April: An Automatic Graph Data Management System Based on Reinforcement Learning

Hongzhi Wang[1, 2], Zhixin Qi[1], Lei Zheng[1], Yun Feng[1], Junfei Ouyang[1], Haoqi Zhang[1], Xiangxi Zhang[1], Ziming Shen[1], Shirong Liu[1]
[1]School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
[2]Peng Cheng Laboratory, Shenzhen, China
{wangzh,qizhx}@hit.edu.cn,19S103245@stu.hit.edu.cn,fengyun5264@outlook.com,
{ouyangjunfeio2o2,inquisitiveZH,xiangxi.zhang.cs,shenziming1120}@gmail.com,1173710126@stu.hit.edu.cn

ABSTRACT

The great amount and complex structure of graph data bring a big challenge to graph data management. However, traditional management approaches cannot tackle the challenge. Fortunately, reinforcement learning provides a new approach to solve this problem due to its automation and adaptivity in decision making. Motivated by this, we develop April, an automatic graph data management system, which performs storage structure selection, index selection, and query optimization based on reinforcement learning. The system selects storage structure, indices effectively and automatically, and optimizes the SPARQL queries efficiently. April also offers a friendly interface for users, which allows users to interact with the system in a customized mode. We demonstrate the effectiveness and efficiency of April with two graph data benchmarks.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

graph data management; reinforcement learning; storage structure selection; index selection; query optimization

ACM Reference Format:
Hongzhi Wang[1, 2], Zhixin Qi[1], Lei Zheng[1], Yun Feng[1], Junfei Ouyang[1], Haoqi Zhang[1], and Xiangxi Zhang[1], Ziming Shen[1], Shirong Liu[1]. 2020. April: An Automatic Graph Data Management System Based on Reinforcement Learning. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340531.3417422

1 INTRODUCTION

With the rapid growth of graph data applications, there has been great amount of graph data in various domains, such as chemical compounds, protein networks, social networks, and knowledge graphs. To take advantage of massive graph data, effective graph data management is identified as the basic premise. In real applications, it is challenging to manage graph data effectively due to the following characteristics of graph data.

• **Large scale.** As the graph data applications attract more users, the scale of graph data rises rapidly. For example, there are more than 1.39 billion vertices and 400 billion edges in social network Facebook [1], and the size of knowledge graph Yago is about 91 GB. The large-scale property makes it difficult to decide how to store graph data and construct effective indices for them.

• **Various patterns.** Since graphs usually represent different semantics in various applications, the graph data patterns are various. For instance, a graph of users' preferences contains personal and transaction information of users. Hence, it can be managed with the pattern *user*(*name*, *gender*, *age*, *city*, *transaction date*, *items*, *amount*). A knowledge graph involves a great number of ground truths from various fields. To manage it, we consider all facts in it with a triple pattern (*subject*, *predicate*, *object*). The difference between patterns brings the challenge of designing specific storage structures and indices for the given graph data.

• **Frequent changes.** The frequency of updating graph data becomes higher as the number of users increases. For example, when a new user joins in a social network, personal information is brought as well as his relationships with other users. In addition, the workloads of graph data also change frequently with the increasing users. Taking intelligent speaker as an example, it responses to users' questions with its inner knowledge graph. As the question varies, the query workload of knowledge graph changes. This character requires a graph data management system to adjust the storage structures and indices according to the dynamic graph data and workloads.

Existing methods for selecting graph data storage structures and indices depend on the database administrators (DBAs) [6]. Unfortunately, they are unable to tackle the intractable challenges brought by the characteristics of graph data. On the one hand, there are many candidate partition approaches and storage structures due to the large scale and various patterns of graph data. It is over-demanding for DBAs to decide how to partition and store a large-scale graph. On the other hand, since the graph data and workloads change frequently, it is necessary to ensure the selected storage structure and indices are suitable to the current data and workloads. The knowledge of DBAs is not sufficient to adjust the storage structures and indices of graph data in time.

In the light of these challenges, effective techniques for graph data storage and retrieval are in a urgent demand. Fortunately, in recent years, reinforcement learning (RL for brief) methods are widely

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3417422

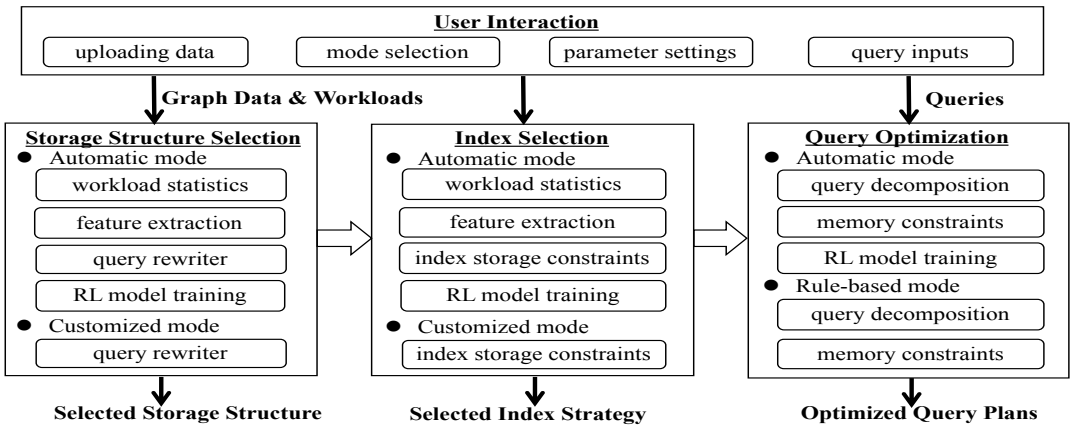


Figure 1: Framework Overview of APRIL

used to solve data management problems [2]. The automation and adaptivity of RL provide new opportunities for graph data management. The working definition of RL is "learning by doing". It makes decisions automatically by reforming its learned behavior via the performance metric on changing data or workloads [3]. Motivated by this, we develop April, an automatic graph data management system based on reinforcement learning. Our system focuses on the following objectives.

(1) **Effectiveness in graph data management.** After a thorough practice research, we adopt RL to manage graph data and develop three major graph data management functions in April, i.e. storage structure selection, index selection, and query optimization. These approaches are verified to be effective in the demo scenarios with graph data benchmarks.

(2) **Automatic graph data management mode.** Given the graph data and workload, our automatic graph data management mode provides users with the selected storage structure and the strategies of index creation. Based on the determined graph data storage and indices, the automatic mode produces optimized query plans according to the input queries without any user intervention.

(3) **Customized graph data management mode.** To satisfy users' personalized requirements and for the convenience of comparisons, we show the selected storage structure or indexing strategy and its corresponding total elapsed time of workloads to users. Users can decide to use our selected result or choose the customized graph data management mode which stores graph data and creates indices according to their requirements.

Our contributions are as follows: 1) We develop April, an automatic graph data management system based on RL. 2) April implements three automatic graph data management functions to store, index, and query graph data. 3) April provides a well-considered interface design for users to customize graph data management.

2 SYSTEM OVERVIEW

Figure 1 shows the framework of APRIL, which has three graph data management functions: 1) Storage structure selection, 2) Index selection, and 3) Query optimization. It also has a user interaction module for customization.

Backend. For sufficient storage size, efficient query processing, and mature transaction management mechanism, we choose the relational database as the backend. We store the graphs as relations and convert SparQL into structured query language (SQL) with user-defined functions (UDFs) to enable the external query optimization.

Storage structure selection. After users upload the graph and workloads, April first aims to select a storage structure for the graph data. For comparison, the system provides an automatic mode and a customized mode. In the automatic mode, the storage structure for the graph is determined and constructed in the backend database automatically. For the automatic mode, we develop an RL-based storage structure selection algorithm, which will be introduced in Section 3.1. In the customized mode, April provides a friendly interface for users to determine the storage structure with the guidance of the system.

Index selection. This component is in charge of selecting proper indices for the graph. Similar as the storage structure selection component, April provides an automatic mode and a customized mode. In the automatic mode, the column to build index as well as the index type and parameters are determined automatically. The index is constructed according to the determination. The RL-based index selection algorithms will be introduced in Section 3.2. In the customized mode, an interface is provided for users to add the index, similar as the corresponding function in traditional databases.

Query optimization. Query optimization component converts the input SparQL queries into SQL with UDFs. To demonstrate the effectiveness of RL-based query optimization, we choose rule-based query optimizer [4] as the competitor. The RL-based query optimization strategy will be introduced in Section 3.3.

User interaction. April provides interfaces for users to input data, workload as well as the queries. For the demonstration, the users could compare the performance of the RL-based approaches in April with the competitors.

3 IMPLEMENTATIONS

In this section, we will introduce the RL-based algorithms in April. The decisions are made according to the data D and the workload W. Note that even though RL is adopted in all the components of

storage structure selection, index selection and query optimization, the algorithms are different. Since the queries could be various for the same data, and query optimization should be performed online, we choose off-policy, which optimizes the queries different from one that was used to train the model and could make decision online. As a comparison, the decision of storage structure and index selection could be made according to the data and workload themselves, and the time limit is not strict. Thus, we choose on-policy, which evaluates or improves the same policy that is used to make storage or indexing decisions.

3.1 Storage Structure Selection

Logically, all the edges of the graph are in the global table T, and the storage strategy aims to split T and merge some tables to make a table corresponding to some specific graph pattern. For example, the frequent subgraph matching pattern in the workload W is to query the students who were born in the same city as their advisors. To answer the query efficiently, we split the graph data whose edges are 'wasBornIn' and 'hasAcademicAdvisor' from T, and then merge them with the order of 'Student Name', 'Born City', 'Advisor Name', and 'Born City' as a new table. The determined storage is evaluated with the given workload.

State Space The state space is all possibilities of graph partition and merging. Each state is represented as a set of tables $\{T_1, T_2, \cdots, T_n\}$.

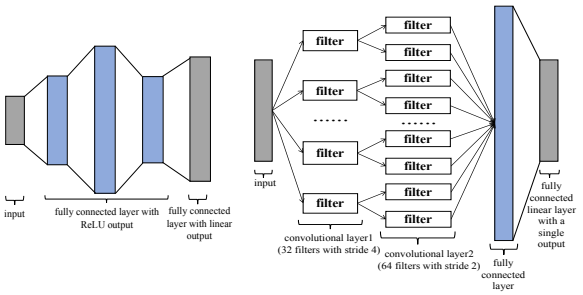
Action Space The action space includes table splitting and table merging. Table splitting splits a table according to some rules including separating the edges or subgraphs with the same pattern as a single table, and table merging joins tables on the vertex id, meaning connecting the subgraphs sharing the same vertex into one. Note that the actions are enumerable since the number of possible splitting and merging actions is $O(n^2)$, where n is the number of edges with different vertex and edge labels. With the consideration that in practice, it is useless to split the table with too few tuples, we could set a threshold of table splitting and enumerate the actions before learning.

Reward The reward is the benefits on the workload gained by the action, i.e. the evaluation cost of W on T minus that on the storage after the action.

Transition After an action a, a state s with table set T transits to another state s' by splitting a table in T or merging two tables in T.

Architecture and Training We develop deep Q-learning (DQN) with architecture in Figure 2(a) to solve this problem. To enhance the non-linear mapping ability, we add three hidden levels. Since the number of input dimensions is larger than that of the output, we increase the dimension of input to add features and then decrease it to the output for gradual dimension promotion. Even though deeper network may achieve better accuracy, for the efficiency issues, we design three hidden levels.

During the training, a concern is that the traditional DQN may over-estimate the Q values, and the error will increase with the number of actions [5]. If the over-estimation is not uniform, the Q value of an over-estimated suboptimal action will be larger than that of the optimal one, which misses the optimal strategy. To avoid such problem, we adopt double DQN [5] for more accurate Q values to ensure convergence.



(a) Storage Structure Selection (b) Query Optimization

Figure 2: Network Architectures

3.2 Index Selection

The goal is to determine the index for the columns to maximize the efficiency with a space constraint.

State Space The state space represents all possible states of the index construction. A state is a vector with each entry corresponding to a column in the storage. Each entry is an encoding which represents whether the index is built, whether it is a clustered index, index type, and index parameters.

Action Space The action space includes all actions that change the states of indices. Each action represents adding, deleting or changing an index.

Reward The reward is the gain of an action, which is measured by the difference of workload evaluation time before and after the action.

Transition When an index is added to some column, the clustered index of a table is changed, or the parameter of an index is changed, the current state is transitioned to the next state.

Architecture and Training Since the space is relatively small, to simplify the solution, we adopt Q-learning as the RL algorithm.

3.3 Query Optimization

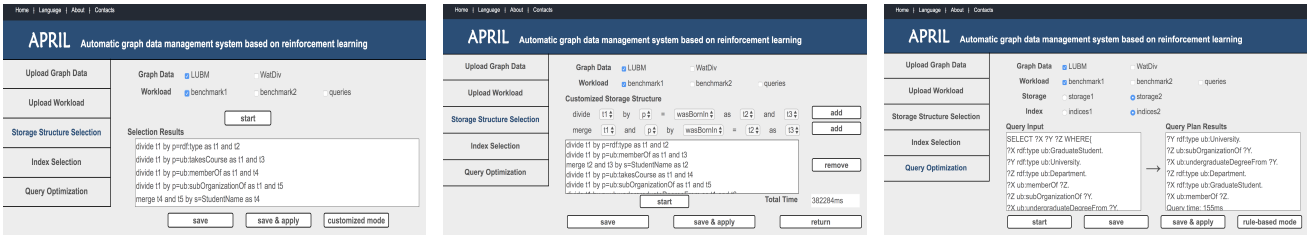
In our system, RL-based query optimization focuses on basic graph pattern (BGP) processing, which could be considered as a series of join operations on corresponding relations. Thus, the focus is the order selection for the join operations which minimizes the estimated execution time. We formulate the query optimization problem as follows.

State Space Given a BGP q=$\{t_1, t_2, \cdots, t_n\}$ corresponding to table sets $\mathbb{T}$=$\{T_1, T_2, \cdots, T_m\}$, where each t_i is a triple in a query, and each T_i corresponds to one or more connected tuples in the BGP, each state is an execution order of a subset of $\mathbb{T}$, denoted by a sequence $<T'_1, T'_2, \cdots, T'_m>$, where $T'_i \in \mathbb{T}$.

Action Space The agent can choose a table T_j that is not in the state as a sequence S to join with the partial results corresponding to S. Such join is an action.

Reward The minus of the estimated execution time of the state S is the reward.

Transition The term s' denotes the state that the agent reaches after executing action a at state s. Then the transition from s to s' is to append the table T_j corresponding to a to s.



(a) RL-Based Storage Structure Selection (b) Customized Storage Structure Selection (c) RL-Based Query Optimization

Figure 3: System Interfaces

Architecture and Training In order to reduce the number of training parameters, we use two deep convolutional neural networks (Q-network and target-network) with the same architecture except their parameters. Both networks are composed of an architecture shown in Figure 2(b). The input to the network is a vector representing the join state, in which a 1 entry represents the corresponding attribute including in the joined result and vice versa. The output of the network are Q values of all possible join operations, and the join operation with the largest Q is selected. The target-network parameters are only updated with the Q-network parameters every C steps and are held fixed between individual updates. Both networks are trained using the standard deep Q-network (DQN) algorithm with a simple fixed learning rate of 10^{-4}.

4 DEMONSTRATIONS

We intend to demonstrate the three graph data management functions in April with the visualization of each managing step, and show how the system works with graph data benchmarks.

Data sources. We use two graph data benchmarks, WatDiv[1] and LUBM[2], to demonstrate our system.

(1) *WatDiv data*. The data set is designed for measuring how an RDF data management system performs across a wide spectrum of SPARQL queries with varying structural characteristics and selectivity classes. It provides data generators and query templates. We manage the generated data and execute 20 queries on the data.

(2) *LUBM data*. The data set consists of a university domain ontology, customizable and repeatable synthetic data. We generate a data set with 850 universities and execute 14 queries on it.

Demonstration scenarios. April allows users to upload the graph data and workloads. After that, users choose the automatic mode or customized mode to select storage structure. As shown in Figure 3(a), in the automatic mode, April selects storage structure for the given graph data without any interaction with users. When the selection results are shown to users, users can decide to save the results only or apply the selected storage structure. In addition, if users are unsatisfied with the selected results, they can choose the customized mode.

As shown in Figure 3(b), users are allowed to input the user-specific storage structure in the customized mode. By adding the operations of dividing or merging tables, April obtains their customized storage plan. Then, the system will compute a total elapsed time of the given storage structure on the workload. If users are satisfied with the time results, they can choose to save the storage strategy only or apply it on the graph data. Also, April allows users to return to the automatic mode.

After the storage structure is determined, April allows users to choose the automatic mode or customized mode to select indices. In the automatic mode, the system produces the indexing strategy automatically. Users can choose to save the results only or apply the indices on the graph data. In addition, they are allowed to select the customized mode.

In the customized mode, users add their user-specific indexing strategies. By specifying the table name, the column name, and the index type, April obtains the customized indices and will produce a total elapsed time. If users satisfy the time results, they can choose to save the indexing strategy only or apply it on the graph data. Also, they are allowed to return to the automatic mode.

After the indices are created, April allows users to input their queries and choose the mode of query optimization. In the automatic mode shown in Figure 3(c), the system produces the query plan results without any interaction with users. Users can choose to save them only or apply them to the query processing. In addition, users are allowed to choose the rule-based mode. In this mode, April produces the query plans with the rule-based method. Users can choose their satisfied results according to the corresponding query processing time.

5 ACKNOWLEDGEMENTS

This paper was partially supported by NSFC grant U1866602, 61602129, 61772157, CCF-Huawei Database System Innovation Research Plan DBIR2019005B.

REFERENCES

[1] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *PVLDB* 8, 12 (2015), 1804–1815.
[2] Pedro Domingos. 2018. Machine learning for data management: problems and solutions. In *SIGMOD*. 629–629.
[3] Xi Liang, Aaron J Elmore, and Sanjay Krishnan. 2019. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint* (2019).
[4] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *ICDT*. 4–33.
[5] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*. 2094–2100.
[6] Tin Vu. 2019. Deep Query Optimization. In *SIGMOD*. 1856–1858.

[1] https://dsg.uwaterloo.ca/watdiv/
[2] http://swat.cse.lehigh.edu/projects/lubm/