its children in p^i. It captures the structure feature of the query execution plan and allows the model to learn the association relationship between the nodes more accurately. Thus, the adaptive attention network measures the relevance scores between node v_i and each child node in p^i by:

$$a_i = \frac{\exp(p^i v_i)}{\sum_{v_k \in p/p^i} \exp(p^i v_k)} \quad (8)$$

where $v_i \in R^{K\times 1}$ and $p^i \in R^{m\times K}$.[3] p/p^i represents the set of all nodes in p except the nodes in p^i. $a_i \in R^{m\times 1}$ is the adaptive attention vector for node v_i. The larger the value

[3] K is the dimension of the latent vector, and we set it as a constant 32 by default in our experiments.

of a_i, the higher the correlation between node v_i and the corresponding node in p^i. It allows the trained network to learn the connections between nodes better and gives each node a more appropriate weight distribution.

As shown in the Node-Aware Attention layer in Fig. 5, $H_i = [h_1, h_2, ..., h_n]$ is the hidden state of node v_i, which is produced by the LSTM network. Then we form the relationship representation of node v_i by a weighted sum of a_i:

$$P = \sum_{i=1}^{n} H_i a_i \tag{9}$$

where $P \in R^{m \times n}$ represents the relational feature between each node v_i and its own particular related child nodes.

Resource-Aware Attention Layer. As mentioned above, the resource is an essential factor in the cost of query execution plans. To allow each operation node to capture changes in resources, we add a resource-aware attention layer and give more attention to nodes sensitive to resource changes.

We let $Re = \{r_1, r_2, r_3, ..., r_j\}$ denote the system's real-time resources. Intuitively, the resources affect various nodes differently. As shown in Fig. 5, we propose a resource-aware attention mechanism to capture the extent to which resources affect different nodes. We capture the correlation of resources with each node:

$$b_i = \frac{\exp(Rev_i)}{\sum_1^n \exp(Rev_n)} \tag{10}$$

$$M = \sum_{i=1}^{n} H_i b_i \tag{11}$$

where $Re \in R^{j \times 1}$ is the representation vector of real-time resources.[4] A larger value of b_i represents a stronger influence of real-time resources on node v_i. Notation $M \in R^{j \times n}$ is a feature representation reflects the impact of the real-time resource Re on the node v_i.

Prediction Layer. We join P and M to obtain H^*, which is used for subsequent execution cost prediction. We join H^* and other statistical features together, and then map them to a lower-dimensional vector representation by multiple dense layers to predict the cost. In the RAAL model, we use the mean squared error (MSE) as our loss function, which is the most commonly used regression loss function.

V. Experiments

The critical question we focus on in our experiments is whether our proposed cost model RAAL considering the resource information, can have higher query performance in Spark SQL than the state-of-the-art models. To answer this question, we test the performance of the RAAL and the overall improvement in query performance with the addition of resource status. We evaluate the proposed approach and validate the performance of the system from the following three aspects:

[4]To be consistent with the dimensionality of the node v_i, we multiply Re with T in our experiments. T is a K-dimensional random vector that is automatically corrected during the training process.

TABLE III
Cluster Configurations

Configuration	**Details**		
Installation	Tencent Cloud	Ali Cloud	On Premises
Node	4	5	1
Core	4	12	8
Main Memory	16GB	48GB	64GB
Disk	100GB	500GB	5TB
Version	Hadoop 2.6.5 + Spark 2.4.7		

1) evaluating the efficiency of RAAL model and compare it with other methods;
2) assessing the impact of real-time resources on improving the accuracy of the predicted cost;
3) evaluating the adaptability of RAAL and show that RAAL can adapt to different workloads.

A. Setup

Datasets. We evaluate RAAL using two different benchmarks:

1) IMDB: a real-world dataset IMDB, which is an extension of the join order benchmark test [12]. Estimating the query cost of the IMDB dataset is much more complex compared to the standard dataset TPC-H. The correlation and skew distribution of the IMDB dataset are more complex than that of TPC-H. We use 6000 queries with 0-5 joins that contain two types of query workloads [17].
2) TPC-H: the standard TPC-H benchmark [40], using a scale factor of 100. We generated 5000 queries based on the benchmark query templates.

These two datasets contain two types of query workloads. The first workload type has only predicates with numeric attributes. The second workload type contains complex predicates with string attributes.

Training Data. We collect data from a cloud installation of Spark and a local installation of Spark, respectively. Detailed descriptions of configurations are in Table III. We implement the RAAL on Spark SQL built in a local environment to compare with the relational database cost model. To approximate the variation of resources in a real scenario, we run all queries in multiple resource states on the clouds. We run the IMDB dataset on *Tencent Cloud* [41] and the TPC-H dataset on *Ali Cloud* [42] to verify that RAAL can be adapted to different clusters and data. Each training sample comprises a query execution plan, the resource consumption, the related statistical features, and the corresponding execution cost. Note that the available resources are measured at the start of an execution plan. All experiments were conducted by randomly placing 80% of the available queries in a training set and using the remaining 20% of the available queries as a test set.

Evaluation Methods. We compare our RAAL with two most related state-of-the-art studies, which we also discussed in Sec. II-A:

1) GPSJ [10]: This is a cost model for Spark SQL that covers the class of Generalised Projection, Selection, Joining (GPSJ) queries. The cost model bases on cluster

and application parameters and a set of database statistics. It is the state-of-the-art cost model for Spark SQL but requires significant person-hours of engineering to tune the parameters, so we used the hand-crafted models from this paper directly.

2) TLSTM [17]: We implemented the TLSTM-based cost model, which is the state-of-the-art approach to cost estimation applicable to relational databases. For each operator in a query plan, TLSTM uses an LSTM unit to estimate the operator's cost. The inputs to each LSTM unit are the features of the operator and the intermediate results of the sub-operators, and the outputs are the predicted execution costs. These units are organized into a tree structure to obtain the total cost of the query. Note that TLSTM does not highlight the strong correlation between different nodes. In addition, it is not directly applicable to big data processing engines like Spark SQL.

Metrics. To evaluate the cost estimation performance of each method, we used two metrics: relative error (RE) and mean square error (MSE). Let P be the set of test query plans, es be the estimated cost of p ($p \in P$), and ac be the actual cost of p. The relative error (RE) is defined as follows:

$$RE = \frac{1}{|P|} \sum_{p \in P} \frac{|ac(p) - es(p)|}{ac(p)} \tag{12}$$

Relative errors sometimes tend to be underestimated. We also measured the MSE, which penalizes overestimation and underestimation symmetrically:

$$MSE = \frac{1}{|P|} \sum_{p \in P} [ac(p) - es(p)]^2 \tag{13}$$

The smaller the value of RE and MSE means the higher the prediction accuracy of the method.

In addition, we measured statistical correlations (COR) and coefficients of determination (R^2) to evaluate the relationship between the estimated cost and actual costs. Their values range from 0 to 1, with the higher the value, the better the fit.

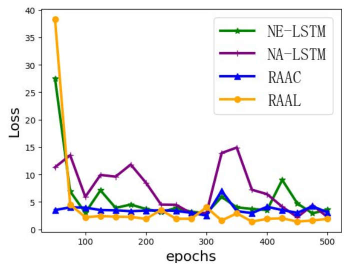
$$COR = \frac{\sum_{p \in P} [ac(p) - \overline{ac}][es(p) - \overline{es}]}{\sqrt{\sum_{p \in P} [ac(p) - \overline{ac}]^2 \sum_{p \in P} [es(p) - \overline{es}]^2}} \tag{14}$$

$$R^2 = 1 - \frac{\sum_{p \in P} [ac(p) - es(p)]^2}{\sum_{p \in P} [ac(p) - \overline{ac}]^2} \tag{15}$$

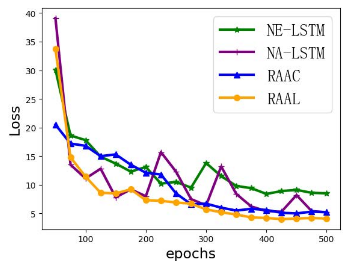
Training Environment. We use a machine with Intel CPU i7-6700k v4, 48GB Memory, and GeForce RTX 2070 GPU to train RAAL. We implement our models with PyTorch in a Windows 10 64bit system.

B. Evaluation on Our Methods

This section contains three parts: 1) A validation of the performance of the RAAL model, including an analysis of each of its modules; 2) A comparison between RAAL and the state-of-the-art cost model for relational databases; 3) A comparison between RAAL with the state-of-the-art cost model for Spark SQL.



(a) IMDB dataset



(b) TPC-H dataset

Fig. 6. Cost errors on different datasets.

TABLE IV
EVALUATION OF OUR APPROACH.

Method	IMDB				TPC-H			
	RE	MSE	COR	R^2	RE	MSE	COR	R^2
NE-LSTM	0.134	1.157	0.965	0.933	0.183	3.791	0.968	0.946
NA-LSTM	0.120	0.776	0.980	0.956	0.196	4.165	0.976	0.927
RAAC	0.133	1.214	0.966	0.931	0.178	2.722	0.982	0.956
RAAL	**0.102**	**0.553**	**0.986**	**0.969**	**0.172**	**1.997**	**0.985**	**0.977**

1) Analysis of Our Model: Our proposed RAAL model consists of three components: the plan embedding module, the feature extraction module, and the aware attention module. We have verified and analyzed the performance of each module, and the results are presented in Table IV. Fig. 6 shows the comparison between the different methods, with the vertical coordinate being the loss value of models and the horizontal coordinate being the number of training iterations. The results show that the RAAL model can effectively extract features of query plans and resources and improve the accuracy of cost prediction.

Structure Embedding vs. Non-Structure Embedding. The plan embedding module of the RAAL model consists of two parts: node-semantic embedding and structure feature embedding. Previous studies focus more on node feature extraction while neglecting the structural features of query plans. To verify the effectiveness of structure feature embedding, we implemented the RAAL model without structure feature embedding (named as **NE-LSTM**), and Fig. 6 shows the comparison results. The method with structure feature embedding (RAAL) outperforms the method without structure feature embedding (NE-LSTM) because structure feature embedding explicitly reveals lower-level nodes' effect on upper-level nodes in query execution plans.

Node-Aware Attention vs. None Node-Aware Attention. The method with node-aware attention (RAAL) outperforms the method without node-aware attention (named as **NA-LSTM**) because node-aware attention can capture corresponding nodes and learn the relational features between them. Fig. 6 shows the training process of the two models, and we can observe that the loss of NA-LSTM fluctuates dramatically, and thus the model is not stable. It suggests that the NA-LSTM model does not learn the relationships between strongly associated nodes well and thus does not stably capture the structural features of the query execution plan. More details

TABLE V
COMPARISON WITH THE COST MODEL OF RELATIONAL DATABASES.

Dataset	Method	RE	MSE	COR	R^2
IMDB	TLSTM	0.148	5.983	0.969	0.936
	RAAL	**0.074**	**1.856**	**0.993**	**0.985**
TPC-H	TLSTM	0.314	6.767	0.839	0.825
	RAAL	**0.262**	**1.672**	**0.862**	**0.843**

for the impact of resource-aware attention will be discussed in Sec. V-C.

LSTM vs. CNN. The difference between RAAL and **RAAC** (replacing the **L**STM with a **C**NN) is the extraction of the query plan embedding features. RAAL uses an LSTM network to learn the query execution plan features, while RAAC uses a convolution neural network (CNN). The CNN considers elements to be independent of each other during learning, and inputs and outputs are also independent, which is not conducive to learning the semantics of the executed statements of the nodes. In addition, the LSTM has an additional memory cell to avoid the omission of complex information.

2) Comparison with Relational Database Cost Models: Existing cost models are mainly applicable to relational databases and do not consider how they might be better suited to Spark SQL. Unlike relational databases, the resources of Spark SQL share across multiple tasks. To compare with relational database cost models, we installed Spark SQL locally and fixed the resources available for each query. It is similar to the application scenario of a relational database running in a specified resource environment. Fixed resource vector input to the RAAL's resource-aware attention layer at this point and the resource parameters are the same for each training data.

We compare RAAL with the state-of-the-art relational database cost model TLSTM and Table V shows the results. RAAL has a lower MSE and RE than TLSTM and a higher COR and R^2 than TLSTM. The reason is that RAAL uses a plan structure embedding and a node-aware attention mechanism to more easily capture the structural features of multi-layer query plans. While TLSTM uses a tree structure to model the influence of lower-level nodes on higher-level nodes in a query plan, which can also learn the structural features of query plans, it may leave out the information about the underlying nodes. In addition, the RAAL model gives more attention to the nodes that have a more significant impact on the execution cost.

3) Comparison with the cost model GPSJ for Spark SQL: GPSJ is a cost model for Spark SQL that covers the generalized projection, selection and join query classes. GPSJ is the first to take the Spark computational paradigm into account and is the state-of-the-art cost model for Spark SQL. We show the results of the comparison between RAAL and GPSJ in Table VI. GPSJ is a cost model developed based on cluster and application parameters and a set of database statistics, and it is not a learning-based approach. GPSJ has significant errors in estimating the cost of query plans according to our experiments. The reasons are two-fold: 1) over-reliance on statistical information; 2) artificially constructed cost functions have difficulty learning complex non-linear mappings.

TABLE VI
COMPARISON WITH THE GPSJ COST MODEL IN SPARK SQL.

Dataset	Method	RE	COR
IMDB	GPSJ	0.203	0.984
	RAAL	**0.102**	**0.986**
TPC-H	GPSJ	0.242	0.965
	RAAL	**0.172**	**0.977**

C. Evaluation on the Impact of Resources

In this section, we evaluate the impact of resources on the accuracy of cost models. We installed Spark SQL in Tencent Cloud and Ali Cloud respectively, and recorded the actual performance of all possible plans for each query against different resource states. We ran IMDB on Tencent Cloud and obtained $63,000$ data records, and ran TPC-H on Ali Cloud and obtained $50,000$ data records.

To highlight the significant impact of resources on the predictive performance of cost models, we implemented the NE-LSTM, NA-LSTM, RAAC, and RAAL without the resource-aware attention layer, respectively. Table VII shows the comparison results, where the data on the left in each column are the results without resource-aware attention, and the bolded font on the right is the results with resource-aware attention. We can see that adding the resource-aware attention mechanism improves the performance of each method. For TPC-H, the MSE of each model with resources considered is much lower than that of each model without considering resources. It indicates that resource features have a significant impact on big data queries.

To provide a more intuitive description of the effectiveness of the resource-aware attention mechanism, we plot the actual and estimated costs of each query execution plan. Fig. 7 shows the distribution of estimated costs for the RAAL model, one without the resource-aware mechanism and the other with resource-aware attention. We can see that the green dots are significantly more divergent than the blue dots, indicating that the model with the resource-aware attention layer performs better. Note that the figures (7(c) and 7(d)) for the TPC-H dataset are more sparse than those for the IMDB dateset. The reasons for this are: (1) TPC-H has fewer queries than IMDB; and (2) the cost distribution for TPC-H has a larger variance.

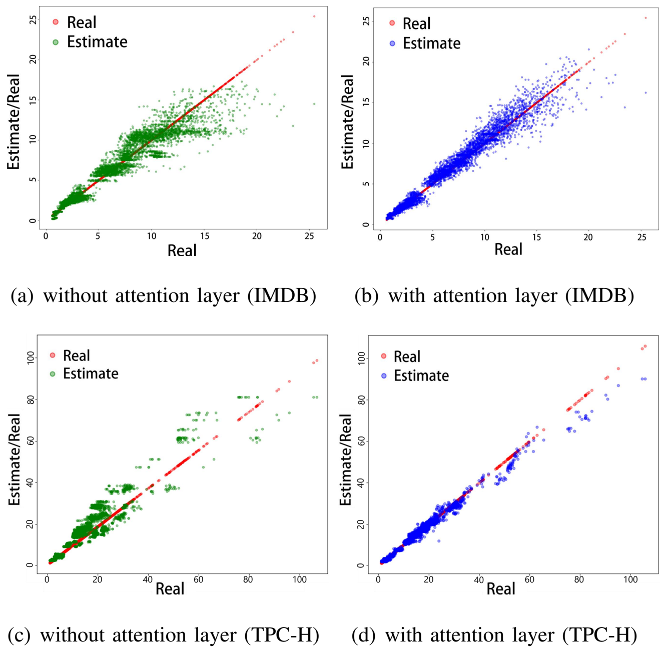
D. Evaluation on Adaptability

Different running environments can significantly affect query performance. We tested the adaptability of RAAL when the running environment changes. As the memory is often a performance bottleneck, we verified the effectiveness of RAAL in clusters with different executor memory sizes. Fig. 8 shows the results of the IMDB dataset.

We have the following observations: the performance of the RAAL model remains stable under different cloud environments. As can be seen from Fig. 8, the values of both COR and R^2 remain above 0.9, which tends to be flat. It indicates a strong correlation between the estimated cost of the RAAL

TABLE VII
EVALUATING THE EFFECTIVENESS OF RESOURCES

Method	IMDB								TPC-H							
	RE		MSE		COR		R^2		RE		MSE		COR		R^2	
NE-LSTM	0.252	**0.134**	1.434	**1.157**	0.964	**0.965**	0.918	**0.933**	0.233	**0.183**	16.602	**3.791**	0.959	**0.968**	0.685	**0.946**
NA-LSTM	0.215	**0.120**	1.756	**0.776**	0.968	**0.980**	0.908	**0.956**	0.225	**0.196**	16.661	**4.165**	0.960	**0.976**	0.693	**0.927**
RAAC	0.155	**0.133**	2.367	**1.214**	0.964	**0.966**	0.881	**0.931**	0.215	**0.178**	13.929	**2.722**	0.972	**0.982**	0.815	**0.956**
RAAL	0.155	**0.102**	0.785	**0.553**	0.980	**0.986**	0.956	**0.969**	0.217	**0.172**	13.228	**1.997**	0.957	**0.985**	0.817	**0.977**



(a) without attention layer (IMDB) (b) with attention layer (IMDB)

(c) without attention layer (TPC-H) (d) with attention layer (TPC-H)

Fig. 7. Distribution of estimated cost by RAAL

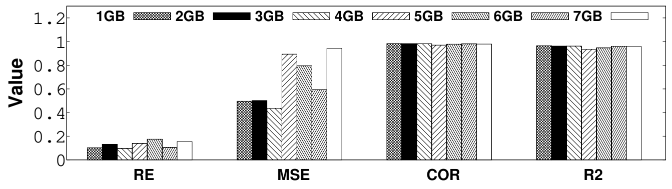


Fig. 8. Adaptability on the increasing memory size, where we increase the memory size from 1GB to 7GB to observe the changes of four metrics, respectively.

model and the Spark SQL real execution time. In addition, RE stays around 0.1 and MSE stabilises below 1. Thus the RAAL model can adapt to different memory sizes, and its performance remains stable under different cloud environments.

E. Evaluation on Efficiency

We show the training time and test errors for different data sizes in Table VIII. We can see that it takes less than two hours to train our model with 50K training examples. Even if we need to update our cost model with many new data records, it will not take much time to retrain our model. The test error decreases as the amount of training data increases, proving that the more historical data is provided, the more accurate the knowledge learned by the model. However, even with less historical data, the testing error of our model is still reasonably small.

We also evaluated the online estimation time of the RAAL model, and Table IX shows the results. We can see that

TABLE VIII
TIME AND ERRORS WITH VARIOUS TRAINING SIZES

Dataset	Metric	Training Size (k)				
		10K	**20K**	**30K**	**40K**	**50K**
IMDB	MSE	2.726	2.542	1.978	1.141	0.746
	Time (min.)	55	59	62	87	99
TPC-H	MSE	4.637	3.458	2.853	2.244	1.997
	Time (min.)	49	56	65	84	97

TABLE IX
TIME(MS) ON VARIOUS ESTIMATION SIZES

Dataset	Estimated Size				
	50	**100**	**150**	**200**	**300**
IMDB	2.215	2.386	2.652	2.783	2.915
TPC-H	2.636	3.177	3.489	3.638	4.194
Average	2.423	2.782	3.071	3.207	3.555

our method predicts the execution time of 100 queries in only 2.782ms. TLSTM, which uses batch techniques to improve computational efficiency, takes 3.342ms to estimate the execution time of 100 queries. GPSJ computes the cost of a query plan with a charge of up to 50ms. We can see that the performance of the learning-based methods is much higher than that of the statistical-based methods. Based on the above observations, we can conclude that the estimation time of RAAL is negligible.

VI. CONCLUSIONS

We investigated a resource-aware deep cost model for query processing on Spark SQL in this paper. We first conducted a detailed analysis of the impact of resources on the cost of execution plans. Then, we proposed an LSTM-based cost model RAAL with an attention mechanism to predict the cost for each query plan and further selected the fastest one to execute. Experiments on different datasets verified that our proposed model beats state-of-the-art models in Spark SQL in terms of both efficiency and accuracy.

In the future, we will focus on the dataset's effect on the selection of the optimal query plans and algorithms [43], [44]. Rather than using two datasets in this paper, the main objective is for the cold-start query optimization when we need to conduct queries on a newly loaded dataset without training new models.

ACKNOWLEDGMENTS

This work was partially supported by the National Key Research and Development Project of China (No. 2018YFB1003400), the National Natural Science Foundation of China (No. U1811263).

References

[1] "Twitter," https://twitter.com/, 2021.

[2] A. Dagnino, *Data Analytics in the Era of the Industrial Internet of Things*. Springer, 2021.

[3] Q. Wang, F. Zhou, J. Xu, and Z. Xu, "Efficient verifiable databases with additional insertion and deletion operations in cloud computing," *Future Gener. Comput. Syst.*, vol. 115, pp. 553–567, 2021.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[6] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *SIGMOD*, 2014, pp. 147–156.

[7] "Oracle — Integrated Cloud Applications and Platform Services," https://www.oracle.com/index.html, 2021.

[8] "MySQL," https://www.mysql.com/, 2021.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *SIGMOD*, 2015, pp. 1383–1394.

[10] L. Baldacci and M. Golfarelli, "A cost model for SPARK SQL," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 5, pp. 819–832, 2019.

[11] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: a survey," *Proc. VLDB Endow*, pp. 1–22, 2021.

[12] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015.

[13] L. Ma, B. Ding, S. Das, and A. Swaminathan, "Active learning for ML enhanced database systems," in *SIGMOD*, 2020, pp. 175–191.

[14] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *SIGMOD*, 2021, pp. 1275–1288.

[15] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019.

[16] R. Marcus and O. Papaemmanouil, "Towards a hands-free query optimizer through deep learning," in *CIDR*, 2019.

[17] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 307–319, 2019.

[18] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le, "Cost models for big data query processing: Learning, retrofitting, and our findings," in *SIGMOD*, 2020, pp. 99–113.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[20] S. Manegold, P. A. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *VLDB*, 2002, pp. 191–202.

[21] Y. Theodoridis, E. Stefanakis, and T. K. Sellis, "Cost models for join queries in spatial databases," in *ICDE*, 1998, pp. 476–483.

[22] H. Lan, Z. Bao, and Y. Peng, "A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration," *Data Sci. Eng.*, vol. 6, no. 1, pp. 86–101, 2021.

[23] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte, "Cardinality estimation using neural networks," in *CASCON*, 2015, pp. 53–59.

[24] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *CIDR*, 2019.

[25] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica, "Learning to optimize join queries with deep reinforcement learning," *CoRR*, vol. abs/1808.03196, 2018.

[26] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "Learning state representations for query optimization with deep reinforcement learning," in *SIGMOD*, 2018, pp. 1–4.

[27] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based query performance modeling and prediction," in *ICDE*, 2012, pp. 390–401.

[28] R. C. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1733–1746, 2019.

[29] X. Zhou, J. Sun, G. Li, and J. Feng, "Query performance prediction for concurrent queries using graph embedding," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1416–1428, 2020.

[30] Y. Li, M. Li, L. Ding, and M. Interlandi, "RIOS: runtime integrated optimizer for spark," in *SoCC*, 2018, pp. 275–287.

[31] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017, pp. 469–482.

[32] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan, "Perforator: eloquent performance models for resource optimization," in *ACM*, 2016, pp. 415–427.

[33] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel, "Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling," in *2017 USENIX ATC*, 2017, pp. 97–109.

[34] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[35] L. Viswanathan, A. Jindal, and K. Karanasos, "Query and resource optimization: Bridging the gap," in *ICDE*, 2018, pp. 1384–1387.

[36] A. Jindal, S. Qiao, R. Sen, and H. Patel, "Microlearner: A fine-grained learning optimizer for big data workloads at microsoft," in *ICDE*, 2021, pp. 2423–2434.

[37] C. Gao, X. He, D. Gan, X. Chen, F. Feng, Y. Li, T.-S. Chua, and D. Jin, "Neural multi-task recommendation from multi-behavior data," in *ICDE*, 2019, pp. 1554–1557.

[38] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[40] M. Pöss and C. Floyd, "New TPC benchmarks for decision support and web commerce," *SIGMOD Rec.*, vol. 29, no. 4, pp. 64–71, 2000.

[41] "Tencent Cloud," https://cloud.tencent.com/, 2021.

[42] "Ali Cloud," https://cn.aliyun.com/, 2021.

[43] S. Wang, Y. Sun, and Z. Bao, "On the efficiency of k-means clustering: Evaluation, optimization, and algorithm selection," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 163–175, 2020.

[44] M. A. Muñoz, Y. Sun, M. Kirley, and S. K. Halgamuge, "Algorithm selection for black-box continuous optimization problems: A survey on methods and challenges," *Information Sciences*, vol. 317, pp. 224–245, 2015.