Grep: A Graph Learning Based Database Partitioning System

XUANHE ZHOU, Tsinghua University, China
GUOLIANG LI, Tsinghua University, China (corresponding author)
JIANHUA FENG, Tsinghua University, China
LUYANG LIU, Huawei, China
WEI GUO, Huawei, China

Database partitioning is a fundamental but challenging task in distributed databases, which selects specific columns as a partitioning key for each table and uses the partitioning key to allocate the table data into different compute nodes in order to maximize the performance. However, this problem is NP-hard and existing distributed databases require users to manually specify the partitioning keys, which may cause potential performance degradation. Although reinforcement learning based methods have been proposed, they have several limitations. First, they do not capture the complex data distributions and query access patterns, and thus involve high computation cost across different compute nodes to answer a query. Second, they involve an expensive step to repetitively partition the data into different compute nodes in order to train a learned key-selection model, which is a waste of time and resources. To address these limitations, we propose a practical learned database partitioning system Grep. We first adopt a graph model to encode data and query features, where vertices are columns, edges are query relations, and the weights of columns are computed based on the localized graph structures (e.g., data diversity, joined columns). We then utilize graph neural networks to embed the partitioning factors into embedding vectors in order to capture the data and query correlations. Next we propose a key-selection model to select appropriate partitioning keys based on the graph model. Finally, we propose an evaluation model to estimate the partitioning performance without actually partitioning the database. We have implemented Grep in a commercial distributed database, and experiments show the effectiveness of our system (e.g., 68% higher throughput for 30K queries in a real banking scenario). The code is available at https://github.com/TsinghuaDatabaseGroup/AI4DBCode/DatabasePartition.

CCS Concepts: • **Information systems → Database management system engines**.

Additional Key Words and Phrases: database partitioning, graph neural networks

ACM Reference Format:
Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. 2023. Grep: A Graph Learning Based Database Partitioning System. *Proc. ACM Manag. Data* 1, 1, Article 94 (May 2023), 24 pages. https://doi.org/10.1145/3588948

1 INTRODUCTION

Nowadays, with the increase of data volume, many database customers are migrating their data from centralized databases to distributed databases (e.g., Snowflake [4], Azure Data Warehouse [2], Amazon Redshift [1]) to efficiently process massive data. For example, in a real banking scenario, there are over 20K tables and the biggest table has more than 50 billion tuples. We partition these tables into a distributed database with 60 nodes. The average running time of processing 30K

Authors' addresses: Xuanhe Zhou, Tsinghua University, China, zhouxuan19@mails.tsinghua.edu.cn; Guoliang Li, Tsinghua University, China (corresponding author), liguoliang@tsinghua.edu.cn; Jianhua Feng, Tsinghua University, China, fengjh@tsinghua.edu.cn; Luyang Liu, Huawei, China, liuluyang2@huawei.com; Wei Guo, Huawei, China, guowei@huawei.com.

2836-6573/2023/5-ART94 $15.00
https://doi.org/10.1145/3588948

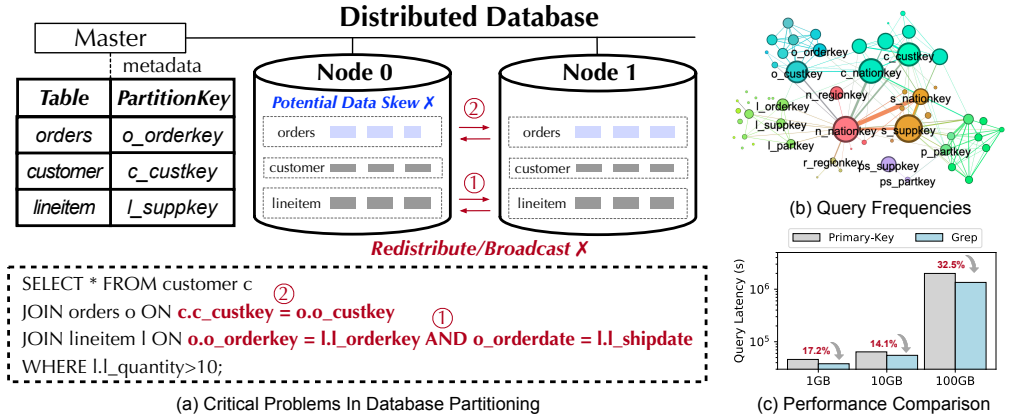


Fig. 1. Example Database Partitioning on TPC-H Schema.

queries is optimized to 0.7 second from 40 seconds. However, existing distributed databases rely on users to specify partitioning keys in order to allocate the data to multiple compute nodes, which is tedious and may not meet the performance objectives (e.g., fully utilizing CPU resources and gaining high throughput) [10, 20].

It calls for an automatic database partitioning method, which, given a collection of tables, first selects a single column (or multiple columns) as a partitioning key for each table, and then applies a given function (e.g., a modified version of consistent hash) on the values of this partitioning key, and distributes the tuples in each table by the partitioning values into different nodes.

However, the database partitioning problem is NP-hard (see Section 2.1). It requires to design effective algorithms to select appropriate partitioning keys. Existing methods [5, 10] rely on single primary keys for partitioning tables (e.g., o_orderkey for *orders*), which could be less effective in some scenarios (e.g., over 36% TPC-H queries and almost all the JOB queries join on non-foreign-key relations). As shown in Figure 1, primary-key partitioning can cause expensive tuple redistribution during the composite join between *lineitem* and *orders*. In contrast, selecting the composite join columns as partitioning keys can significantly improve the performance (e.g., 14.1%-32.5% latency reduction with different data sizes). Meanwhile, improper partitioning keys can result in imbalanced data distribution (*lineitem* table), which may cause data and query overloading on a few nodes, negatively affecting the throughput. Therefore, it is vital to design appropriate partitioning strategies based on both *the data distribution (e.g., tuple values) and query access patterns (e.g., the costs of joining different columns)* so as to optimize the performance.

Traditional heuristic methods [15, 44, 47, 59] do not consider the data distributions of different columns and correlations between data and query distributions, and cannot get good partitioning results. Deep reinforcement learning (DRL) based methods [21, 22] have two limitations. First, they neglect important data features (e.g., distinct values in columns) or query features (e.g., range queries, multiple joins). Second, in order to train a learned model, they require to evaluate the performance of many strategies by really partitioning the data. However, it is costly to really partition the databases and then evaluate the performance, e.g., taking 10 hours to distribute 1PB data to a two-node cluster if there is no obvious hardware bottleneck. Although [22] adopts a cost model to evaluate the performance, this cost model takes single queries as input and cannot encode *the data and query distribution under selected partition keys* and causes unreliable estimation. For example, the same partitioning keys may cause various performance with different node numbers.

Challenges. There are three challenges to design an effective database partitioning method. First, *how to efficiently capture the partitioning factors* (**C1**). There are various partitioning factors (e.g.,

columns, tuple values, query features) and they exhibit intricate correlations, while existing methods [15, 21, 22, 59] tend to only encode basic features (e.g., query template frequencies) and cause the loss of crucial information. Second, *how to effectively select partitioning keys to optimize the performance* (**C2**). We need to generate near-optimal partitioning keys on numerous data columns, which is an NP-hard problem. Third, *how to evaluate the partitioning performance* (**C3**). It is essential to evaluate the quality of chosen partitioning keys on sample data before actually partitioning the tables, which offers feedback for the partitioning-key selection model and substantially reduces partitioning overhead (e.g., preventing excessive resource consumption).

Our Proposed Methods. To address these challenges, we propose a database partitioning system `Grep` using graph neural networks. First, `Grep` builds a graph model to encode data and query features, where vertices are columns and edges are join relations. `Grep` prioritizes promising columns in the graph model with an attention mechanism (for **C1**). Second, as partitioning features scatter across the graph model, `Grep` utilizes a graph neural network to embed the global graph structures for each vertex in the graph, which approximate the partitioning benefits of different column combinations. We utilize a classifier to select partitioning keys (single or multiple columns) based on the beneficial column combinations (for **C2**). Third, to estimate the performance without actually partitioning the database, we propose an evaluation model using graph representation learning, which, given selected keys, generates a sample graph based on these keys and sampled tuples, and trains a graph-level GNN model to map the performance on samples to the performance on the whole dataset. Our evaluation model provides relatively accurate evaluation while taking much less time than actual partitioning (for **C3**).

Contributions: We make the following contributions.
(1) We propose a database partitioning system using graph neural networks (GNNs). To our best knowledge, this is the first work that uses GNNs to recommend partitioning keys (see Section 2).
(2) We propose a graph-based model to capture data and query features. And we utilize the attention mechanism to enhance partitioning by prioritizing these "important" columns (see Section 3).
(3) We propose a key-selection model that encodes the partitioning benefits for columns together with their joined columns, and accordingly select appropriate partitioning keys (see Section 4).
(4) We propose a deep evaluation model to estimate the partitioning performance without actual partitioning (see Section 5).
(5) We have implemented `Grep` into a commercial database. Experimental results on real customers (e.g., workloads in banking scenario) show that `Grep` can find high-performance partitioning keys and outperforms the state-of-the-art approaches (see Section 6).

2 PRELIMINARIES

2.1 Problem Formulation

Database Partitioning. When centralized database customers find the performance cannot meet their requirements due to a large volume of data, they will partition the database and migrate their data to a distributed database. Since most distributed databases mainly support horizontal partitioning, we only consider horizontal partitioning in line with existing studies [7, 11, 21].

Definition 1 (Database Partitioning). *Given tables $\{T_1, T_2, ..., T_m\}$ and a partitioning function $\mathcal{F}$, for each table T_i with columns $\{c_1^{(i)}, c_2^{(i)}, ..., c_n^{(i)}\}$, database partitioning selects some columns as the partitioning key of T_i and allocates the tuples in T_i into different nodes using $\mathcal{F}$, such that the performance of a given set of SQL queries (e.g., total latency, average throughput) is optimal.*

Example 1. *Considering an example of database partitioning on two compute nodes P_0 and P_1. We use l_orderkey as the partitioning key of lineitem, and one tuple is allocated into P_0 and three tuples are allocated into P_1. Instead, if we partition lineitem with l_suppkey, two tuples are allocated into P_0*

and two tuples are allocated into P_1 and the data is well-balanced. The partitioning key of supplier is s_suppkey, and there are two remote joins between nodes P_0 and P_1. Instead, when we partition the two tables by the joined columns, there is no remote join and we gain over 80% latency reduction.

In real-world scenarios, databases may have numerous columns (e.g., millions of columns in a commercial database) and the problem gets hard for two reasons. First, the solution space is large, and traditional heuristic methods may not find a good partitioning strategy. Second, some partitioning keys may contain multiple columns. For example, suppose an *Order* table has 2 years of data. Given a query requesting *orders* for a week, if we only use the column `year` as the partitioning key, this query will access all the data; but if we use both the `year` and `week` as the composite partitioning key, the query will only access a week of data.

Column Graph. We use a *column graph* to characterize the partitioning factors, where vertices are columns and there is an edge between two columns if they are joined by some queries.

Complexity Analysis. With the *column graph*, we can prove that the database partitioning problem is NP-hard. For simplicity, we consider a simplified case: *(1) the partitioning key contains a single column (no composite partitioning key with multiple columns); (2) the cost of remote joins across different nodes is much more expensive than that of local joins within the same node.* We assign each edge with a weight, which is the remote join cost on the two columns for a given set of SQL queries, in the case that the corresponding two tables are partitioned based on the two columns. For any two unconnected vertices on the *column graph*, we add a virtual edge with a weight of 0, because there is no query joining the two columns. Then the database partitioning problem aims to find a maximum-weight k-clique from the graph (taking the columns in the clique as partitioning keys) in order to minimize the remote join cost, which has been proven to be NP-hard [8]. Thus the database partitioning problem is also NP-hard.

Theorem 1. *The database partitioning problem is NP-hard.*

Remark. (1) The single-column partitioning problem is a special case of the multi-column partitioning problem, and thus the latter is also NP-hard. (2) We use hashing partitioning functions, which are most widely adopted in real scenarios [1–3]. (3) SQL queries are usually available in scenarios like migrating data from one database to another, which is especially common for distributed database clusters in both cloud and traditional business environments.

2.2 System Overview

To address the database partitioning problem by judiciously selecting partitioning keys, we propose a graph-embedding based partitioning system (`Grep`). Figure 2 demonstrates the architecture of `Grep`, which includes three main modules, i.e., *Column2Graph*, *Partitioning Model*, *Evaluation Model*.

2.2.1 ***Partitioning Workflow.*** (1) Given a database, `Grep` first trains an *Evaluation Model*, which predicts the performance of a partitioning strategy. (2) `Grep` builds the *Column2Graph Model* based on data and query features. With the graph model, `Grep` trains a *Partitioning Model*, which is used to select partitioning keys. (3) `Grep` uses the *Evaluation Model* to evaluate the selected keys and gives feedback to the *Column2Graph Model* (updating the vertex weights) and *Partitioning Model* (updating the GNN weights). (4) `Grep` repeats steps 2-3 until convergence and reports the selected partitioning keys.

2.2.2 ***Column2Graph.*** *Column2Graph* collects and characterizes behaviors of queries on the columns in the form of a graph model.

Edge Weight. For any edge $E(c_i, c_j)$, the edge weight $W(c_i, c_j)$ denotes the cost of executing the join predicate $c_i = c_j$. There are two main factors that affect join cost: (i) Query frequency: the number

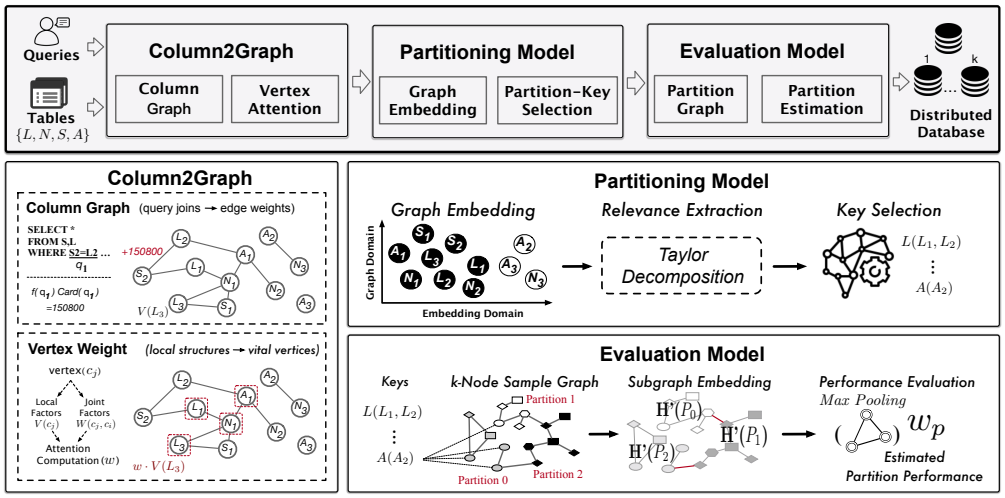


Fig. 2. The Grep Overview.

of queries that contain the join predicate $c_i = c_j$; (ii) Cardinality: the number of tuples in the result set. To estimate query cardinalities, we sample tuples from the tables, execute predicates on the sampled tuples, and use the results on the samples to estimate the cardinality (see Figure 2). Hence, we compute edge weight as the multiplication of query frequency and estimated cardinality, which represents the overall cost of $c_i = c_j$. For example, in Figure 2, the weight of joining `l_orderkey`, `c_nationkey` equals $1 \times 150800 = 150800$, with one query involving this join and producing 150800 result tuples. Note the join costs are also affected by other query operations, e.g., filters on the join tables, which we need to characterize in the vertex features (see Section 3.1).

Vertex Weight. The column graph may contain numerous vertices, and if we enumerate every column combination to select partitioning keys, it will cause heavy computation overhead. Instead, we utilize the localized graph structures (e.g., read/write frequency, join costs) of each vertex to compute the vertex weights, identify important vertices with the vertex weights, and prioritize important vertices as partitioning keys. Thus, the *Partitioning Model* takes fewer iterations to select high-quality keys. For example, in Figure 2, column A_3 is rarely accessed and is not joined, and we assign A_3 a small weight value, representing a low possibility of being selected.

2.2.3 ***Partitioning Model.*** There are two challenges in selecting columns as partitioning keys. First, considering a graph with *millions of columns in a commercial database*, traditional linear regression methods cannot efficiently process the graph structures and cause significant information loss. Second, the problem of partitioning-key selection is NP-hard (Section 2.1), and existing learned methods (e.g., RL) require significant system resources to adapt to complex scenarios (e.g., using 10 machines for 7 days to run). To address these problems, *Partitioning Model* utilizes a graph neural network to embed column features and join patterns into vectors, and selects partitioning keys based on these vectors.

Graph Embedding. We first encode the global graph structures for each vertex c_i into an embedding vector $H(c_i)$. We compute $H(c_i)$ iteratively. Suppose the embedding vector of c_i at the t-th iteration is $H_t(c_i)$ and the edge weight matrix is W where $W(c_j, c_i)$ is the edge weight of (c_j, c_i). We utilize the graph neural network (GNN) to compute the embedding vector of c_i at the $(t+1)$-th iteration based on (i) the embedding vector $H_t(c_i)$, (ii) the embedding vectors of c_i's neighbors $\{c_j | \forall (c_j, c_i) \in E\}$, and (iii) the edge weights W. GNN iteratively updates the network weight ω to learn the embedded vertex vectors with maximal partitioning benefits.

Partitioning-Key Selection. Intuitively, we can directly append a classifier to select keys based on the embedding vectors. However, we observe the embedding vectors are of high dimension (e.g., $256 \times vertex\_number$) and we only need a small part of information (e.g., importance to each embedding vector) to find promising partitioning keys. Thus, we first estimate the overall partitioning benefits of each column with Taylor decomposition [38], which reversely computes the proportion of the columns in the embedding vectors (see Section 4.3) ; then we append a binary classifier that inputs the column relevance and outputs selected columns, where 1 represents the column is selected and 0 otherwise.

Example 2. *In Figure 2, $\{N_1, L_3, S_1, L_1, A_1, S_2, L_2, N_2\}$ forms a connected subgraph of N_1 and part of them are embedded into a vector $H(N_1)$. With Taylor decomposition, we find that A_1 has high contribution to $H(N_1)$, because $E(A_1, N_1)$ is large and can avoid many remote joins; and the contribution of L_2 is low, because L_2 joins with N_1 via A_1 and $E(L_2, A_1)$ is relatively small.*

Remark. Our method can support replicate tables (DBAs generate replicate small tables with fewer than 500K tuples): if there is no selected partitioning key for a table, this table will be replicated.

2.2.4 ***Evaluation Model.*** To well train the above learned models in an end-to-end style, we need to know the performance of selected partitioning keys. However, partitioning the database is costly and simple metrics (e.g., deviation of partition size, the scale of largest partition) cannot get accurate estimation. Hence, we propose an *Evaluation Model* to estimate the partitioning performance.

k-Node Sample Graph. We first sample tuples from tables and allocate the tuples into k compute nodes based on the selected keys. Then we add edges based on the join predicates and generate a *k-node sample graph* (see Section 5.1), which captures the behaviors of queries on the k nodes. We empirically decide a suitable sample rate to tradeoff the evaluation quality and efficiency [11].

Performance Evaluation. Next we estimate the performance based on the *k-node sample graph*. (i) For each node P_v, we encode the performance-related features (e.g., local joins, local filters/writes) into an embedding vector $H(P_v)$, which represents the costs of local queries in P_v. (ii) Next we use the k embedding vectors $\{H(P_1), H(P_2), ..., H(P_k)\}$ of the k nodes to generate a k-vertex graph G^P, where vertices are the nodes and there is an edge between two vertices P_i and P_j in G^P if there exists an edge $(v_i \in P_i, v_j \in P_j) \in G$. We pool the k embedding vectors of nodes into an embedding vector of the graph G^P, denoted as $H(P)$ [61]. For example, in Figure 1, there are 2 remote joins across 2 nodes, which separately cause the redistribution of the *lineitem* and *order* tables and have the highest execution costs. We extract the maximal values in each dimension of $H(P)$ (the features of nodes and joins) to capture remote-join costs. (iii) Then we use a fully-connected (FC) network to map the embedding vector $H(P)$ into desired performance metrics with non-linear transformations.

Model Training. We supervisedly train the *Evaluation Model* with a small number of partitioning scenarios and generalize the *Evaluation Model* to other scenarios. First, we obtain training data D in two ways: (i) Collect from real query logs, which record either daily transactions or costly queries; (ii) Since partitioning keys are rarely changed in real scenarios, we simulate typical partitioning strategies on sample datasets. Second, with every training sample in the form of $\langle Q^d, D^d, C^d, P^d \rangle$, where Q^d denotes queries, D^d denotes dataset, C^d is the selected columns, and P^d is the performance metrics like execution time, we generate a k-node sample graph based on $\langle Q^d, D^d, C^d \rangle$, utilize the *Evaluation Model* to estimate the performance, and update network weights with the loss of estimated/actual performance.

3 COLUMN GRAPH MODEL

We present how to build the graph model to capture the data and query features. We first formally define the model in Section 3.1. We then explain how to compute the edge weights in Section 3.2 and the vertex weights in Section 3.3.

3.1 Column Graph

Existing methods are table-based and cannot capture column correlations, which is vital to database partitioning. Hence, we model the columns and their correlations as a graph, where the vertices are columns and there is an edge between two columns if they are joined in some queries or have foreign key relationships. Note that we mainly support SPJ queries here, and users can extend the *column graph* so as to support more complex queries (e.g., nested subqueries, UDFs).

Definition 2 (Column Graph). *Given a set of tables* $\{T_1, T_2, ..., T_{|T|}\}$ *and queries* $\{Q_1, Q_2, ..., Q_{|Q|}\}$*, we model the table columns and their correlations into a graph model. The vertices are columns used in the queries and there is an edge between two columns if they are joined in the queries or have foreign key relationships.*

Vertex Feature Vector. Each vertex contains the main data and query features of the corresponding column. We encode the features of the vertex as a vector, which contains 7 dimensions:
(1) **table id:** the table of this column;
(2) **table size:** the size of the table of this column;
(3) **tuple selectivity:** the rate of distinct values in the column, i.e., $\frac{\#-distinctValues}{\#-tuples}$;
(4) **tuple length:** the maximum tuple length in this table;
(5) #**filters:** the number of filter operations w.r.t. the column;
(6) #**aggregates:** the number of aggregates w.r.t. the column;
(7) #**writes:** the number of write operations w.r.t the column;
Note *table id* is only used to distinguish the tables (e.g., whether different columns are from the same table), and it does not need to be learned and transferred. Thus, for simplicity we denote $V(c_j)$ as the 6 features of a column c_j except *table id*.

Vertex Weight Computation. We utilize an attention mechanism to compute vertex weights for two reasons: (1) The *column graph* is of large scale and causes low efficiency by equally processing these vertices; (2) The vertex does not include the edge relations, which are vital to partitioning benefits. For example, `p_name` and `p_partkey` are from the same table, but `p_name` has one joined column, while `p_parkey` has seven directly joined columns and can potentially avoid more remote joins by partitioning the joined columns. To this end, we propose a deep attention model to compute the importance $a(c_j)$ of a vertex c_j(Section 3.3). Given a vertex c_j, we multiply the importance $a(c_j)$ with the vector $V(c_j)$ and gain a weighted vertex vector, i.e., $\tilde{V}(c_j) = a(c_j)\ V(c_j)$.

Edge Feature Vector. The edge features of an edge $E(u, v)$ should capture the join cost. To this end, we encode the edge features as an edge vector $E(u, v) = (V(u), V(v), W(u, v))$, where $V(u)$ and $V(v)$ are vertex vectors and $W(u, v)$ is the edge weight of (u, v). For example, in Figure 2, the edge vector $E(C_2, L_2)$ is denoted as $(V(C_2), V(L_2), 150800)$, where $V(C_2)$, $V(L_2)$ denote the vertex feature vectors and 150800 equals the estimated cost of $C_2 = L_2$.

3.2 Predicate-based Edge Weight

We consider two main features to compute the edge weights:

(1) Frequencies of join predicates. The frequency of a join predicate between two columns is the number of join predicates containing the columns, which is important to database partitioning, because if two columns are frequently joined, we can reduce more remote joins by using the two columns as partitioning keys.

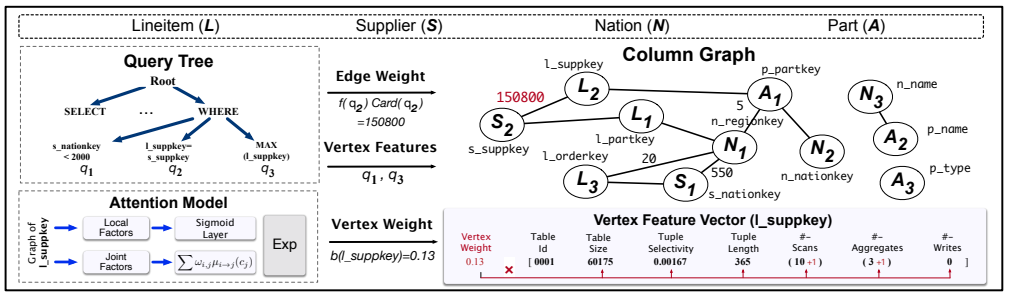


Fig. 3. Column Graph Construction. Note we showcase a common transactional query with (i) a two-table join; (ii) an atomic predicate; (iii) a simple aggregate operation.

(2) Cardinalities of join predicates. Cardinality estimates the result set size and can reflect query cost [51]. Note it is troublesome to apply existing estimation methods [48], because they rely on query plans, which cannot be obtained before partitioning. Hence, we sample tuples from the tables, and estimate cardinalities by executing predicates on the sampled tuples. For example, for the three predicates in Figure 2, we sample 1% tuples from tables `lineitem` and `customer` and execute predicates on the sampled tuples in the order of $q_1 > q_2 > q_3$, where the result set size of q_2 equals 1508. So we approximate the join cost of q_2 as 1508/1% = 150800. Note we use stratified sampling [18] to reflect data distribution with limited tuples and we can estimate any query predicates with multi-joins.

Edge Weight Computation. For any edge (u, v), we compute the edge weight by summing up the estimated costs of all the joins between u and v, i.e., $\mathrm{W}(u, v) = \sum_q f(q)Card(u, v, q)$, where q is any join predicate between u and v, $f(q)$ is the occurrence times, and $Card(u, v, q)$ is the estimated cardinality.

3.3 Attention-based Vertex Weight

There may be numerous columns in the database and we want to *prioritize vertices that are important to database partitioning*, which can greatly reduce the partitioning overhead. There are two challenges in computing the column importance. First, we may have different performance objectives (e.g., higher throughput, lower latency) based on the user and application requirements, but traditional importance ranking methods [16, 17, 34] like PageRank cannot dynamically compute the importance based on the targets. Second, the Long Short Term Memory networks (LSTMs) in NLP mainly consider semantic relations among words in a sentence, which commonly contains 15–20 words, while our graph model may have hundreds of thousands of columns and numerous column orderings and the relations are much more complex.

To address these challenges, we propose a graph-based attention network to evaluate the column importance, with several salient features. First, the attention model can get the importance of each vertex and enables `Grep` to prioritize important vertices. Second, the attention model can better reflect performance requirements by learning different encodings of the graph features.

Attention Network. Our attention network is composed of a dense layer, which maps the *column graph* G into attention values $\{a(c_j)\}$. The procedure can be written as $V(G) = \sum_{c_j \in G} a(c_j)V(c_j)$, where $V(c_j)$ is the vertex feature vector and $a(c_j)$ is the assigned attention to find the optimal partitioning strategy. Next we explain how to compute attentions $\{a(c_j)\}$ to infer column importance.

Attention Computation. An important column usually satisfies at least one of the two conditions: (i) Values in the column are evenly distributed and will not lead to data skew; (ii) The column is frequently queried or joined with other columns. So for each column c_j, we compute its attention value based on two kinds of graph features: (i) *Vertex features* describe the column characters,

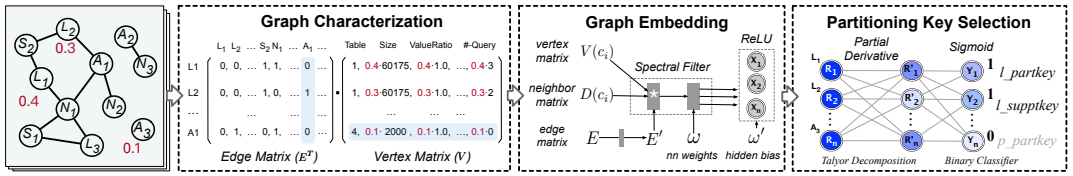


Fig. 4. Graph-based Key Selection Model.

including data features (e.g., table size, distinct values) and query features (e.g., selects, aggregates). Since the vertex features (see Definition 2) are from different feature spaces and have different value ranges, we use an activation function to normalize these features, i.e., $\psi(c_j) = Sigmoid(V(c_j))$, where $Sigmoid$ on any feature x equals $\frac{1}{1+e^{-x}}$; (ii) *Edge features* describe the overhead of remotely joining a column with other columns, which can be reduced by selecting them as partitioning keys. We compute edge features by aggregating the joined edges of each vertex. Based on the message-passing techniques [52], for any vertex c_j, we compute the edge feature vector as $\mu_{i\to j}(c_j) = W(c_i, c_j)\psi(c_i)\psi(c_j)$, where c_i is any adjacent vertex of c_j and $W(c_i, c_j)$ is the edge weight.

For any vertex c_j, we compute attention $a(c_j)$ by enumerating vertex/edge features and utilize the attention network to encode these features into a scalar value based on the partition targets,

$$a(c_j) \propto exp\left(\hat{\omega}_{jj}\psi_{c_j} + \omega_{jj}\mu_{j\to j}(c_j) + \sum_{c_i} \omega_{ij}\mu_{i\to j}(c_j)\right) \tag{1}$$

where ω_{ij} is the network weight, and $\hat{\omega}_{jj}$ is the unit vector of w_{jj}. The first two terms are the vertex features and the last item denotes the edge features. We use ω_{ij} of the attention network to learn the importance of c_i to the partition targets (e.g., increased throughput).

Model Training. In each iteration, the attention model computes the vertex weights in the *column graph*, the *Partitioning Model* selects partitioning keys based on the graph, and we update the network weights w_{ij} based on the performance feedback via gradient descent. The model converges until the performance satisfies the partition requirements or arrives the maximum iteration number.

4 PARTITIONING KEY SELECTION

Next we study how to select partitioning keys using the graph model. We first introduce the architecture of our *Partitioning Model* in Section 4.1. Then we explain the graph embedding algorithm in Section 4.2 and the key selection algorithm in Section 4.3. We present model training in Section 4.4.

4.1 Key Selection Model

The *Partitioning Model* takes as input the graph model in the form of vertex matrix and edge matrix and outputs selected columns. The idea is to first embed each column into an embedding vector, then compute the benefit of each column based on all the embedding vectors, and finally select partitioning keys based on the benefits. The framework of our *Partitioning Model* includes two components.

Graph Embedding aims to encode each column c_i and their joined columns into an embedding vector $H(c_i)$ that maximizes the partitioning benefits. The intuitions are (i) we rely on data and query features to select partitioning keys and (ii) column features and their join patterns are hard to capture with traditional heuristic methods like random walk [56], which may cause information loss. Hence, for each column c_i, we denote the graph structures as $G(c_i) = (V(c_i), D(c_i), E)$, where $V(c_i)$ is the column feature vector (Section 3.1), $D(c_i)$ is the neighborhood matrix with each row representing the feature vector of a joined column of c_i , and E is the edge matrix. And we need to embed these three factors into an embedding vector $H(c_i)$. To this end, our *Graph Neural Network* includes a graph embedding layer and an activation function *ReLU*: based on the performance objectives, the graph embedding layer encodes the join patterns via graph convolution; and the *ReLU* layer converts the embedded columns into a fixed-length vector.

Partitioning-Key Selection aims to select partitioning keys based on the embedding vectors. Although *Graph Neural Network* has extracted graph features into the embedded vertex vectors, it is still hard to directly select columns from the embedding vectors, which are high dimensional and require further training to capture correlations. Instead, we first utilize Taylor decomposition to reversely extract the relevance of each column to the embeddings in the *Graph Neural Network*; and then select keys based on the relevance distribution, which denotes the partitioning benefits to all the embeddings. Note we have conducted ablation experiments to verify the effectiveness of Taylor decomposition, which converges much faster and achieves higher performance within limited resource usage (see the experimental report[1]). To this end, this module includes two parts: (i) Taylor decomposition: It extracts benefits of each column from the embedding vectors, which is similar to backward propagation for training neural networks, but we compute the benefits as the partial derivative of the embeddings on the corresponding column. (ii) Binary classification: It inputs the partitioning benefits of all the columns and outputs the selection keys with multinomial logistic regression, where 1 denotes a column is selected and 0 otherwise.

4.2 Graph Embedding Algorithm

For every column c_i, we obtain local graph structures, i.e., $\{(c_j, c_i)|\forall(c_j, c_i) \in E\}$, and utilize the graph neural network (GNN) to embed the graph structures and the vertex vector of c_i into an embedding vector $H(c_i)$ by multiplying the input vertex features (joined columns) with the GNN weights ω. To generate embedding vectors that maximize the partitioning benefits, we iteratively update the GNN weights ω based on the performance objectives. For iteration $t+1$, the embedding procedure can be denoted as

$$H_{t+1}(c_i) = \sigma(H_t(c_i), H_t(c_j|\forall(c_j, c_i) \in E), W, \omega), \tag{2}$$

where $H_t(c_i)$ is the embedding vector of column c_i in iteration t, W is the edge weight matrix, ω is the GNN weight, and σ is the activation function. Next we further explain how to compute the embedding vectors in three steps.

Step 1 - Represent Graph Structure. We first represent the structural information of each column c_i as $G(c_i) = (V(c_i), D(c_i), E)$, where ($i$) we extract column and query features and encode them into a vertex vector $V(c_i)$ (Section 3.1); (ii) we use the edge matrix E to denote the graph structure (Section 3.2); (iii) the neighborhood matrix of c_i, $D(c_i)$, is computed by $E(c_i)^T \cdot V$ based on the joined columns of c_i.

Step 2 - Embed Neighbor Information. Next we embed features within $G(c_i)$ with graph convolutions. (i) We add self-connection relationships into the edge matrix E, because the column features of itself are most important to partition benefits (e.g., tuple selectivity). We normalize values in the edge matrix E into $[0, 1)$ and then sum the identity matrix I_n with the edge matrix, i.e., $E' = E + I$, where $I(i, j) = 1$ if $i = j$ and $I(i, j) = 0$ if $i \neq j$. (ii) And then we embed join features into the vertex vector. We take E' as a spectral filter on $V(c_i)$, i.e., $E' * V(c_i) \approx D(c_i)^{-\frac{1}{2}} E' D(c_i)^{-\frac{1}{2}} V(c_i)$. ($iii$) Next we propagate the embedded vertices across the graph neural network, which assigns the network weights ω to the vertices and outputs an embedding vector $H(c_i)$. We iteratively update the network weights to maximize the partitioning benefits. To enhance the embedding efficiency, we approximate $\omega \cdot H(v_i)$ with a truncated expansion by Chebyshev polynomials $T_z(H)$ [25, 65], i.e., $\omega H \approx \sum_{z=0} \theta'^z T_z(H)$, where H is the embedded vertex vector, θ'^z is the z-th weight value, and the Chebyshev polynomials $T_z(H)$ is recursively defined as $T_z(x) = 2T_{z-1}(x) - T_{z-2}(x)$ ($T_0(x) = 1$, $T_1(x) = x$).

[1] https://github.com/TsinghuaDatabaseGroup/AI4DBCode/DatabasePartition

Step 3 - Embed into Fixed-Length Vector. Finally we apply an activation function to convert the embedded features into a fixed-length vector. It has two benefits: (i) Activation function conducts non-linear transformation and can derive partitioning features from the embedded graph features, most of which are unrelated to performance objectives; (ii) It is easier to generalize a well-trained model to different graphs with fixed-length vectors. To this end, we use ReLU, an activation function in the form of $\max(L_1(h_i), 0)$, where h_i is an embedded feature and L_1 denotes the L1 norm, to introduce non-linearity transformations from spectral domain to embedding domain. For example, suppose higher throughput is expected, ideally queries are distributed based on the accessed/joined columns and there are no remote joins across different nodes. Hence, *ReLU* decreases weight penalties on the vertices with high degrees, which are more likely to be queried and may reduce more remote joins.

4.3 Key Selection Algorithm

Next we select partitioning keys from the embedding vectors, which encode local graph structures that maximize partitioning benefits for every vertex. We compute the relevance of each column to the embedding vectors (overall partitioning benefits), and select partitioning keys from the relevance distribution.

Taylor Decomposition. The first problem is how to compute the proportion of any column in all the embedding vectors? For an embedding vector $H(c_i)$, we only know that the vector comes from c_i together with some joined columns and represents potential benefits using c_i, but which columns are encoded and how much they contribute to the embedding vector are unclear, because the learning of network weights acts like *"black box"* [27] and it is hard to obtain the relations between embeddings.

To address this issue, we utilize Taylor decomposition [6, 38] to compute the relevance of each column c_i to all the embedding vectors in the graph neural network reversely and aggregate them as the overall partitioning benefit $R(c_i)$. Taking the embedding vectors as $X = \{x_1, x_2, ..., x_n\}$, where n is the column number, and the total relevance of the embedding vectors as $R = \{R_1, R_2, ..., R_n\}$, we reversely compute the relevance of every neural unit in the GNN based on the layer-wise propagation rule. (i) In the last layer L, we denote the relevance of each neural unit as 1, i.e., $\{R_i^L = R_i = 1 | \forall i \in [1, n]\}$, which denotes that the relevance has not been distributed. (ii) For any other layer l, we denote the relevance passing from layer (l+1) to layer l as $R_i^{(l+1)} = \sum_{j \in layer\ l} R_{i \to j}^{(l+1,l)}$, where $R_{i \to j}^{(l+1,l)}$ represents the relevance passing from neural unit $(l+1, i)$ in layer $l+1$ to neural unit (l, j) in layer l. Hence, we can take $R_i^{(l+1)}$ as a continuous and derivable function and compute the partial-derivative of $R_i^{(l+1)}$ in neural unit (l, j) as the relevance, i.e.,

$$R_j^l(x) = \sum_{i \in layer\ l+1} \frac{\partial R_i^{(l+1)}}{\partial x_j}|_{x_j} \cdot (x_j - \hat{x}_j) \tag{3}$$

where $\hat{x}$ denotes the "background" features that do not affect partitioning. $(x_i - \hat{x}_i)$ represents the changes in the embedding before/after the neural unit (l, j). Here we take $\hat{x}$ as 0-value vectors which have equal size to the vertex vectors x. This equation defines the propagation rule of relevance from layer $(l+1)$ to layer l.

By following this layer-wise propagation rule from the output to the input layer, we redistribute the relevance of all the embedding vectors onto input columns, which we take as the overall partitioning benefits, i.e., $\{R(c_1), R(c_2), ..., R(c_n)\}$. High benefit reflects the column is promising to reduce more global scans and remote joins.

Partitioning Key Selection. After obtaining partitioning benefits of each column, i.e., $\{R(c_i)\}$, we use multinomial logistic regression, a binary classification method, to predict the probability of

using a column c_j as the key, i.e., $\mathcal{P}r(R(c_k)) = \frac{e^{\beta_k R(c_k)}}{1+\sum_i e^{\beta_k R(c_k)}}$, where $\{\beta_j\}$ are learnable weights that simulate the joint probability distribution when using different columns. It inputs $\{R(c_1), R(c_2), ..., R(c_n)\}$ and outputs 0 ($\mathcal{P}r < 0.5$)/1 ($\mathcal{P}r > 0.5$) for each column, where 1 represents c_j is selected and 0 otherwise. For example, in Figure 4, S_2 is selected as S_2 is encoded into the embedding vectors of both S_2 and the two adjacent columns $L_{1,2}$ with high relevance.

4.4 Model Training

Training Data. The training data is a set of quadruples $\langle Q^d, D^d, C^d, P^d \rangle$, where Q^d is a set of queries, D^d is a set of tables, C^d is the set of columns, and P^d is the performance metrics. When the real performance is not available before partition, we use the *Evaluation Model* to estimate the performance (see Section 5).

Training Procedure. We iteratively train the GNN model (Taylor decomposition is a classic technique and does not need training) with different combinations of queries and data. In each iteration, we first sample 60% generated predicates as the queries and generate tables whose data size is selected within 10-100GB. And then we build the *column graph*, select columns from the *column graph*, and gain the estimated performance from the *Evaluation Model* in real time. Next we compute the performance changes, i.e., $\Delta v = C_1 \frac{T_t - T_0}{T_t} + C_2 \frac{L_0 - L_t}{L_0}$, where T_t and L_t denote the query throughput and latency with selected columns as partitioning keys, and T_0 and L_0 denote the query throughput and latency on non-partitioned tables, and C_1 and C_2 are the coefficients that are set based on user requirements. Then we update the network weights $\{\omega_{ij}\}$ via gradient descent, i.e., $\Delta\omega_{ij} = \mu/(\Delta v + \varepsilon)$, where μ controls the update rate and ε is a small number like 1×10^{-6}. This equation explores better partitioning keys towards the optimal direction.

We terminate the training when the performance converges or we have arrived the maximum iteration number. Then we test the performance on the test set. If the performance improvement is not good enough, we continue to train the model on this training set; otherwise, the model is converged on this workload and we enter next iteration or complete training. Note that we adopt batch gradient descent to enhance training efficiency, which selects partitioning keys for several workloads together, and computes the gradient and updates the GNN network weights in batch.

5 PERFORMANCE EVALUATION MODEL

We first introduce the *Evaluation Model* in Section 5.1. Then we explain the procedure of model training in Section 5.2.

5.1 Evaluation Model

There are two challenges in performance estimation. First, it is expensive to actually partition the database every time when selecting new partitioning keys. Second, existing performance evaluation models cannot effectively estimate the performance [19, 22, 48], because (*i*) they rely on query plans from optimizers, which are not available before partition; and (*ii*) they are not aware of data/query distributions across nodes, which are critical on a partitioned database. To address these challenges, we first generate a k-node sample graph, where the vertices are sampled tuples and the edges are join correlations between tuples, and k denotes the node number. Then we utilize graph learning to estimate the partitioning performance using the k-node sample graph.

k-Node Sample Graph. To model a partitioning strategy, we first sample tuples from tables using stratified sampling, which can reflect the data distribution with limited tuples. And then we allocate the sampled tuples into k nodes based on the selected keys and hash partitioning function. For example, for the sampled tables *supplier* and *lineitem* in Figure **??**, if we partition on their primary columns (a partitioning strategy), 5 tuples are allocated in node P_0 and 3 tuples in node P_1. Next, we compute the edge weights (join frequencies) and vertex weights as discussed in Section 3.

Given a subgraph, the vertex features are the weighted frequencies of local operations (i.e., filters, aggregates, writes), and the edges are the join frequencies across tuples in the same node. Thus, we generate a *k-Node Sample Graph* that represents the query and data distribution on k nodes. Note that the execution cost of local joins in a node and that across nodes can be different and we explain how to use graph embedding to learn the different costs with trainable network weights.

Performance Evaluation. There are two challenges in evaluating the performance based on the *k-node sample graph*. First, edges between different nodes are more expensive than edges within the same node, and we separately learn the actual weights of edges in the same node and edges across different nodes. Second, we cannot directly map the *k-node sample graphs* into query performance, which are of non-linear relations and can be affected by factors like configurations. Hence, we first embed the *k-node sample graph* into an embedding vector; and then use deep learning to map the performance on the sampled tuples into that on the whole datasets.

Step 1 - Representation of Single Node. We separately aggregate the features within the subgraph of each node (e.g., local joins/selects/writes) and embed them into k embedding vectors $H(P_i)$ by multiplying the edge weight matrix, vertex matrix of each node with the globally-shared graph network weights, which represent the data scales (e.g, the number of vertices) and computation costs of local operations (e.g., the edges of vertices in the same subgraph).

Step 2 - Representation of Multiple Nodes. Next we generate a k-vertex graph, which is in the form of a compound vertex matrix $\widetilde{V} = [H(P_1), H(P_2), ..., H(P_k)]$ and an edge matrix $\widetilde{E}$. (i) We first conduct convolution on $\widetilde{V}$ and $\widetilde{E}$ to embrace remote join relations into partitioning vectors, denoted as $H(P) = \widetilde{E} * \widetilde{V} \cdot \widetilde{\omega}$, where $\widetilde{\omega}$ is the network weight. (ii) We conduct *max pooling* on $H(P)$ to pool up the partitioning vectors into a graph vector, i.e., taking the maximum values in each feature of $H(P)$, to represent the overall query costs within the *k-node sample graph*. The reason of using *max pooling* is the query latency on nodes depends on the slowest multiple joins and correlated local select operations.

Step 3 - Query Performance Mapping. The graph vector encodes features within the *k-node sample graph*, which contains similar data and query distribution as that on the whole datasets. Hence, we further use a fully-connected (FC) layer, with ReLU as the activation function, to map the graph vector on sampled tuples into the partitioning performance on the whole dataset. The FC layer aggregates the features of the graph vector with non-linear transformations and approximates performance metrics by learning the network weights based on deviation from the actual performance.

Remark. *Evaluation Model* supports the case k is changed, because our model reflects partitioning differences in the graph, i.e., data/query allocation across nodes, and can adapt to structural changes by pooling the k node vectors and finetuning the graph embedding weights (e.g., joins cause changes in the edge weight matrix).

5.2 Training for Evaluation Model

We first train the *Evaluation Model* with some partitioning strategies extracted from the real-world system logs and simulated on sample datasets (offline training); and then we utilize the *Evaluation Model* to estimate the performance of *Partitioning Model* on the selected partitioning keys. After partitioning the database, we use the real performance to fine-tune *Evaluation Model* (online training).

Offline Training. We first train the *Evaluation Model* with some partitioning strategies and generalize it to other strategies. First, we obtain training data D in two ways. (i) We generate 1000 training samples, where each sample includes queries, partitioning strategies, and the performance of running the SQL queries on the partitioning strategies (Section 6.1). These samples are collected

Table 1. The Performance-Critical Hyper-Parameters.

Component	Parameter	Value
Partitioning Model	GNN Layers	2-3
	#-vertices	$1.12 \times \|max\ columns\|$
	node_dim	7
Evaluation Model	GNN Layers	1
	#-vertices	$\|sampled\ tuples\|$
	node_dim	3
Common Parameters	Sample Ratio	0.01%
	Dropout	0.5

from various settings with different queries, datasets, partitions, etc. It takes 2.3 hours to generate the samples. These samples are shared by different partitioning tasks and the generation time is not counted into the partitioning latency. (*ii*) We extract 800 workloads from the logs of 200 real clusters with 8000 nodes (the largest cluster has 512 nodes). Since the clusters rarely change the partitioning strategies, for each workload, we iteratively utilize the *Partitioning Model* and *Evaluation Model* (trained on the 1000 synthesized samples) to select new keys (partitioning strategies) and evaluate the performance scores. We select 500 high-quality partitioning strategies (with higher performance) for each dataset to generate training samples. Second, given a training sample $\langle Q^d, D^d, C^d, P^d \rangle$, where Q^d denotes queries, D^d denotes dataset, C^d is the selected columns, and P^d is the performance metrics like latency, we generate *k-node sample graph* based on $\langle Q^d, D^d, C^d \rangle$, utilize the *Evaluation Model* to estimate the performance, and update network weights with the loss of estimated performance and actual performance. The model converges when the loss values change little.

Loss Function. As there may be many noises in the performance metrics, which are brought by non-partitioning factors like different join orders and exclusive locks, we utilize a robust loss function Least Mean Log Squares (LMLS) [26] to reduce the side effects of these noises in real performance y_i by gradually decreasing the impact of large errors and computing an average of all training samples, i.e., $Loss_{LMLS} = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \frac{1}{2}|y_i - f(x_i)|)$, where n is the size of the training set and $f(x_i)$ is the evaluated performance.

Online Training. Next we fine-tune the *Evaluation Model* for evaluating different partitioning strategies. For a new partitioning request, the *Evaluation Model* inputs the keys selected by the *Partitioning Model* and re-computes a new *k-node sample graph* (counted in latency) . Then *Evaluation Model* estimates the latency L_t and throughput T_t based on the k-node sample graphs. If the computed feedback (Section 4.4) is high enough, we partition tables with the selected keys, execute queries, and obtain the real performance on these nodes to optimize the network weights in *Evaluation Model*.

6 EXPERIMENTS

Our database partitioning system `Grep` has been deployed into a commercial distributed database GaussDB [23]. We evaluate `Grep` from three aspects. (*i*) We compare the performance of the *Partitioning Model* with an enumeration method (EM) that enumerates candidate partitioning keys, and two state-of-the-art methods [22, 59]. (*ii*) We evaluate the accuracy and efficiency of *Evaluation Model*. (*iii*) We evaluate the adaptability of `Grep` on different workloads.

6.1 Experiment Setting

We implement `Grep` using Pytorch and utilize libraries such as psycopg2, scikit-learn, and numpy to interact with databases and pre-process data. We train the neural networks (GNNs) on a Titan RTX 2080Ti GPU with 11GB frame buffer. We conduct our experiments on: (1) An open-sourced

distributed database Postgres-XL with three compute nodes on two servers (16GB RAM, 256GB disk, 4Ghz CPU); (2) A commercial distributed database with six compute nodes on three servers (256GB RAM, 2TB disk, 3Ghz CPU).

Datasets. We use three datasets. (1) TPC-H is an OLAP benchmark with 50G data. It contains 61 columns and 3,224 synthetic queries with 1-5 joins; (2) JOB is an OLAP benchmark [30] with 13.1G data. It uses a real-world dataset IMDB, which contains 134 columns and 20,187 synthesis queries with 1-8 joins; (3) XuetangX is a real-world OLTP workload [2] for online education with 132.5G data. We take 14 tables with 204 columns and 22,000 real queries with 7-45 columns; (4) ICBC is a real banking database with 23 tables and 30K queries. The ICBC tables contain 29-2.5M rows and 5-66 columns. To train the networks, we split the generated training data into training/validation/test sets by 8:1:1 (10-fold cross-validation).

Hyper-Parameter Tuning. The hyper-parameters in `Grep` and recommended values are summarized in Table 1. First, we utilize the tuning tools like AutoGL [3] to search for the suitable GNN hyper-parameters. Second, we empirically tune other hyper-parameters (e.g., sample ratios in the evaluation model) to report the best performance. For `DRL`, we also tune the hyper-parameters (e.g., $5 \cdot 10^{-4}$ for the learning rate) as discussed in [22].

Training Data Generation. Besides the samples collected from real-world clusters (Section 5.1), we synthesize more training data based on the data and join features to pre-train the *Evaluation Model*. Taking TPC-H as an example, each synthetic query is generated by (*i*) picking 1-8 joined tables from the 50 join predicates; (*ii*) picking 1-61 columns from the 74 selection predicates and supplementing tables in the picked columns; (*iii*) concatenating all selected predicates with "AND"/"OR" and generating a query statement. To train the *Evaluation Model*, we run queries under typical partitioning strategies (e.g., PK-FK) to generate the training samples (Section 5.2). It takes 2.3 hours to generate 1000 training samples, where the features are queries and partitions, and the label is performance score. A well-trained *Evaluation Model* together with historical samples can be reused for *Partitioning Model*, which estimates based on the pooling of all the node vectors and can support training samples with various numbers of graph nodes.

Partitioning Metrics. (1) *Query Performance* includes the total latency and average throughput. In some cases, throughput is very important, e.g., OLTP workloads; in some other cases, query latency is more important, e.g., OLAP workloads. We use query performance as a general optimization objective; (2) *Partitioning Latency*: we use partitioning latency to estimate the overhead caused by database partitioning; (3) *Training Time*: training time includes data preparation (e.g., predicates parsing), and model training. *Partitioning Model* finishes training if the performance estimated by *Evaluation Model* on the test set meets the requirements (e.g., 20% latency reduction) or arriving the maximum iteration number.

Evaluation Metrics. We evaluate the *Evaluation Model* using evaluation latency, error rate and training time. Since the actual performance may be affected by factors that are not considered in database partitioning (e.g., exclusive clocks, network fluctuations), we use the mean error rate (MER) to estimate the error rate, formalized as $\frac{1}{N}\sum_{i=1}^{N}\frac{|Y_i - f(X_i)|}{Y_i}$, where N denotes the number of samples in the test set, Y_i and $f(X_i)$ are the actual/evaluated performance values of sample X_i respectively. Note that we use *LMLS* to train *Evaluation Model* (Section 5.2) and use *MER* to validate its accuracy.

Baseline Methods. We implement state-of-the-art studies to compare the performance of *Partitioning Model* and *Evaluation Model*. For *Partitioning Model*, we compare with three baselines:

[2]https://www.xuetangx.com/global

[3]https://github.com/THUMNLab/AutoGL

1. Enumeration-based method (Brute-force): We implement a basic enumeration-based method by enumerating every possible column combination as partitioning keys and utilizing the *Evaluation Model* to estimate the performance of these keys (on the sampled tuples) and pick the optimal solution. Brute-force can find the high-quality partitioning strategy but is time-consuming. For example, even if *Evaluation Model* takes less than 0.5s for a partitioning strategy, Brute-force requires hours to days to partition a dataset with 100+ columns.

2. Heuristic Search method (Heuristic): Heuristic search method [59] builds a table-level graph where vertices are tables and edges are foreign key constraints between tables. To optimize queries that can be executed locally per node, it searches for maximum spanning tree on the table-level graph, and partitions tables based on the foreign keys in the tree. However, it cannot consider column features and take the edge weight as the size of smaller table. It allows one column as a partitioning key for each table, but cannot select multiple columns as the partitioning key.

3. Deep-reinforcement-learning-based method (DRL): The DRL-based method [22] encodes the features of tables, samples queries into a binary vector s_t and iteratively selects an action to replicate/partition a table with the maximum estimated reward to s_t. It adopts a greedy strategy, which searches for a table chain with high reward. Besides, it also requires single-column partitioning. Note in the Postgres-XL cluster we train a RL agent for each benchmark, because DRL requires a cluster of agents to adapt to different workloads and tables.

For partitioning performance evaluation, we compare the performance (e.g, evaluation latency, error rate, training time) of *Evaluation Model* with two baselines:

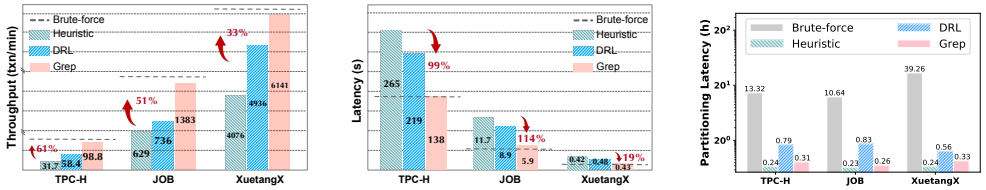
1. DL-based cost model (*NNCost*): We implement a neural-network (NN) based cost model [22]. *NNCost* is a three-layer dense network, which inputs a query and the selected columns and outputs the estimated latency. It works for single queries and cannot capture query/data distributions.

2. TLSTM-based cost model (*TLSTMCost*): We implement a TLSTM-based cost model [48], which estimates the query cost on query plans. For each query operator, it uses a Long Short-term Memory (LSTM) unit to estimate the execution cost, where the inputs are the operator features (e.g., predicates, tables) and the intermediate results of its child operators, and the output is the predicted cost. These units are organized in the tree structure to get the total query cost. However, *TLSTMCost* requires to partition the database, which is time-consuming; and it takes long time to execute the TLSTM model.

6.2 Performance Comparison

First we compare our *Partitioning Model* with (i) three state-of-the-art methods, i.e., enumeration-based method (Brute-force), heuristic method (Heuristic) [59], DRL-based method (DRL) [22]; (ii) Grep without attention (Grep(-A)) and Grep without Taylor Decomposition (Grep(-T)). On Postgres-XL, we compare the query performance, partitioning latency, and training time with these methods on the test workloads of TPC-H (Table 2), JOB (Table 3), and XuetangX (Table 4) respectively. On the commercial database, we showcase the superior latency reduction of the origin 22 TPC-H queries and queries in two real banking scenarios.

Query Performance on Postgres-XL. Grep outperforms Heuristic and DRL in all the cases and achieves similar performance as Brute-force. For example, Grep gains over 68.23% throughput improvement and 36.81% total latency reduction than DRL. The reasons are three-fold. (i) Grep encodes column features and query relations (e.g., read frequencies, multi-join costs) between columns as input. These features are important to partitioning-key selection but Heuristic only takes the table-level graph as input and DRL only encodes the query template frequencies as query features. For example, p_branch is frequently accessed by the TPC-H queries but has low distinct value ratio (around 0.001). So Grep considers the two factors and picks p_partkey and p_branch as the partitioning key, while others cannot. (ii) Grep uses the graph neural network to learn the



(a) Average Throughput (b) Total Latency (c) Partitioning Latency

Fig. 5. Performance Comparison with Baselines.

Table 2. Latency Reduction (%) on the Test Set of TPC-H.

Methods	90th	95th	99th	min	mean
Brute-force	58.23	54.31	50.02	42.13	67.11
Heuristic	27.79	21.46	13.27	6.38	32.82
DRL	35.12	33.17	28.63	23.41	44.61
Grep	**57.12**	**54.01**	**48.13**	**41.79**	**65.12**
Grep(-A)	49.39	48.59	47.02	37.23	54.27
Grep(-T)	51.24	49.38	47.31	38.19	56.34

Table 3. Latency Reduction (%) on the Test Set of JOB.

Methods	90th	95th	99th	min	mean
Brute-force	60.31	58.04	55.10	53.32	71.32
Heuristic	23.32	18.72	18.03	15.31	34.10
DRL	37.71	33.42	30.12	28.13	50.06
Grep	**59.93**	**56.73**	**54.23**	**52.17**	**66.91**
Grep(-A)	53.23	49.12	41.17	38.26	57.31
Grep(-T)	54.32	49.82	42.28	40.24	61.93

Table 4. Throughput Increase (%) on the Test Set of XuetangX.

Methods	90th	95th	99th	min	mean
Brute-force	473.3	466.8	441.6	425.2	527.3
Heuristic	288.6	246.6	159.2	123.2	391.5
DRL	316.5	250.1	216.4	198.0	437.3
Grep	**460.5**	**442.5**	**416.5**	**403.6**	**511.3**
Grep(-A)	391.2	377.1	352.2	324.4	462.2
Grep(-T)	423.2	402.8	391.2	355.8	485.8

maximal partitioning benefits of columns in the *column graph*, which involve factors like complex join patterns, data and query features. GNN can embed these three features together with graph convolutional operations and learned neural weights, while `Heuristic` selects FK-PK constraints as partitioning keys based on the table sizes and `DRL` denotes column features as 1(selected)/0(not selected), which cause sub-optimal partitioning strategies in complex cases. (*iii*) `Brute-force` enumerates all the partitioning strategies and achieves the best performance, but it is resource-consuming (e.g., taking up machines for tens of hours) and unaffordable in practice. `Grep` achieves similar performance as `Brute-force`, because with n columns, `Grep` generates n embedding vectors to approximate the origin complex column graph. Besides, `Grep` also outperforms `Grep(-A)` and `Grep(-T)`. First, from the experimental results, we find `Grep` selects fewer columns on tables like *lineitem*, *partsupp* (1.9 columns for each table on average) and gains much better performance. For example, for the *lineitem* table, `Grep` selects 2 columns as the key, while `Grep(-A)` and `Grep(-T)` both select 4 columns but cause negative effects, because `Grep(-A)` assigns the same initial weights for the input columns and `Grep(-T)` takes much longer time to learn the relations between the performance and complex embedding features (e.g., 512 neurons for the hidden layer), which both fail to remove useless columns from the final results. Thus, it is vital to prioritize important columns with attention and filter embedding features (into 1-d relevance) with Taylor Decomposition, especially for new columns without prior knowledge. Second, `Grep(-A)` achieves a bit worse performance than `Grep(-T)`, because *identifying important columns could be more beneficial than using complex algorithms.*

Query Performance on A Commercial Database. We separately test on standard (TPC-H) and real workload (ICBC). On TPC-H, as shown in Figure 6, `Grep` reduces the execution time of **17** queries than `DBA`, ranging from 3.7% (Q_7) to 73.5% (Q_{11}). Because `DBA` either uses primary keys or replicates the tables, while `Grep` selects more costly join columns and can gain higher performance. For example, `Grep` selects s_nationkey for *supplier* that occurs in nested queries and is joined with more columns. Besides, `Grep` partitions *nation* rather than replicating, because there are many joins on r_regionkey and it is more efficient to partition *region* into smaller parts on the join column and join these parts with other tables on different nodes. Second, `Grep` obtains similar performance as `DBA` on five queries, because queries like Q_1 only access primary columns and both `DBA` and `Grep` can gain relatively good performance. On a real scenario ICBC, there are 30K queries and `Grep` achieves 68% higher throughput. Specifically, we show some representative queries in Table 5.

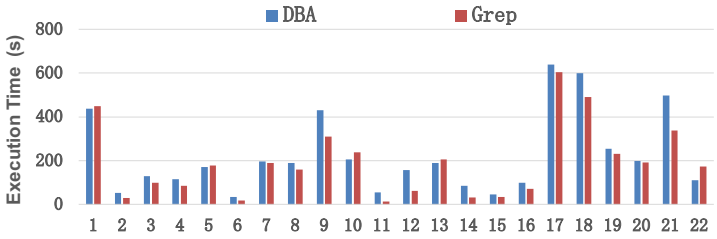


Fig. 6. Performance of TPC-H Queries (on Commercial Database). Grep gains 14.2% reduction in total.

Table 5. Latency Reduction of 5 Example ICBC Queries.

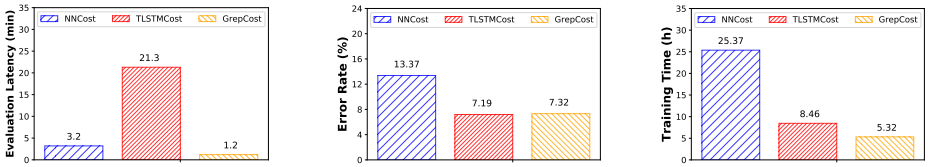
Query	Tokens	Execution Time (DBA)	Execution Time (Grep)
Q'_1	95	0.144s	0.130s (9.7% ↓)
Q'_2	508	0.178s	0.131s (26.4% ↓)
Q'_3	514	5.577s	0.642s (88.5% ↓)
Q'_4	433	0.155s	0.124s (20.0% ↓)
Q'_5	738	0.211s	0.131s (37.9% ↓)

These ICBC queries are very complex, which involve at least 6 tables, 25 columns, 5 joins, and 4 subqueries, and they cannot be well optimized by the DBA. Instead, Grep can judiciously select partitioning keys and achieve 9.7%-88.5% latency reduction for these queries (Table 5). The reasons are two-fold. First, the complex queries involve multiple joins on relatively small tables. Similar to TPC-H, DBA replicates these tables, while Grep partitions on join columns, and concurrently joins the smaller parts of these tables on different nodes so as to enhance query execution. Second, rather than greedily selecting columns with high join costs, Grep selects column combinations with high global benefits, which are computed based on the overall column graph structures (e.g., single joins in the edges and point queries in the vertices).

Partitioning Overhead. Grep outperforms Brute-force and DRL in all cases and achieves similar partitioning latency as Heuristic. For Heuristic, it has the least partitioning latency for two reasons. (*i*) It searches on the table-level graph, whose vertices (tables) are fewer than *column graph*. (*ii*) It directly outputs the results and does not optimize the solution based on the performance feedback. On the contrary, Grep improves partitioning efficiency from three aspects: (*i*) Grep utilizes an attention mechanism to reduce the number of iterations in order to achieve fast convergence; (*ii*) Grep concurrently embeds all the columns using the graph neural networks rather than one vertex at a time, where all the vertices share the same neural weights and can avoid local optimum; (*iii*) Grep selects partitioning keys based on the relevance of columns to the n embedded vertex vectors, which is further simplified and mainly reserves partition-related features. DRL takes relatively long time to generate the m-column sequence, where m is the number of tables. Brute-force takes the longest time by enumerating all partitioning strategies even if utilizing the *Evaluation Model*.

Training Time. We train Grep and DRL on the training set of TPC-H from scratch, and Grep takes much less training time than DRL (0.64 hours for Grep and 13.2 hours for DRL). The reasons are two-fold. First, the training of Grep is much more efficient. In each iteration, Grep partitions each table and gains the overall performance, while DRL only partitions for a table and cannot gain the reward until it finishes all the nodes. Second, as discussed in Section 6.3, *GrepCost* in Grep can estimate more accurate partitioning performance than *NNCost* in DRL, and thus Grep can optimize the GNN policy (neural weights) more efficiently and select a better partitioning strategy than DRL.

Summary. Grep outperforms state-of-the-arts methods in terms of both query performance and partitioning overhead. Brute-force takes longest partitioning latency (over 45.1x slower than Grep) and is not practical in real scenarios; and Heuristic gains the shortest latency but has the



(a) Evaluation Latency (b) Error Rate (c) Training Time

Fig. 7. Performance Evaluation of TPC-H workloads on Postgres-XL.

worst performance (over 2.1x worse in throughput and 48.7% worse in latency than `Grep`). `Grep` gains similar performance as `Brute-force` and similar partitioning latency as `Heuristic` and outperforms `DRL` in all the three aspects.

6.3 Evaluation on Performance Evaluation

Next we evaluate our *Evaluation Model*, which estimates performance for different partitioning strategies. We compare evaluation latency, accuracy and training time (*GrepCost*) with *NNCost* and *TLSTMCost*. We train each method with the training set of TPC-H on Postgres-XL and the validation results are shown in Figure 7.

Evaluation Latency. It is vital to quickly get relatively accurate partition-performance feedback, which is used to evaluate the partitioning strategies or train the model. As shown in Figure 7(a), *GrepCost* achieves the lowest evaluation latency, 62.5% lower than *NNCost* and 99.0% lower than *TLSTMCost*. The reasons are two-fold. First, *GrepCost* concurrently estimates performance within nodes and that across nodes in graph level, which takes seconds, while the other two methods work for single queries and take longer time to enumerate the costs of all the queries. Second, *GrepCost* utilizes sampled tuples to estimate performance and does not partition all the datasets, while *TLSTMCost* relies on query plans and needs to actually partition the database, which takes over 20 mins to load the 50GB TPC-H data. Besides, although *NNCost* achieves relatively low latency, its error rate is much worse than *GrepCost* and cannot provide reliable feedback for *Partitioning Model*.

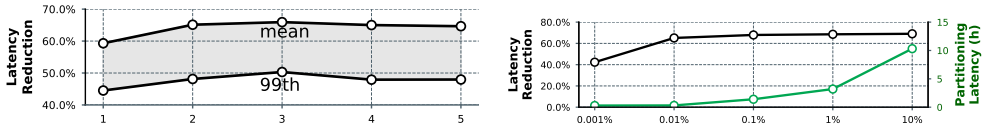
Evaluation Accuracy. First, *GrepCost* achieves much lower error rate than *NNCost*, because *NNCost* only inputs a single query and the selected columns, which cannot capture query and data allocation information and cause information loss. Instead, our model simulates nodes on sampled tuples with the *k-node sample graph* and learns the overall performance by embedding the whole graph. Second, the error rate of *GrepCost* is a bit higher than *TLSTMCost*, because *TLSTMCost* works on physical operators and captures detailed execution features. However, there are two problems in *TLSTMCost* for database partition. (*i*) *TLSTMCost* requires actual database partitioning to obtain query plans, which are costly and impractical; (*ii*) *TLSTMCost* estimates based on query features and cannot capture differences in data allocation caused by different partitioning strategies.

Training Time. *GrepCost* takes the least training time. The reasons are two-fold. First, `Grep` extracts key data features (e.g., tuples within different nodes) and query features (e.g., queries within each node and across nodes) as an input graph and can learn from graph changes under different partitioning strategies; while other methods are unaware of partition changes and take much more iterations to learn the hidden partitioning factors. Second, *GrepCost* computes loss function (LMLS) based on the overall performance of all queries, which helps reduce noises caused by single queries, and *TLSTMCost* takes longer time to converge for single queries.

Summary. *GrepCost* outperforms state-of-the-art performance evaluation methods in latency by 62.5%–99.0% and training time by 37.1%–79.0%, and achieves similar error rate as the first-partition-then-evaluation method *TLSTMCost*.

6.4 Evaluation on Hyper-Parameters

Next we evaluate the impact of two hyper-parameters to the performance of `Grep` on TPC-H, where *layer numbers* affect the feature embedding capability of the GNN model and *sample rates* affect



(a) Layer Number (b) Sample Rate

Fig. 8. The Effectiveness of Hyper-Parameters.

the accuracy of the *Evaluation Model*. We report the average latency reduction of our method on running TPC-H queries (Figure 8).

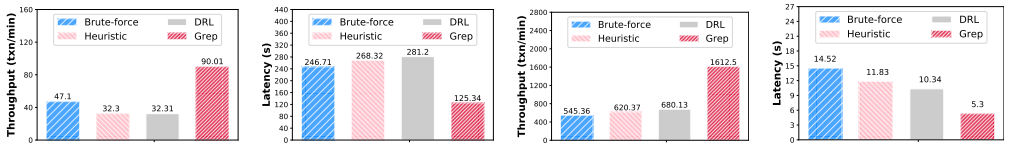
Layer Numbers. From Figure 8 (a), we have two observations. First, as adding more hidden layers when the layer number is smaller than 3, the performance increases, because the column features can be captured with one GNN layer but other important features like multi-joins need to be captured with 2-3 GNN layers. Second, when the layer number is larger than 3, the new embedding features make minor contributions but incur higher overhead. Thus, `Grep` adopts 2 layers to balance performance and training overhead.

Sample Rates. As shown in Figure 8 (b), `Grep` achieves better performance with larger sample rates. This is because we can sample many more tuples to improve the accuracy of *Evaluation Model*, which better approximate the real data distribution and could capture complicated partitioning problems (e.g., data skew cannot be identified with too few tuples). And *Partitioning Model* can efficiently select high-quality keys based on the accurate evaluations. However, even slightly increasing the sample ratio may incur millions of additional tuples, which slows down *Evaluation Model* and causes great partitioning overhead. Thus, generally we set the sample ratio as 0.01%, especially for large datasets (e.g., over 50GB).

6.5 Adaptability on Varying Datasets

In real-world scenarios, both queries and tables can change. Hence, we verify the performance when dataset changes. We conduct two experiments. (1) We use the *Partitioning Model* trained on XuetangX (RW) for evaluating TPC-H (RO). (2) We use the *Partitioning Model* trained on TPC-H (RO) for evaluating JOB (RO). We plot the estimated performance after 20 minutes and Figure 9 shows the results. We initialize the learned methods with all tested datasets (e.g., 150-d action vector for `DRL`, 150×150 edge matrix for `Grep`).

XuetangX to TPC-H. We have two observations. First, `Grep` outperforms the other three methods in both latency and throughput, because (*i*) `Grep` characterizes data changes into the features of columns (e.g., table size, tuple selectivity) and query changes into the correlations between columns, based on which `Grep` could adapt the learned knowledge to other datasets (e.g., unique columns involved in costly joins are more possible to select as the partitioning keys; multiple columns from the same table on the sample path with high cost are more possible to select as the composite keys). We can finetune the learned knowledge and use them to recommend new partitioning keys. While other methods only characterize simple column features (e.g., 0/1 in `DRL`) and fail to learn new performance corelation; (*ii*) Most of `Grep` features like tuple selectivity (Section 3.1) can be easily transferred because columns may have similar data distribution and the knowledge learned on existing columns could be generalized to similar columns. Note the *table id* is only used to distinguish tables, and the different values of table id do not need to be transferred; (*iii*) Since the embedding relations (e.g., join costs) may change on TPC-H, the evaluation model helps to finetune the network weights of the selection model by estimating the benefits of selected keys; (*iv*) The *Evaluation Model* adaptively gives accurate feedback based on the tuple/query changes in *k-node sample graph*. Instead, `DRL` assumes data will not change and cannot find good solution in the new action space within limited time; `Heuristic` can capture table changes with the table-level



(a) XuetangX to TPC-H (b) XuetangX to TPC-H (c) TPC-H to JOB (d) TPC-H to JOB

Fig. 9. Dataset Adaptability on Postgres-XL.

graph and find new PK-FK chain, but it neglects query changes and finds sub-optimal solution. Second, `Grep` achieves better query latency than that originally trained on TPC-H, because queries in XuetangX involve many columns and `Grep` recommends composite keys for XuetangX. When migrated to TPC-H, `Grep` still selects multiple columns and can reduce the total latency by accurately locating TPC-H queries to desired nodes.

TPC-H to JOB. We reuse the well-trained key selection model on TPC-H for JOB and get similar results. The main reason is that we pre-train the key selection models to determine beneficial columns from various graph structures (TPC-H workloads). The pre-trained models can potentially be useful for JOB workloads. Note that, for JOB, it mainly needs to learn the column features, which cannot be directly compared by values. Instead, other methods cannot utilize historical partitions and find sub-optimal solutions.

Summary. `Grep` can adapt to different datasets using the trained model on a dataset, because `Grep` only use the database basic information, e.g., join cost, but does not rely on the query information.

7 RELATED WORK

Database Partitioning. Most of existing methods horizontally partition data in table level, which can be divided into three categories: (1) Partitioning-function-based methods [7, 11, 35, 41, 50] clustered data tuples into blocks based on query predicates and searched for a suitable range partitioning function that minimized the costs of all the blocks. Besides, there were methods that aimed to dynamically repartition data based on the join frequencies [35] or transaction relationships [11]. However, they focused on Hadoop and Spark systems and we need a partitioning method applicable on any database systems. (2) Foreign-key-based heuristic methods [15, 42, 44, 45, 47, 59]. Eadon et al. [15] proposed to select reference relations in table-level graph and improve data locality by copying the referenced tuples to the compute nodes of the referencing tuples. However, this method brought in data redundancy. So Zamanian et al. [59] extended this method to minimize data replicates by enumerating the costs of all partitioning strategies with maximal data-locality. Besides, Panos et al. [42] proposed to apply either heuristic or exact algorithms by time limit, but also selected partitioning keys in table level and can be further optimized. (3) Reinforcement-learning-based methods [21, 22]. Recently various learning-based methods have been proposed to optimize database components [9, 12, 24, 28, 31, 31–33, 43, 49, 53–55, 57, 58, 60, 62–65]. For database partitioning, Hilprecht et al. [22] encoded the tables, query frequencies, foreign keys into a state vector and used the DRL model to select a key or a replication table at each iteration. However, it could not adapt to new workloads and the cost model works for single queries and cannot estimate the partitioning quality. Experimental results showed our method outperforms them in terms of efficiency, quality and adaptivity (Section 6). RL-based method for vertical partitioning [13] explores how to split table columns into different partitions so as to optimize the performance on specific workload. It meets similar problems as that for horizontal partitioning.

Attention on Graph Model. Attention mechanism identifies task-related inputs and can enhance downstream processing tasks, especially when the graph is large. There are some graph-based attention methods [29, 37] for tasks like graph classification [14] and visual dialog [46]. To our best knowledge, this is the first work that uses an attention model to enhance database partitioning.

Graph Embedding. Graph embedding provides a natural way to learn the representation of a graph model, which can be largely divided into spectral-based approaches [25, 36, 65] and spatial-based approaches [39, 40]. In database partitioning, we combine the advantages of spectral and spatial graph embedding – we learn a partitioning-related representation with the local subgraph, edge matrix and trainable weights, which efficiently embeds local and global graph features. To our best knowledge, this is the first work that uses graph embedding for database partitioning.

8 CONCLUSION AND FUTURE WORK

In this paper we proposed a graph learning based database partitioning system. We proposed a graph model to represent data and queries, where vertices are columns and edges are column correlations. We computed the embeddings of the graph and used the embeddings to select partitioning keys. To improve the training efficiency, we proposed a learned evaluation model that estimated the performance of a partitioning strategy without needing to actually partition the data. Experimental results showed that our method outperformed state-of-the-art studies.

There are some open problems. First, `Grep` focuses on partitioning a centralized database to a distributed database, which is conducted offline and not sensitive to partitioning overhead. For online database repartitioning, we should design an adaptive partitioning function (policy) that considers the re-partitioning cost. Second, we can consider more parititioning functions, e.g., range, and more partitioning methods, e.g., vertical partitioning.

ACKNOWLEDGEMENTS

This paper was supported by NSF of China (61925205, 62232009, 62102215), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] [n. d.]. aws.amazon.com/cn/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-distribution-styles-and-distribution-keys.
[2] [n. d.]. docs.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-tables-distribute.
[3] [n. d.]. https://www.ibm.com/docs/en/db2-warehouse?topic=database-choosing-hash-distribution-key.
[4] [n. d.]. https://www.snowflake.com/wp-content/uploads/2014/10/A-Detailed-View-Inside-Snowflake.pdf.
[5] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. 1110–1121. https://doi.org/10.1016/B978-012088469-8.50097-8
[6] Stephen Bazen and Xavier Joutard. 2013. The Taylor decomposition: A unified generalization of the Oaxaca method to nonlinear models. *tech. rep.* (2013).
[7] Martin Boissier and Kurzynski Daniel. 2018. Workload-Driven Horizontal Partitioning and Pruning for Large HTAP Systems. In *ICDE*. https://doi.org/10.1109/ICDEW.2018.00026
[8] Immanuel M Bomze, Marco Budinich, Panos M Pardalos, and Marcello Pelillo. 1999. The maximum clique problem. In *Handbook of combinatorial optimization*. Springer, 1–74.
[9] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *SIGMOD*. ACM, 646–659. https://doi.org/10.1145/3514221.3517882
[10] Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*. ACM, 3–14. http://www.vldb.org/conf/2007/papers/special/p3-chaudhuri.pdf
[11] Carlo Curino, Yang Zhang, Evan P. C. Jones, and et al. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *VLDB* (2010). https://doi.org/10.14778/1920841.1920853
[12] Jialin Ding, Ryan C. Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. SageDB: An Instance-Optimized Data Analytics System. *Proc. VLDB Endow.* 15, 13 (2022), 4062–4078. https://www.vldb.org/pvldb/vol15/p4062-ding.pdf
[13] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. 2019. Automated Vertical Partitioning with Deep Reinforcement Learning. In *ADBIS (Communications in Computer and Information Science, Vol. 1064)*. Springer, 126–134. https://doi.org/10.1007/978-3-030-30278-8_16

[14] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, and et al. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *NIPS*. http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fingerprints

[15] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, and et al. 2008. Supporting table partitioning by reference in oracle. In *SIGMOD*. https://doi.org/10.1145/1376616.1376727

[16] Massimo Franceschet. 2011. PageRank: standing on the shoulders of giants. *Commun. ACM* (2011). https://doi.org/10.1145/1953122.1953146

[17] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan O. Pedersen. 2004. Combating Web Spam with TrustRank. In *VLDB*. https://doi.org/10.1016/B978-012088469-8.50052-8

[18] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *PVLDB* 11, 4 (2017), 499–512. https://doi.org/10.1145/3186728.3164145

[19] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1477–1492. https://doi.org/10.1145/2723372.2749438

[20] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. 2011. Query optimization techniques for partitioned tables. In *SIGMOD*. https://doi.org/10.1145/1989323.1989330

[21] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards learning a partitioning advisor with deep reinforcement learning. In *aiDM@SIGMOD*. https://doi.org/10.1145/3329859.3329876

[22] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *SIGMOD*. https://doi.org/10.1145/3318464.3389704

[23] Ltd. Huawei Technologies Co. 2022. Basic Knowledge of Database. In *Database Principles and Technologies–Based on Huawei GaussDB*. Springer, 41–86.

[24] Stratos Idreos and Tim Kraska. 2019. From Auto-tuning One Size Fits All to Self-designed and Learned Data-intensive Systems. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 2054–2059. https://doi.org/10.1145/3299869.3314034

[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*. https://openreview.net/forum?id=SJU4ayYgl

[26] Miroslaw Kordos and Andrzej Rusiecki. 2016. Reducing noise impact on MLP training - Techniques and algorithms to provide noise-robustness in MLP network training. *Soft Comput.* (2016). https://doi.org/10.1007/s00500-015-1690-9

[27] Mayuresh Kunjir and Shivnath Babu. 2020. Black or White? How to Develop an AutoTuner for Memory-based Analytics. In *SIGMOD*. ACM, 1667–1683. https://doi.org/10.1145/3318464.3380591

[28] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* 6, 1 (2021), 86–101. https://doi.org/10.1007/s41019-020-00149-7

[29] John Boaz Lee, Ryan A. Rossi, and Sungchul Kim et al. 2019. Attention Models in Graphs: A Survey. *TKDD* (2019). https://doi.org/10.1145/3363574

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, and Peter A. Boncz et al. 2015. How Good Are Query Optimizers, Really? *VLDB* (2015). https://doi.org/10.14778/2850583.2850594

[31] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*. ACM, 2859–2866. https://doi.org/10.1145/3448016.3457542

[32] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *PVLDB* (2019). https://doi.org/10.14778/3352063.3352129

[33] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041.

[34] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Found. Trends Inf. Retr.* (2009).

[35] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *VLDB* (2017). https://doi.org/10.14778/3055540.3055551

[36] Yi Ma, Jianye Hao, Yaodong Yang, Han Li, Junqi Jin, and Guangyong Chen. 2019. Spectral-based Graph Convolutional Network for Directed Graphs. *CoRR* abs/1907.08990 (2019). arXiv:1907.08990 http://arxiv.org/abs/1907.08990

[37] Volodymyr Mnih, Nicolas Heess, Alex Graves, and et al. 2014. Recurrent Models of Visual Attention. In *NIPS*. http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention

[38] Grégoire Montavon, Sebastian Lapuschkin, and Alexander Binder et al. 2017. Explaining nonlinear classification decisions with deep Taylor decomposition. *Pattern Recognit.* (2017). https://doi.org/10.1016/j.patcog.2016.11.008

[39] Federico Monti, Davide Boscaini, Jonathan Masci, and et al. 2017. Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs. In *CVPR*. https://doi.org/10.1109/CVPR.2017.576

[40] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. ICML *(JMLR Workshop and Conference Proceedings)*. JMLR.org. http://proceedings.mlr.press/v48/niepert16.html

[41] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *VLDB J.* 29, 1 (2020), 569–591. https://doi.org/10.1007/s00778-019-00580-x

[42] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. 2020. Fast and Effective Distribution-Key Recommendation for Amazon Redshift. *VLDB* (2020), 2411–2423. http://www.vldb.org/pvldb/vol13/p2411-parchas.pdf

[43] Andy Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (2021), 3211–3221. https://doi.org/10.14778/3476311.3476411

[44] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 315–330. https://doi.org/10.1145/3035918.3064052

[45] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. 2002. Automating physical database design in a parallel database. In *SIGMOD*. ACM, 558–569. https://doi.org/10.1145/564691.564757

[46] Idan Schwartz, Seunghak Yu, and Tamir Hazan et al. 2019. Factor Graph Attention. In *CVPR*. https://doi.org/10.1109/CVPR.2019.00214

[47] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (2016), 445–456.

[48] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *VLDB* (2019). https://doi.org/10.14778/3368289.3368296

[49] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97. https://doi.org/10.14778/3485450.3485459

[50] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and et al. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*. https://doi.org/10.1145/2588555.2610515

[51] Immanuel Trummer. 2019. Exact Cardinality Query Optimization with Bounded Execution Cost. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, and et al (Eds.). https://doi.org/10.1145/3299869.3300087

[52] Petar Velickovic, Guillem Cucurull, and Arantxa Casanova et al. 2018. Graph Attention Networks. In *ICLR*. https://openreview.net/forum?id=rJXMpikCZ

[53] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84. https://doi.org/10.14778/3485450.3485458

[54] Junxiong Wang, Immanuel Trummer, and et al. 2021. Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *SIGMOD*. 2794–2797.

[55] Sai Wu, Ying Li, Haoqi Zhu, Junbo Zhao, and Gang Chen. 2022. Dynamic Index Construction with Deep Reinforcement Learning. *Data Sci. Eng.* 7, 2 (2022), 87–101. https://doi.org/10.1007/s41019-022-00186-4

[56] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[57] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936. https://www.vldb.org/pvldb/vol15/p3924-li.pdf

[58] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[59] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware Partitioning in Parallel Database Systems. In *SIGMOD*. https://doi.org/10.1145/2723372.2723718

[60] Lixi Zhang, Chengliang Chai, Xuanhe Zhou, and Guoliang Li. 2022. LearnedSQLGen: Constraint-aware SQL Generation using Reinforcement Learning. In *SIGMOD*. ACM, 945–958. https://doi.org/10.1145/3514221.3526155

[61] Muhan Zhang, Zhicheng Cui, and Marion Neumann et al. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *AAAI*. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17146

[62] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2022. Database Meets Artificial Intelligence: A Survey. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1096–1116. https://doi.org/10.1109/TKDE.2020.2994641

[63] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58. https://doi.org/10.14778/3485450.3485456

[64] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. AutoIndex: An Incremental Index Management System for Dynamic Workloads. In *ICDE*. IEEE, 2196–2208.

[65] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *VLDB* (2020). http://www.vldb.org/pvldb/vol13/p1416-zhou.pdf

Received July 2022; revised October 2022; accepted November 2022