Query Performance Prediction for Concurrent Queries using Graph Embedding

Xuanhe Zhou, Ji Sun, Guoliang Li, Jianhua Feng
Department of Computer Science,Tsinghua University, Beijing, China
{zhouxuan19,sun-j16}@mails.tsinghua.edu.cn; {liguoliang,fengjh}@tsinghua.edu.cn

ABSTRACT

Query performance prediction is vital to many database tasks (e.g., database monitoring and query scheduling). Existing methods focus on predicting the performance for a single query but cannot effectively predict the performance for concurrent queries, because it is rather hard to capture the correlations between different queries, e.g., lock conflict and buffer sharing. To address this problem, we propose a performance prediction system for concurrent queries using a graph embedding based model. To the best of our knowledge, this is the first graph-embedding-based performance prediction model for concurrent queries. We first propose a graph model to encode query features, where each vertex is a node in the query plan of a query and each edge between two vertices denotes the correlations between them, e.g., sharing the same table/index or competing resources. We then propose a prediction model, in which we use a graph embedding network to encode the graph features and adopt a prediction network to predict query performance using deep learning. Since workloads may dynamically change, we propose a graph update and compaction algorithm to adapt to workload changes. We have conducted extensive experiments on real-world datasets, and experimental results showed that our method outperformed the state-of-the-art approaches.

PVLDB Reference Format:
Xuanhe Zhou, Ji Sun, Guoliang Li, Jianhua Feng. Query Performance Prediction for Concurrent Queries using Graph Embedding. *PVLDB*, 13(9): 1416-1428, 2020.
DOI: https://doi.org/10.14778/3397230.3397238

1. INTRODUCTION

Query performance prediction is a vital task in database systems to meet the service level agreements (SLAs), which can benefit many database applications, such as transaction scheduling [11], parameter tuning [16], and progress monitoring [21]. Traditional prediction methods are designed for single queries [29, 31] and they cannot effectively predict the performance for concurrent queries, because they cannot capture the correlations between concurrent queries, e.g., global buffer sharing and lock conflict, which can significantly affect the query performance of concurrent queries.

There are some studies on performance prediction for concurrent queries. BAL [8] captures logical I/O metrics (e.g., page access time) and neglects many resource-related features (e.g., buffer size, computation costs), and cannot effectively predict the performance. Besides, BAL [8] uses linear regression and cannot get accurate prediction results for complex features. DL [20] uses neural units to learn from query plans and does not consider many potential relations between concurrent queries (e.g., data/resource conflicts). Moreover, these methods cannot effectively predict the performance when the queries dynamically change.

To address these problems, we propose a performance prediction system `GPredictor` using graph embedding, which provides real-time query performance prediction for concurrent and dynamic workloads. `GPredictor` first uses a graph model to represent the workload characteristics, in which the vertices represent operator features extracted from query plans and edges between two operators denote the query correlations and resource competitions between them. Then `GPredictor` feeds the workload graph into the prediction model, and we propose a graph embedding algorithm to embed graph features and design a deep learning model to predict query performance. Moreover, if a graph is too large, it may affect the prediction efficiency. So we propose a graph update and compaction algorithm, which removes redundant vertices and combines similar vertices.

Contributions: We make the following contributions.
(1) We propose a graph-embedding model to predict the query performance of concurrent queries. *To the best of our knowledge, this is the first performance prediction system that uses graph embedding to predict query performance of concurrent and dynamic workloads* (see Section 2).
(2) We propose a graph-structured workload model to capture features of query operators and the correlations between different operators, e.g., data sharing/conflict, and resource competition (see Section 3).
(3) We propose a graph-based prediction model to predict query performance in a workload graph by (*i*) learning the performance-related features using a graph embedding method; (*ii*) learning high-dimensional mapping from embedded features to performance using deep learning (see Section 4).
(4) For dynamic workloads, we adaptively predict the performance on the workload graph. And we propose a graph compaction algorithm to reduce the graph size and cut down the prediction overhead (see Section 5).

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097..
DOI: https://doi.org/10.14778/3397230.3397238

(5) We have conducted extensive experiments on different workloads and running environments. Experimental results showed that GPredictor achieved high accuracy and outperformed the state-of-the-art methods (see Section 6).

2. SYSTEM OVERVIEW

Architecture. Given a query workload with multiple concurrent queries, GPredictor aims to predict the execution time of each query. Figure 1 shows the architecture of GPredictor, which mainly includes three modules. First, Workload2Graph extracts features from the query workload that may affect the query performance of concurrent queries. For example, it extracts physical operators from the query plans, and obtains the correlations between different operators, e.g., data sharing between operators and lock conflicts between operators. It also collects statistics (e.g., database configuration values, active tasks) from database system views. Then Workload2Graph uses these features to characterize the behaviors of concurrent queries in the form of a graph, where the vertices are operators in the queries and the edges are the correlations between two operators (see Section 3). Then PerformancePredictor adopts a graph-based learning model to embed the graph, and predicts the query performance using a deep learning model (see Section 4). Note that PerformancePredictor first predicts the latency of each vertex in the graph, with which we derive the latency of each concurrent query. Next GraphOptimizer optimizes the graph model from two aspects. Graph Updater updates the graph for dynamic workloads by inserting new queries and removing finished queries. Graph Compactor compacts the graph by merging the vertices that may be executed concurrently in order to reduce the graph size and prediction overhead (see Section 5).

Workflow. When a new query comes, Workload2Graph extracts its operator features from the query plan, inserts these operators into the workload graph as new vertices, and updates the edges of the graph model. Then GraphOptimizer compacts the graph by aggregating some vertices. Next PerformancePredictor embeds the graph into vectors, updates the embedded vectors whose local graph features have changed, and re-predicts the performance of all queries under execution. When some operators are complete, Graph Updater updates the graph by removing the corresponding vertices. Then PerformancePredictor re-predicts the query performance of all queries under execution.

Figure 2 shows a running example of building workload model for four concurrent queries. We first extract the query plan for each query, take nodes in the plans as vertices, add some edges between the query nodes (e.g., data-sharing/lock-conflict relationships between different nodes), and generate a graph. Then we embed the vertices and edges into a vertex matrix and an edge matrix. Next we predict the performance based on the two matrices.

Remark: We assume that (1) we get a query plan from the database before execution, and do not consider pipelined query execution; (2) the plan will not change during execution, and we do not consider adaptive query processing.

3. GRAPH MODEL FOR CONCURRENT WORKLOADS

In this section we study how to characterize concurrent queries. There are two main challenges in concurrent query modeling. First, concurrent queries in a workload may have different numbers of operators, and it is hard to model the

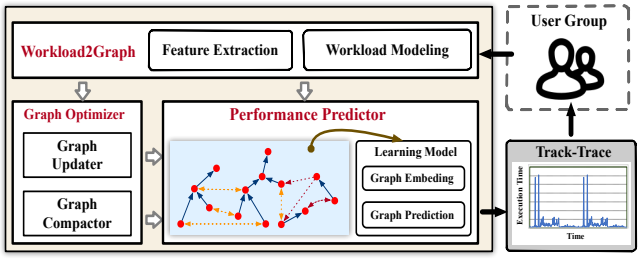


Figure 1: The GPredictor Architecture

queries using traditional fix-length encoding methods. Second, it requires us to capture the correlations between different operators. Note operators in a concurrent workload may have different relationships (e.g., buffer sharing, data conflict), which is hard to be represented using traditional modeling methods (e.g., linear regression, statistic model). To address this problem, we propose a graph-structured workload model, which is composed of two matrices, vertex matrix and edge matrix, and we use the two matrices to predict the performance of concurrent queries.

3.1 Graph Model

Given a set of concurrent queries in a workload, we extract the query plan tree for each query. Then we add edges between different operators in the plan trees (e.g., buffer sharing, data conflicts, resource competition), which forms a graph to represent the workload characteristics. For example, Figure 2 shows the graph model for four queries. We extract 14 operators and add 20 relationships between them.

DEFINITION 1 (GRAPH MODEL). *Given a set of queries, we model the queries into a graph. The vertices are physical operators in the query plans and each represents the features of corresponding operator. There is an edge between two operators if they satisfy one of the following relationships:*
(1) ***parent-child relationship*** *in the same query plan;*
(2) ***data-sharing:*** *they access the same data, i.e., visiting the same table or index;*
(3) ***data-conflict:*** *they have read-write/write-write access conflicts;*
(4) ***resource-competition:*** *they compete the resource at the same time, e.g., competing memory, CPU, I/O bandwidth;*

Next we introduce the details of constructing a graph.

3.1.1 Vertex Modeling

Vertices are physical operators in the query plan tree. Each vertex contains key features of the corresponding operator, including estimated execution cost, operator type, predicates, and sample bitmap.

Estimated Cost. The execution cost of an operator includes the size of the input table and the complexity of executing the operator (e.g., CPU time to process all the tuples). We use the cost values provided by the database optimizer to estimate the actual execution cost [29]. To avoid large errors in cost estimation [15], we tune the parameters in the cost model (e.g., seq_page_cost, cpu_tuple_cost) based on the hardware statistics before extracting cost values.

Operator Type. Each operator in the execution plan represents a type of data manipulation operation on database tuples, including aggregation, hash join, sequential scan, etc. These physical operators decide the resources required for processing these operators. We encode operation types into one-hot. As Figure 2 shows, for vertex v_3, it is a sequence scan operator and is encoded into "001000" based on the

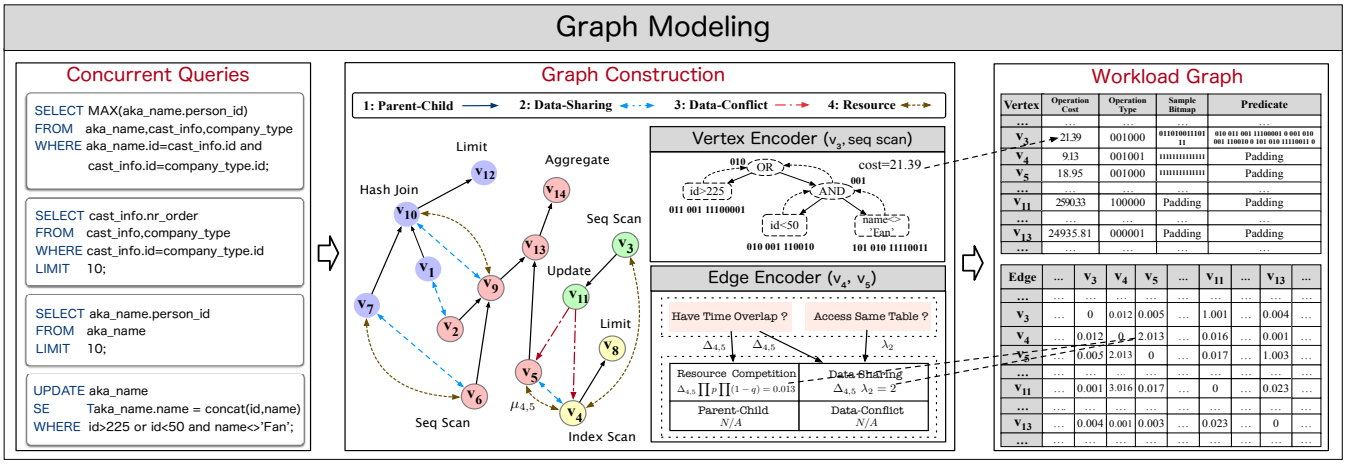


Figure 2: An Example Graph Modeling of Four Concurrent Queries (Δ: the weight of time overlap).

encoding table. As the number of database operators is limited, we can pre-define the size of operation types.

Predicates. Predicates are the filter/join conditions in an operator. On one hand, predicates affect the cost of operator execution, because different predicates may access different tuples. On the other hand, predicates can capture the similarity of data access, e.g., "`aka_name.id>1000`" and "`aka_name.id>990`". To encode a predicate (e.g., `aka_name.id>225` or `aka_name.id<50` or `aka_name.name='Julie'`), we adopt the method used in cardinality estimation [25], which encodes each condition in the predicate into a one-hot vector and then concatenates these vectors in a depth-first order.

Sample Bitmap. For each table, we sample some tuples. With sample tuples, we can execute the operator, get the query result on those tuples, and use it to provide an approximation of the query cost. Moreover, in case of skewed data, we use stratified sampling [13], which better covers distinct tuples with limited feature bits.

Vertex Formulation. For a vertex v_i, we denote features of `Cost`, `Operator Type`, `Predicates`, `Sample Bitmap` as C_i, O_i, Pr_i, S_i. We concatenate these features and formulate the vertex feature vector as: $V_i = [\hat{C}_i, \hat{O}_i, \hat{Pr}_i, \hat{S}_i]$, where $\hat{C}_i, \hat{O}_i, \hat{Pr}_i, \hat{S}_i$ are the normalized features of C_i, O_i, Pr_i, S_i respectively. We use estimated cost $\hat{C}_j$ and operator type $\hat{O}_i$ to reflect query features, and use predicate $\hat{Pr}_i$ and sample bitmap $\hat{S}_i$ to reflect data features (e.g., the accessed data ranges). For example, in Figure 2, the feature vector of v_4 is (18.88, 001000, 11...111, Padding), where the predicate feature is padded with 0, because v_4 has no predicate.

3.1.2 Edge Modeling

The execution of concurrent queries has the following features: (1) There are data dependencies between operators in the same query plan; (2) Data in the shared buffer is available for multiple queries, bringing in data sharing and conflicts; (3) Resources allocation is affected by database configurations (e.g., Work_Mem, Max_Connections). We use edges to capture *potential* correlations between vertices. We consider four types of correlations to add edges, and other types of correlations can be easily integrated into our method.

(1) Parent-Child Relationship. In the query plan tree of a query, parent-child relationships capture the data flow between different vertices. First, the start time of a parent vertex depends on the finish time of its children. Second, the resource usage of a parent vertex depends on the result size passed from its children. Thus the parent-child relationship should be used to predict the performance.

(2) Data Sharing. Different operators may share the same data. If one operator loads the data into memory, other operators sharing the data can directly utilize the cached data and thus can reduce the computation cost. There are three categories of data sharing,

1) *Index Sharing.* Different operators may utilize the same index. If one query operator already uses the indexes and caches some information, the cached information can benefit other queries. For example, on the same table `aka_name`, both `Hash Scan` (`id > 90`) and `Hash` (`id=100`) can use the index `index_id` of `aka_name`.

2) *Table Sharing.* Different operators may scan the same table. If one query operator loads the table (or a part) into memory, other query operators can utilize the cached data. For example, in Figure 2, the operators of vertex v_5 and vertex v_4 use the same table `aka_name`.

3) *Intermediate Results Sharing.* Database may cache frequently accessed intermediate results for data sharing between operators. The vertex with the same sub-tree in the query plan will share the same intermediate results. We add an edge between two vertices if they have the same sub-tree. In Figure 2, vertex v_{10} and vertex v_9 may share the same intermediate results, because they have the same sub-query.

(3) Data Conflict. Different operators may have data conflict, e.g., read-write, write-write conflicts on tables and indexes. The conflict data may be locked and this will affect the concurrency. Formally, if two operators manipulate the same data and at least one of them is a write operator, there could be data-conflict relationship between them. For example, in Figure 2, there is data-conflict relationship from vertex v_{11} to vertex v_5, because vertex v_{11} updates table `aka_name` and comes earlier than vertex v_5. So vertex v_5 may wait for the lock until vertex v_{11} finishes. We evaluate whether two operators have conflict based on their starting time. For any operator o of query q, we denote its starting time as t_q+c_o, where t_q is the commit time of query q, and c_o is the startup time of operator o related to the commit time of query q. c_o is estimated by the database optimizer.

(4) Resource Competition. There may be resource competition between two concurrent operators. First, for analytical queries, memory size is usually the bottleneck. When conducting too many large-scale join operators, the data may move to disk and affect the efficiency. Second, for transactional tasks, we have two observations. (1) The num-

ber of concurrent queries may be larger and the concurrency level is limited by the maximal connection threshold. (2) There are frequent I/O operations and so the I/O bandwidth can be the bottleneck. So here we mainly consider memory, CPU, and I/O bandwidth competitions, which are controlled by database parameters. For example, in PostgreSQL, `work_mem` controls the size of memory space; and `max_connections` controls the maximum connection numbers; `effective_io_concurrency` sets the number of concurrent disk I/O operations. Note the parameters of memory and I/O are usually positively related to query performance; while the parameters of concurrency control may take negative effects, as they may bring more operator competitions.

Execution Time Overlap. For vertices v_i, v_j, if they have no execution time overlap, we can assume they have no resource/lock conflicts and less possibility of data sharing; otherwise, the larger the overlap is, the higher possibility of query conflict is. Thus we need to consider the execution overlap when adding an edge, which affects the strengths of the four relationships. Formally, let s_i/e_i denote the start/execution time of vertex v_i. Then, we use $\Delta_{i,j}$ to denote the time overlap factor:

$$\Delta_{i,j} = \begin{cases} 0, & s_i + e_i < s_j \\ \frac{s_i+e_i-s_j}{e_j}, & s_i + e_i \geq s_j \end{cases} \quad (1)$$

Note that, when the learning model converges, we directly replace the start/end time with the model outputs.

3.2 Graph Construction

We formulate the procedure of graph construction using the extracted features (vertices) and relationships between vertices. As shown in Figure 2, the workload graph is defined as G = (V, E), where V is the vertex matrix with each row representing the feature vector of an operator, and E is the edge matrix with each entry (i, j) representing the relationships between v_i and v_j. Next, we present the details of constructing the graph for a given workload. We will discuss how to incrementally update the graph in Section 5.1.

Step 1 - Vertex Feature Modeling. For each query, we first obtain the query plan from the optimizer, and then extract operator features from the query plan. Next for each vertex v_i, we extract operation type, predicates, operation cost, and sample bitmap features and generate a vector V_i. For example, Figure 2 shows the vertex matrix, where each row is feature vector of a vertex.

Step 2 - Edge Feature Modeling. For each pair (v_i, v_j), if they satisfy the four relationships and $\Delta_{i,j} \neq 0$, we add an edge between them and the weight ($\mu_{i,j} = E[i][j]$) is computed as follows.

(1) If v_i and v_j have a parent-child relationship, $\mu_{i,j} = \mu_{i,j} + \Delta_{i,j}\lambda_1$, where λ_1 is the weight of the parent-child relationship.

(2) If v_i and v_j have a data-sharing relationship, $\mu_{i,j} = \mu_{i,j} + \Delta_{i,j}\lambda_2$, where λ_2 is the weight of the data-sharing relationship.

(3) If v_i and v_j have a data-conflict relationship, $\mu_{i,j} = \mu_{i,j} + \Delta_{i,j}\lambda_3$, where λ_3 is the weight of the data-conflict relationship.

(4) If v_i and v_j have a resource competition relationship, we consider two aspects. Let P^m denote the set of parameters that control memory usage (e.g., `shared_buffer`, `work_mem`) and I/O bandwidth (e.g., `effective_io_concurrency`). The larger the memory usage is, the smaller the conflict is. Let P^c denote the set of parameters that control concurrency levels (e.g., `max_connections`). The larger the concurrency value is, the larger the conflict is. Thus we set the weight as $\mu_{i,j} = \mu_{i,j} + \Delta_{i,j} \prod_{p \in P^c} p \prod_{q \in P^m} (1 - q)$.

Figure 2 shows the edge matrix, where each cell value is the weight of an edge between two vertices. For example, there are data-sharing and resource-competition relationships between v_4 and v_5. $\mu_{4,5}$=$\Delta_{4,5}\lambda_2$+$\Delta_{4,5} \prod_{p \in P^c} p \prod_{q \in P^m} (1 - q)$=2.013, where $\Delta_{4,5}$ denotes the execution time overlap of v_4 and v_5, P^c and P^m denote parameters that control memory and I/O usage (e.g., work_mem) and concurrency levels (e.g., max_connections) respectively. Table 2 shows the values of P^c, P^m in PostgreSQL, normalized into (0,1).

4. PERFORMANCE PREDICTION

We use the vertex matrix and edge matrix to predict the query performance. There are two challenges in performance prediction for a concurrent workload. Firstly, the performance-related features are sparsely scattered in the graph (e.g., for a matrix of 1000×1000, many rows only have two 1s, meaning that the corresponding operator only has 2 directly related operators). Traditional graph traversal algorithms [24] either directly encode the graph or choose random paths, and may not extract useful features. Secondly, considering a graph with thousands of vertices in high concurrency scenarios, the prediction is of high dimension and traditional regression methods are time consuming. To address these problems, we propose a graph-based prediction model, which utilizes a graph embedding algorithm to resemble the local and global structures of each vertex and generalizes neural models to predict query performance.

In this section, we first introduce the architecture of our performance prediction model in Section 4.1 and then explain the graph embedding algorithm in Section 4.2 and the training strategy in Section 4.3; and finally we take execution time as an example to illustrate the details of our performance prediction model in Section 4.4.

4.1 Performance Prediction Model

To address the challenges in graph embedding and performance prediction, we propose a performance prediction model and the framework includes two parts. First, to profile and learn the structure information scattered across the graph, we propose the `Graph Embedding Network` that embeds the local graph structure for each vertex in the vertex matrix V and outputs an embedded matrix H. Second, to predict query performance based on the graph information, we propose the `Graph Prediction Network` that learns the mapping from the embedded matrix H to the performance metrics (e.g., execution time) in an operator level. Figure 3 shows the model architecture.

Graph Embedding Network aims to embed each vertex into a vertex vector that captures both the vertex features and edge features. Obviously, it is expensive to embed all edges into the vectors. To address this issue, we only embed the edges within K hops to the vertex. To this end, we embed each vertex based on the layer-wise propagation rule: for every graph embedding layer, it embeds the features and relationships of 1-hop neighbors of all the vertices in V. Thus by passing through K graph embedding layers, it embeds the local graph features within K hops for each

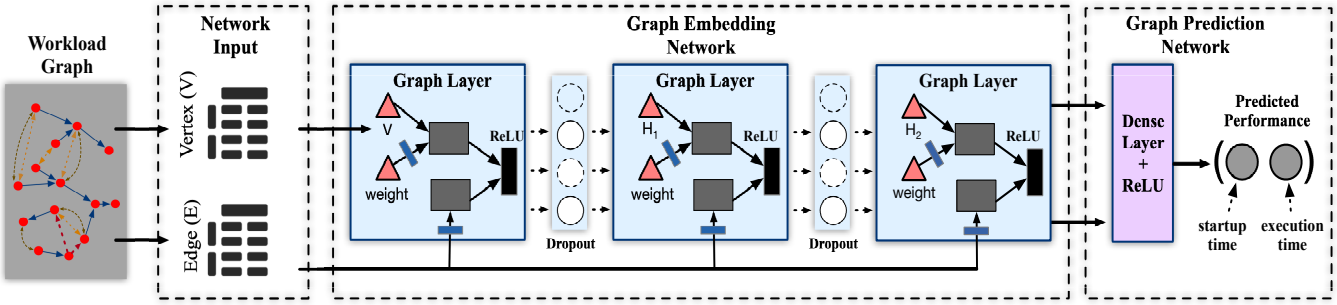


Figure 3: Graph-based Performance Prediction Model.

vertex v_i. We have shown that $K = 3$ achieves the best balance between accuracy and training time according to the experimental results (see Section 6.2.1). So we place three graph embedding layers in the `Graph Embedding Network`. Besides, we add a Dropout layer among every two graph embedding layers to avoid overfitting on the training set, which randomly sets some dimensions to 0 and leads to slight difference in prediction results each time.

The benefits of `Graph Embedding Network` are two-fold. (1) It effectively reduces computation work by extracting local structure instead of searching the whole graph. (2) It can concurrently process the local graphs of all the vertices.

Graph Prediction Network aims to predict the performance based on the embedding vector. We adopt a three-layer perceptron, including input/hidden/output layers, which is good at deriving performance features from the embedded matrix H provided by the *Graph embedding network*. The input layer maps H into preferable feature space H'; hidden layer conducts data abstraction on H' and outputs an abstracted matrix H''; and finally the output layer makes predictions on H'' and outputs the performance matrix P. Each vector P_i in P includes execution time, startup time, memory requirement, and CPU utilization.

For a concurrent workload, this architecture has the following advantages. (1) With large-scale matrix manipulations in the graph layers, we can concurrently predict execution time for each physical operator in the workload graph. (2) The training of the model is semi-supervised, and we only need to label the performance for a part of vertices in each sample, which saves the training time.

4.2 Graph Embedding Algorithm

We propose a graph embedding algorithm to embed the graph information in a vertex level. The idea is to obtain the local graph structures of each vertex, conduct non-linear mapping, and learn the way of embedding graph features by training network weights. And our algorithm is based on a layer-wise propagation rule [6] and we can denote the encoding procedure of the l_{th} layer as $H^l = \sigma^l(D^{-\frac{1}{2}}E'D^{-\frac{1}{2}}W^lH^{l-1})$, where H^l is the output of the l_{th} layer ($H^0 = V$), σ^l is the activation function, $D_i = [E_i^T E_{:,i}]H^{l-1}$ denotes the neighbor vector of vertex v_i within l hops, E' is the sum of edge matrix and identity matrix, and W^l is the weight matrix which is shared globally by all the input data. Next we will further explain our graph embedding algorithm and how to compute H^l in three steps.

Step 1. To prepare the input data, we add self-connection relationships into the edge matrix E, denoted as $E' = E + I$, where $I(i,j) = 1$ if $i = j$ and $I(i,j) = 0$ if $i \neq j$. In other words, vertex v_i is also connected with itself, and the edge weight is the largest, representing that the corresponding feature of v_i is most closely related to the performance.

Algorithm 1: GraphEmbedding(L, V, E)

Input: L: the graph embedding network ($[\sigma^l, W^l]$); V: a vertex matrix; E: an edge matrix

```
H = V;
I = an identity matrix with the same dimension as V;
foreach (σ^l, W^l) ∈ L do
    E = E + I; // Add self-connections
    Initialize the neighbor matrix D;
    foreach E_k ∈ E do
        D_k = [E_k^T E_{:,k}] H; // Compressed adjacent vector
    H = σ^l(D^{-1/2} E D^{-1/2} W^l H);
return H;
```

Step 2. To learn the neighborhood information, we place the local information (one hop) inside a high dimension matrix, denoted as $D_i = [E_i^T E_{:,i}]H^{l-1}$, which represents the neighbor features of each operator v_i within l hops. And then we multiply D, E with the weight matrix W^l, which further embeds the output into features by learning proper weight values from the loss values iteratively. Weights in W^l are globally shared in the network, i.e., we embed the local graphs of all the vertices with the common network weights, which can propagate differences of sub-graphs across the embedding model and avoid local optimum.

Step 3. To learn different performance metrics, we apply an activation function $\sigma^l(*)$ after each graph layer l. In above steps, we only conduct linear operations (e.g., matrix addition, multiplication), while the activation function conducts non-linear transformations (e.g., ReLU, Sigmoid) for complex learning tasks. For example, we may need to predict the startup time and execution time together. The startup time mainly depends on the execution cost of its sub-plan; while the execution time mainly depends on the execution features of the operator. So we add an activation function to provide more mixed and complex data processing methods to predict performance.

Neighbor Approximation. The cost of computing the neighbor matrix D is high for online prediction. Assuming the vertex number is $|V|$. Then for each vertex, the complexity of searching the neighbors is $\mathcal{O}(|V|^2)$, and the total complexity is $\mathcal{O}(|V|^3)$, which is too high for online prediction. Hence we approximate W using a truncated expansion by Chebyshev polynomials $T_k(V)$ to reduce dimension: $W^lH \approx \sum_{k=0}^{l} \theta'^k T_k(V)$, where l denotes the l_{th} graph layer; V is the vertex matrix; θ'^k is the weight values in the k_{th} layer; the Chebyshev polynomials $T_k(x)$ is recursively defined as $T_k(x) = 2T_{k-1}(x) - T_{k-2}(x)$ ($T_0(x) = 1, T_1(x) = x$). $T_k(x)$ has been proven to significantly reduce the dimensions of forward transform and save the embedding time [12].

4.3 Model Training

Training Data. The training data is a set of tuples $\langle Q, P^m, P^c, P\rangle$, where Q is a set of concurrent queries, P^m is the set of database configurations that control memory and I/O usage, P^c is the set of database configurations that control concurrency, and P is the real execution performance of queries in Q. Taking JOB [15] dataset as an example, we generate 20,187 SQL queries from the 113 query templates by randomly assembling {`Table`, `Join`, `Condition`, `Aggregate Operation`, `Column`}, and mix them into 1000 queries with concurrency from 10 to 100, which are split into training, validating, testing sets by 8:1:1. For each workload in the training set, we extract operators from corresponding physical plans, predict the performance of all the operators on the workload graph, and derive query performance from their root operators. With the operators, we train the prediction model by updating the layer weights with the loss values. After the performance converges or arriving the maximum training iterations, we test the performance on validation set. If the loss is still too large, we continue to train the model on this training set; otherwise, the model is converged on this workload and train the model using the next workload. After running out of training samples or the performance is steady, we test the prediction model on the testing set. We adopt batch gradient descent to enhance training efficiency, which predicts the performance for several workloads together, and computes the gradient and updates the layer weights in batch.

Loss function. Loss function is vital in graph embedding, which measures the accuracy of graph embedding algorithm between predicted performance and real performance. There are two challenges to design good loss functions. First, it may lead to overfitting to train the model using the real performance of all vertices. Second, it is time-consuming to get the real performance for all vertices. To address these challenges, for each workload sample, we randomly label 80% vertices with real performance, and use both the labeled and unlabeled vertices to compute the prediction loss [14]. (1) To reflect the prediction accuracy, we calculate the loss on labeled vertices using the Mean Squared Error (MSE): $L_0 = \frac{1}{B}\sum_{i=1}^{B}(f(V_i) - y_i)^2$, where B is the number of labeled vertices, $f(*)$ is the prediction model, V_i is the i-th labeled vertex, and y_i is the real performance. MSE measures average squared error and can minimize the distance from real values. (2) To smooth the prediction results across different vertices, we calculate the loss on unlabeled vertices as a Laplacian regularization term: $L_{reg} = \sum_{i,j}\mu_{i,j}||f(V_i) - f(V_j)||$, where $|f(V_i) - f(V_j)|$ denotes the L1 distance, in order to minimize the sum of absolute differences [7]. L_{reg} means that the larger the edge weight (V_i, V_j) is, the more similar the embedded features of V_i and V_j should be. So L_{reg} can work as a penalty to ensure that the output of two closely linked vertices do not differ too much. Finally, we denote the overall loss function as: $L_{total} = L_0 + \gamma L_{reg}$, where γ is a weight factor to tradeoff the importance between labeled and unlabeled vertices.

4.4 Prediction of Execution Time

Our graph embedding algorithm can be used to predict different types of query performance (e.g., execution time, memory requirement, and CPU utilization). Taking execution time as an example, for each workload, *Workload Graph* generates the workload graph G. First, we obtain the vertex matrix V and edge matrix E from G. Second, we input (V, E) into the graph embedding network and output the embedded matrix H. Third, the graph prediction network inputs H and outputs (st_i, et_i) for each operator v_i in the workload, where st_i is the startup time related to the commit time (we need to add the timestamp of different queries for comparison across queries) and et_i is the execution time of operator v_i. For each root vertex v_r in the query tree, the predicted end time of the query is $st_r + et_r$.

Remark: For every vertex, we do not directly predict its end time, because there may be overlap between startup/end time, which may lead to computation redundancy. For example, the end time of an operator may be the same as the start time of its parent. Instead, we predict the startup/execution time of each vertex and derive the end time of overall plan tree incrementally, where the end time of the sub-plan under vertex v_j is $st_j + et_j$.

5. SUPPORTING DYNAMIC WORKLOADS

Real-world workloads will dynamically change. As more queries are coming, we add the vertices of these queries into the graph, and the graph becomes larger and larger. It is too expensive to execute the whole workload prediction model every time the workload changes, because it not only wastes the memory size but also increases the prediction time. To address this issue, we propose two techniques. The first dynamically updates the graph by removing some unnecessary vertices (see Section 5.1), and the second compacts the graph by combining some vertices (see Section 5.2).

5.1 Graph Update and Prediction

There are two cases to trigger the graph update. The first case is adding a new query and the second case is removing some finished operators. Note that it is expensive to update the graph too frequently, because the graph updating also has overhead. To address this issue, we update the graph if a batch of operators Φ are finished. To obtain Φ, when we update the graph, we predict the performance of each operator and keep a constant of operators (e.g., 6) with the earliest finish time in Φ. If $\Phi = \phi$, we update the graph. Next we present the details on how to update the graph.

Initialization: (1) To estimate the execution progress, we predict the start/execution time for each operator v_i, denoted as (st_i, et_i), with which we estimate the execution progress of operators and queries. (2) To update the workload graph dynamically, we maintain a set Φ to store operators with the earliest end time.

Algorithm: We update the graph when (1) adding a new query or (2) a batch of operators finished, i.e., $\Phi = \phi$. Algorithm 2 shows the pseudo code.

Case 1: When submitting a new query, we update the graph in two steps. (1) We extract the query plan tree of the query, and for each operator in the query, we add it as a new vertex on the graph. For each vertex, we find correlated vertices from the graph and add edges to these vertices. We run the graph embedding and graph prediction algorithms to predict the performance. (2) We update Φ by adding the operators with the earliest finish time, which is predicted by the performance model.

Case 2: When $\Phi = \phi$, we remove the finished operators and the corresponding edges from the graph, run the embedding and prediction algorithms to predict the performance. We add the operators with the earliest finish time to Φ.

Algorithm 2: OnlinePrediction

Input: Q: a workload $\{Q_1, Q_2, \cdots, Q_{|Q|}\}$;
τ: the update degree

```
Initialize an operator set Φ ;
Initialize the workload graph G(V, E);
while true do
    if coming a new query Q_x then
        CompactGraph();
        UpdateGraph-Add(Q_x, G);
    if Φ = φ then
        UpdateGraph-Remove(G, Φ);
```

Function UpdateGraph-Add(Q_x, G)

Input: Q_x: a new workload; G: a workload graph

```
foreach (q_k, st_k, et_k) ∈ Q_x do
    Add feature vector of q_k into V;
    Add edges of q_k into E;
Predict the performance of G;
Φ = {τ vertices with the earliest end time};
return G, Φ;
```

Function UpdateGraph-Remove(G, Φ)

Input: G: a workload graph;
Φ: an operator set $\{(q_k, et_k)\}$

```
foreach (q_k, et_k) ∈ Φ do
    Remove edges and vertices of q_k from G;
Predict the performance of G;
Φ = {τ vertices with the earliest end time};
return G, Φ;
```

Incremental Prediction. Note after updating the graph, we re-predict the performance of vertices whose local graph changes. Generally only a small part of vertices may change in the workload graph and our embedding model can adapt to structural changes without re-training, and so we adopt an incremental way to update the performance. Let H denote the embedded matrix before update and ΔE denote the updated edges. We feed ΔE and V into the `Graph Embedding Network` and output an incremental matrix $\Delta H'$. We input $H' = H + \Delta H$ into `Graph Prediction Network`, and output the updated query performance.

5.2 Graph Compaction and Prediction

Even though we remove redundant vertices from the graph when workload changes, the graph still can be large with hundreds of queries and thousands of operators, which bring more computation and network training overhead. To address this issue, we can compact the graph. Intuitively, `GPredictor` predicts the performance of every operator in the workload, but we only need the overall performance of queries, depending on the longest path in the query tree from a leaf to the root. For operators not on the longest execution path of each query tree, we can compact them. Formally, we compact several vertices into a compound vertex if (1) the execution time of any two vertices has overlap; (2) any two vertices have no parent-child/data-sharing/data-conflict relationships. For example, in Figure 4, v_2, v_5, v_6 have time overlap and do not have direct relationships, so we can compact them into a compound vertex, denoted as c_1.

Graph Compaction Problem. Given a graph $G(V, E)$, we aim to generate a graph $G'(V', E')$ with the minimum vertex size, such that (1) any vertex in $v \in V$ must be in a compound vertex $v' \in V'$; (2) for any edge $(v, u) \in E$, there exists an edge $(v', u') \in E'$ where v and u are in compound vertices v' and u' respectively.

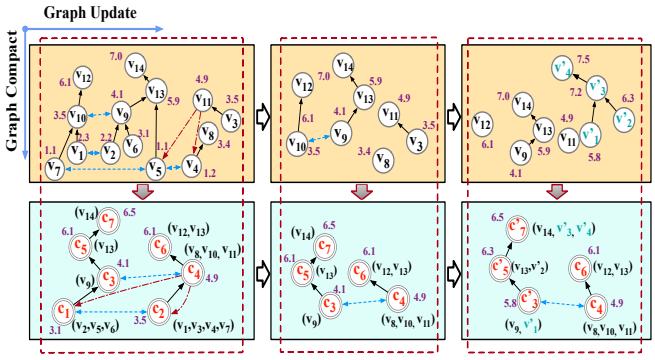


Figure 4: Graph update and compaction.

We can prove that the graph compaction problem is NP-hard by an induction from the problem of finding the maximum clique. Without loss of generality, we assume that all vertices have time overlap. Then we compact the vertices without edges into a compound vertex. To this end, we generate the complementary graph $\hat{G}$ of G, which has the same vertex set as G. For any two vertices G, if $(v, u) \in G$, then $(v, u) \notin \hat{G}$; otherwise $(v, u) \in \hat{G}$. Then we want to find all the cliques from $\hat{G}$, and the vertices in the same clique should be merged into the same compound vertex.

THEOREM 1. *The graph compaction problem is NP-hard.*

Compaction Algorithm. Since the graph compaction problem is NP-hard, we propose an approximate algorithm. The basic idea is to greedily merge the vertices into a compound vertex. To this end, we propose a two-step algorithm. The first step compacts the vertices with time overlap into the same group. The second step splits each group into compound vertices by cutting the vertices with edges.

Step 1: Grouping Vertices with Time Overlap. This step aims to cluster the vertices into different groups, where the vertices in the same graph have time overlap; while the vertices in different groups have no time overlap. In this way, the vertices in different groups cannot be compacted and we only need to compact the vertices in the same group by checking whether they have edges. To generate the groups, we can sort the vertices based on their start time. Then we scan the vertices in order. First we take the first vertex as a group which has a time span (start time, end time). Then we visit the second vertex. If the second vertex has time overlap with the first vertex, we group them together and update the time span; otherwise we take the second vertex as a new group. Iteratively, we can generate the groups. The time complexity of this step is $\mathcal{O}(|V| \log(|V|))$.

Step 2: Splitting Groups into Compound Vertices. This step aims to split the groups into compound vertices by cutting the edges between vertices. Given a group, we enumerate the vertices. First, we take the first vertex as a compound vertex. Then for the second vertex, if it has an edge with the first vertex; we take it as the second compound vertex; otherwise, we compact it into the first compound vertex. Considering the x-th vertex, we enumerate every compound vertex. If the vertex has no edge with the compound vertex, we compact it into this compound vertex; otherwise we check the next compound vertex. If the vertex cannot be compacted with any compound vertex, we take

Table 1: Datasets. JOB is an analytical workload, and TPC-C, XuetangX are transactional workloads.

Name	Table	Cardinality	Size (G)	#Query
JOB	22	20,581,773	1.1	20,187
TPC-C	9	70,785,180	1.3	912,176
XuetangX	14	69,826,530	11.5	22,000

it as a new compact vertex. Iteratively we can generate the compound vertices. The complexity is $\mathcal{O}(|V|^2)$.

Feature Encoding. For a compound vertex, we merge the origin features of those vertices in the compound vertex as the features of this compound vertex, where the `cost` is the maximum estimated cost of those operators; the `end time` is the latest of those operators; and the features of `operator type`, `predicate`, and `sample bitmap` are concatenated and flattened into a no-zero vector respectively. For an edge between two compound vertices, we enumerate edges between the vertices in the two compound vertices, and compute the sum of the weights of these edges as the weight of this edge. In this way, we can generate a compacted graph with compacted features and reduce the graph size. Then we can apply the graph embedding algorithm and graph prediction model on the compacted graph. Note that we adopt the same training method on the compact graphs.

6. EXPERIMENTS

We evaluate our proposed techniques and demonstrate the experimental results of our system from three aspects. (1) We evaluate the efficiency of `GPredictor`. (2) We compare the performance of `GPredictor` with three state-of-the-art methods [8, 20, 25]. (3) We evaluate the adaptability of `GPredictor` to show that `GPredictor` can adapt to different workloads and different hardware environments.

6.1 Experiment Setting

We implement `GPredictor` using Pytorch with TensorFlow as the backend, and process data using Python tools such as psycopg2, scikit-learn, and numpy to interact with databases and pre-process data. We conduct our experiments on a server with 128GB RAM, 5TB disk, 4.00GHz CPU. We use PostgreSQL v11.1.

Datasets. We use three datasets as shown in Table 1. (1) JOB is an OLAP benchmark. It uses a real-world dataset IMDB, which contains 22 tables joined on primary keys and foreign keys, and 20,187 queries, which are generated from the original 113 queries by randomly combing different tables, columns, and predicates; (2) TPC-C is an OLTP benchmark. It contains 9 tables and 912,176 queries, which are read and write operations; (3) XuetangX is a real-world OLTP benchmark (https://www.xuetangx.com/global) for online education. It has 14 tables with complex foreign key relations, and 22,000 queries extracted from the log.

Dynamic Workload. To simulate the changes of concurrent workloads in real scenarios, we combine queries of JOB, TPC-C, and XuetangX into different workloads based on the concurrency level respectively. For example, if the concurrency level is 10, we combine every 10 queries as a concurrent workload W_i and executes them together. `GPredictor` uses W_i to update the workload graph and predicts the performance of queries in the workload graph, and then actually run W_i to verify the performance.

Training Data. We train `GPredictor` to adapt to different workloads, concurrency levels, and resource limitations (e.g., buffer size). So each training sample is in the form of $\langle Q, P^m, P^c, P\rangle$, where Q is a set of queries, $|Q|$ equals to the concurrency level and $|Q| \in \{10, 50, 100\}$, P^m is the set of database configurations that control memory and I/O usage, P^c is the set of database configurations that control concurrency. Typical parameter combinations are shown in Table 2. For example, if the concurrency level is 50 and there are 418 workloads, we generate 20,900 training samples. We consider 10 typical database configurations (e.g., work_mem=512M, max_connections=3, effective_io_concurrency=100). We take 80% samples as the training set, 10% as the validation set, 10% as the test set.

Metrics. We evaluate `GPredictor` using three metrics. (1) `Error rate`: we use the mean squared error (MSE) to estimate the prediction accuracy, formalized as $\frac{1}{N}\sum_{i=1}^{N}(Y_i - f(X_i))^2$, where N denotes the number of labeled vertices in a workload, Y_i and $f(X_i)$ are the actual and predicted performance values of vertex v_i respectively; (2) `Prediction latency`: for a workload W_i, the prediction latency is the time of predicting the latency of all queries in the workload; (3) `Training time`: training time includes data preparation (e.g., extract query plans from the database), and model training. `GPredictor` finishes training if the error rate is lower than 0.01 or arriving the maximum iteration number.

Evaluation Techniques. We compare `GPredictor` with three state-of-the-art studies.

1. `BAL-based approach`: We implement the BAL-based approach, a state-of-the-art latency prediction method in workload level [8]. Buffer Access Latency (BAL) is the average latency of logical I/O operations. This approach collects the values of BAL and uses a linear regression to make continuous latency prediction during query execution. It takes a long time to get relatively accurate results (for days) and cannot adapt to workload changes quickly.

2. `DL-based approach`: We implement the deep-learning-based method, a state-of-the-art latency prediction method in workload level [20]. It uses deep learning to predict query latency under concurrency scenarios, including interactions between child operators, parent operators, and parallel plans. However, it adopts a pipeline structure (causing information loss) and fails to capture data sharing/conflict features and resource limitation information. In PostgreSQL v11.1, we extract 29 most widely-used operators, and build and train 29 neural units for each kind of operators.

3. `TLSTM-based cost model`: We implement the TLSTM-based cost model, a state-of-the-art cost estimation method in query level [25]. For each operator in the query plan, it uses a Long Short-term Memory (LSTM) unit to estimate the cost of the operator. The input of each LSTM unit is the features of the operator (e.g., predicates, tables) and the intermediate results of the child operators, and the output is the predicted execution cost. And these units are organized in the tree structure to get the total cost of the query. The strengths compared with the DL-based method are twofold. First, it encodes predicate features as the input of each LSTM unit, which provide detailed query features. Second, it uses the LSTM units to replace the neural network units, which can effectively avoid gradient explosion or vanishing.

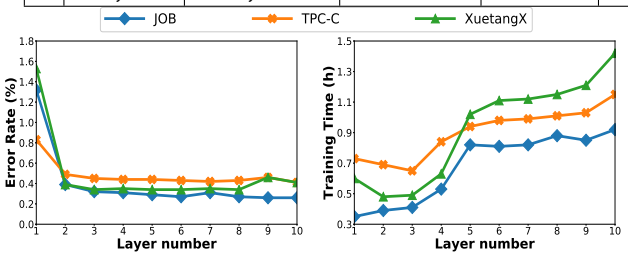
6.2 Evaluation on Our Techniques

6.2.1 *Evaluation on Embedding Layers*

Our graph embedding algorithm is based on the layer-wise propagation rule, in which the number of graph layers decides the maximum hops of connected vertices (local graph)

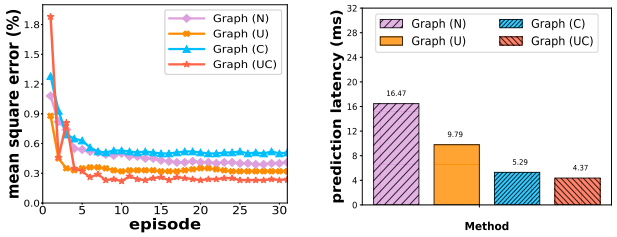
Table 2: Example Database Configurations for Training on PostgreSQL.

Id	P1 work_mem	P2 shared_buffers	P3 temp_buffers	P4 max_stack _depth	P5 maintenance _work_mem	P6 autovacuum _work_mem	P7 max_prepared _transactions	P8 max_ connections	P9 effective_io _concurrency
1	160	2.0	0.1	0.1	100	100	100	100	10
2	240	4.0	0.1	0.1	100	100	100	100	10
3	1,000	50	25	25	75	75	75	75	250
4	2,200	50	25	25	75	75	75	75	250
5	5,000	95	50	40	50	50	50	50	500
6	7,000	100	50	50	50	50	50	50	500
7	8,000	250	75	75	25	25	25	25	800
8	8,500	560	75	75	25	25	25	25	800
9	10,000	1,000	100	100	10	10	10	10	1,000
10	10,000	1,000	100	100	10	10	10	10	1,000



(a) Accuracy (b) Training Time

Figure 5: Evaluation on Embedding Layers on JOB



(a) Mean Square Error (b) Prediction Latency

Figure 6: Validation Error on JOB. `Graph(N)` denotes `GPredictor` without update/compaction; `Graph(U)` denotes update only; `Graph(C)` denotes compaction only; `Graph(UC)` denotes `GPredictor` with update and compaction. Concurrency level is 50.

that can be used in graph embedding. Hence we evaluate the impact of different numbers of graph layers to the performance of our prediction model. Figure 5 demonstrates the results and we make the following observations.

First, `GPredictor` has similar accuracy trend on three different workloads: when the layer number is smaller than or equal to 3, the error rate obviously goes down by increasing the layer number. But when the layer number is bigger than 3, the error rate will not obviously decrease. Second, when the accuracy of `GPredictor` goes steady (layer number $>$ 3), the average error of different workloads is less than 0.5%: 0.29% for JOB, 0.44% for TPC-C, and 0.38% for XuetangX. Our method achieves small errors, because (1) `GPredictor` captures the "potential" key correlations between concurrent queries (data passing/data sharing/data conflict/resource competition) to better reflect the execution features for the prediction model; (2) `GPredictor` embeds local structures using graph convolution filter and captures the complex correlations between concurrent queries. It can better embed global and local features of the graph, and so there is less information loss. Our method achieves the best performance on JOB, because (1) the queries are complex (with nested structures and many joins) and the system noise is trivial compared with the real execution time; (2) the concurrency level is relatively low (less than 5). So there are fewer correlations between queries and it is easier to predict. Third, the training time of `GPredictor` on three workloads is around 1 hour. With increasing the number of layers, the training time increases, because `GPredictor` needs to train larger neural networks. For XuetangX, one-layer graph takes longer time than 2-layer and 3-layer graphs, because one-layer graph network mainly considers the directed connected vertices, and the side effects of other vertices cannot be embedded, and one-layer graph is hard to converge without hidden layers for feature abstraction.

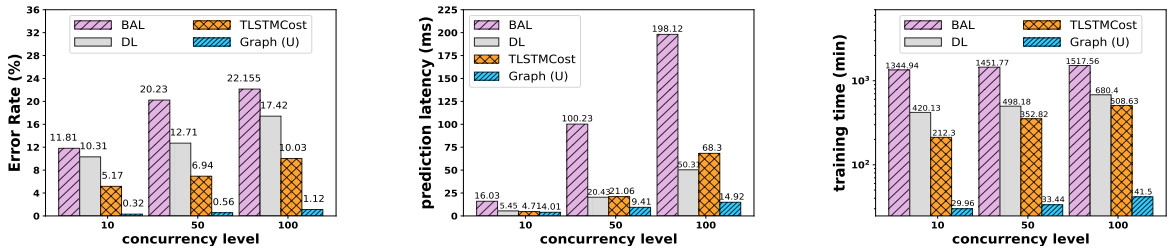
Summary. When the number of graph embedding layers is 3, our model works the best for trade-off between error rate and training time. Thus we choose 3 in our model.

6.2.2 *Evaluation on Graph Update and Compaction*

Next we evaluate our graph update and compaction techniques. We compare the accuracy and prediction latency of four methods, `GPredictor` without graph update or compaction (denoted as `Graph(N)`), `GPredictor` with graph update only (denoted as `Graph(U)`), `GPredictor` with graph compaction only (denoted as `Graph(C)`), and `GPredictor` with graph update and compaction (denoted as `Graph(UC)`). We train each method with the training set of JOB on PostgreSQL and the validation results are shown in Figure 6.

Training time. First, `Graph(U)` converges fastest (around 9 episodes) and takes the least training time (around 20 minutes). By predicting the performance for a query for multiple times (a batch of operators finish or new queries income) and taking the average error rate as the loss value, our graph update algorithm can dynamically remove useless vertex and edge information from the graph and iteratively gain more accurate results within the same training times. Second, `Graph(C)` converges slower than `Graph(U)` for two reasons. (1) Generally a compound vertex connects to more edges than a vertex denoting an operator, whose local graph structures are also harder to embed. But `Graph(C)` can effectively reduce the graph scale and so converges faster than `Graph(N)`. (2) `Graph(C)` only predicts the total execution time and gains one loss value for optimizing network weights, while `Graph(U)` predicts for every query and gains more loss values. Hence, `Graph(UC)` converges slower than `Graph(U)` but faster than `Graph(C)`.

Error rate. First, the four methods all achieve relatively low error rate (lower than 0.6% after convergence). The reasons are two-fold. (1) We encode performance-related features in different dimensions like data, resources, and time overlap, which can capture differences between database scenarios, cut down the impact of outliers caused by feature lack, and adapt to validation data. (2) The graph model is good at representing the complex relationships between concurrent queries, and with the deep graph embedding network we can capture the performance-related structure features efficiently. Second, `Graph(UC)` achieves the lowest error rate. On the one hand, by updating the graph it reduces the ef-



(a) Prediction Error (b) Prediction Latency (c) Training Time

Figure 7: Performance Comparison with Baselines on the JOB Workloads on PostgreSQL.

Table 3: Prediction Errors on the Test Set of JOB workloads. The Concurrency Level is 50.

Methods	median	90th	95th	99th	max	mean
BAL	18.31	24.10	25.43	25.79	26.23	20.23
DL	11.28	16.37	16.82	19.37	21.61	12.71
TLSTMCost	5.53	9.36	10.53	12.36	13.39	6.94
Graph(U)	**0.23**	**0.92**	**1.31**	**2.41**	**3.78**	**0.56**

Table 4: Prediction Errors on the Test Set of TPC-C Workloads. The Concurrency Level is 50.

Methods	median	90th	95th	99th	max	mean
BAL	25.28	29.17	29.51	29.92	30.13	26.81
DL	16.51	20.12	21.18	23.61	25.71	17.16
TLSTMCost	9.75	15.36	16.71	17.03	17.44	10.81
Graph(U)	**0.41**	**2.03**	**2.23**	**2.43**	**2.56**	**0.46**

fects of noise data (useless vertex and edges) and predicts well on the training set; on the other, by compacting the graph it generates different simplified graphs, avoids overfitting, and works well on the validation set.

Prediction latency. Prediction latency is vital to online prediction. Figure 6(b) shows the average prediction latency on the validation set of JOB. First, they all achieve relatively low latency (lower than 17 ms). And the reasons are two-fold. (1) The prediction model of GPredictor embeds and predicts performance in graph level, where all the vertices share the same set of network weights. (2) Rather than encoding the whole graph, the embedding algorithm of GPredictor selects the local graph features for each vertex. Second, compared with Graph(N), Graph(U) reduces the prediction latency by removing useless vertices and edges by about 40.56%; and Graph(C) further reduces the latency by compacting independent vertices by around 67.88%. So with both update and compaction, we can effectively reduce the prediction latency, with 4.37 ms in average.

Summary. GPredictor with update requires less training time and gains lower error rate; while GPredictor with compaction gets slightly higher error rates, but achieves higher prediction efficiency. By combing update and compaction, we balance between error rate and prediction latency.

6.3 Performance Comparison

We compare GPredictor with three state-of-the-art methods, BAL-based method (BAL) [8], DL-based method [20] (DL), and TLSTM-based method [25] (TLSTMCost). BAL estimates the average buffer access latency and uses linear regression to predict query latency for concurrent queries. DL adopts a plan-structured neural network to predict performance for single query, and TLSTMCost uses a tree-structured LSTM network to predict the cardinality and cost for single queries. We first compare prediction accuracy, prediction time, and training time of these methods on the test workloads of JOB. The results are shown in Figure 6.2.1, and then we verify the results on the test workloads of TPC-C and XuetangX respectively. The prediction results are shown in Tables 3-5, and we have the following observations.

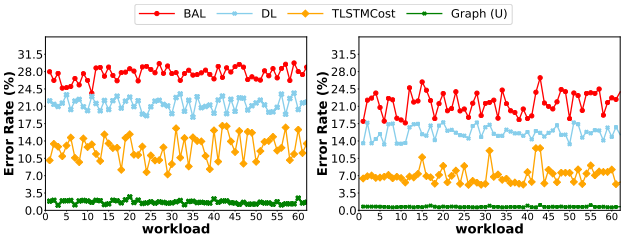
Table 5: Prediction Errors on the Test Set of XuetangX Workloads. The Concurrency Level is 50.

Methods	median	90th	95th	99th	max	mean
BAL	28.11	30.78	31.13	32.61	33.11	28.94
DL	10.98	14.32	15.75	16.38	17.61	11.83
TLSTMCost	12.65	16.83	17.03	17.91	18.41	14.71
Graph(U)	**0.51**	**1.93**	**2.07**	**2.35**	**2.99**	**0.68**

Error rate. GPredictor outperforms the other methods in all cases. In Figure 6.2.1, for different concurrency levels, GPredictor has the lowest error rate, around 29.9x lower than BAL, 22.5x lower than DL, and 11.4x lower than TLSTMCost on the JOB workloads. The reasons are two-fold. First, the workload graph of GPredictor encodes operator vertices features and relationships between concurrent queries so as to catch complex query correlations. Second, the graph embedding network in GPredictor can represent the workload graph structure and improve the generality.

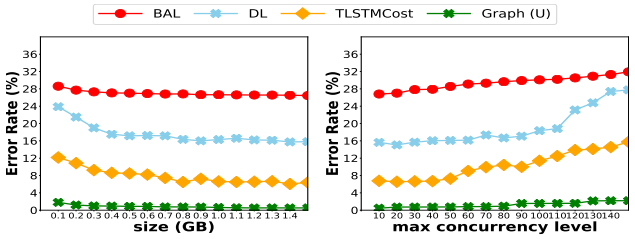
BAL has the worst performance and the reasons are two-fold. First, it only takes buffer access latency into consideration, and linear regression cannot learn the complex correlations between BAL and actual latency. For example, with the same value of BAL, query latency may increase if there is data conflict. Besides, DL and TLSTMCost consider more operator features and can obtain the plan-structural information, therefore achieving better accuracy than BAL. But they cannot directly encode concurrent factors like data share/resource contentions, which may increase query latency of JOB by over 20%, compared with the latency of serial execution. And the accuracy of TLSTMCost is better than DL for two reasons. (1) TLSTMCost considers the predicate and cardinality features, which are important to reflect the data the query accesses. (2) TLSTMCost uses LSTM units. With extra memory channels, LSTM better learn new knowledge and reserve old knowledge, compared with the neural units used in DL.

Prediction Latency. As Figure 6.2.1, when the concurrency level equals 10, GPredictor takes the least prediction time; and when the concurrency level increases, GPredictor outperforms all the other methods. For GPredictor, the prediction model concurrently predicts the execution time of all the vertices. It embeds the localized graphs for each vertex in the workload graph with global trainable weights in every graph embedding layers. So the total prediction time for all the concurrent queries is close the time to predict for the vertex with the largest localized graph. BAL requires the longest prediction time because it needs to predict the performance while executing the workload, and takes relatively long time to estimate the BAL values. For DL and TLSTMCost, on the one hand, they need to propagate intermediate data features across the query plan tree in a bottom-



(a) XuetangX to JOB (b) JOB to TPC-C

Figure 8: Adaptability on Different Workloads.



(a) Work_Mem (b) Max_Connections

Figure 9: Adaptability on Different Configurations.

up fashion, which takes relatively long time to gain the latency of a single query. On the other hand, the prediction model of DL/TLSTMCost only inputs a query at a time without batch processing and cannot predict with all the queries as a unit. DL takes less prediction time than TLSTMCost because (1) TLSTMCost needs to encode predicates; (2) LSTM unit takes relatively longer propagation time than a neural unit with similar functions.

Training Time. GPredictor outperforms all the other methods on training time. GPredictor outperforms BAL, DL, TLSTMCost, because, in graph level, GPredictor can learn more from one training episode. For each episode GPredictor trains the network with the loss value of all the concurrent queries and so can fast capture the general performance-related features. Graph(U) can effectively cut down the graph scale and takes less training time.

For BAL, it needs to extract system metrics (e.g., I/O frequency, block read/write latency) while repeatedly executing workloads, and takes days to learn a proper linear regression function. And for DL and TLSTMCost, their models can only predict for single queries and take longer time in forward propagation. With batch training, the training time of DL and TLSTMCost is reduced. But samples of a batch are independently predicted and the model cannot directly capture the correlations between them.

Summary. GPredictor outperforms state-of-the-arts methods in terms of error rate, prediction time, and training time. For error rate, GPredictor outperforms existing methods by 11–30 times. For prediction time, GPredictor outperforms existing methods by 20%–1,227%. For training time, GPredictor outperforms existing methods by 607%–4,383%.

6.4 Evaluation on Adaptability

6.4.1 *Varying Datasets*

We test the adaptability of GPredictor when dataset changes. We conduct two experiments on PostgreSQL: (1) Use the well-trained model on the training set of XuetangX to predict the test set of JOB. (2) Use the well-trained model on the training set of JOB to predict the test set of TPC-C. In each experiment we compare the error rate of GPredictor with BAL, DL, TLSTMCost. Figure 8 shows the results. Note that here we disable predicate encoding in GPredictor and TLSTMCost, because the patterns learned to parse a predicate are highly related to the dataset. For example, when data distribution changes, the cost of predicate "name like '%Bob' " may change and predicate encoding cannot identify without learning new patterns. Hence we leave the generalization of predicate encoding as the future work and here we verify whether GPredictor can adapt to new datasets and workloads without predicate features.

XuetangX to JOB. GPredictor outperforms the other three methods and achieves the lowest average error rate, about 16.9 times less than BAL, 10.0 times less than DL, and 7.6 times less than TLSTMCost. The main reasons are twofold. (1) In each vertex of GPredictor, the features may change with data (e.g., estimated cost) and queries (e.g., estimated cost, operation type, sample bitmap) and reflect differences in workloads. But the estimated cost of an operator may make different errors on JOB. GPredictor fine-tunes the prediction model based on a very small percent of JOB workloads, and achieves lowest error rate in testing, in which the average error rate is 1.63% and the variance is 0.34. (2) The workload graph of GPredictor encodes the four main relationships between queries and resource limits in JOB as edges, which can properly reflect the concurrency and resource information. And the prediction model of GPredictor can adapt to new graph structures.

JOB to TPC-C. We get similar results. GPredictor can easily migrate knowledge learned from one workload to another, because the queries in JOB contain complex plan structures like nested sub-queries, which have covered simple operators and sub-queries when training GPredictor. And GPredictor mainly needs to learn how to predict write operators and embed a graph with many more edges, since there are hybrid read/write queries and higher concurrency requirement in TPC-C.

6.4.2 *Varying Database Configurations*

Different database configurations may affect query performance significantly, especially under concurrent scenarios. Since memory and CPU are usually the performance bottleneck, we conduct two experiments on JOB workloads. (1) We test the models with different memory sizes. (2) We test the models with different max concurrency limitations. In PostgreSQL, the former is mainly controlled by *Work_Mem*, and the latter is mainly controlled by *Max_Connections*.

Memory size. *Work_Mem* controls the memory size allocated for each database connection, and we have the following observations. First, the error rate of every method decreases when *Work_Mem* gets larger, because with more data put inside memory, there is less I/O accesses and the query performance change becomes smaller, which is easier to predict. Second, GPredictor outperforms all the other methods with different *Work_Mem*, because GPredictor encodes resource limitations on each pair of edges, which can be embedded as graph structures in the graph embedding network. Thus GPredictor can adapt to different memory sizes. While BAL, DL, and TLSTMCost cannot.

Concurrency limitation. *Max_Connections* controls the maximum connection numbers a database instance can concurrently work, and we have the following observations. First, different from memory sizes, the error rate of every method increases when *Max_Connections* gets larger, because with more concurrent queries, there is higher possibility of competition (lock, resource) and the query performance gets harder to predict. Second, GPredictor still outperforms the baselines by encoding the lock/resource relationships, which affect the query performance when the number of concurrent

queries is larger than the limitation of max connections.

Summary. Our method `GPredictor` can adapt to different workloads and different database configurations, because `GPredictor` can capture the features of both queries and the database configurations while baseline methods cannot.

7. RELATED WORK

Performance Prediction for Single Queries. Most of existing query performance prediction methods predict performance on single queries, which can be divided into two categories. The first is statistics-based methods [29, 31]. These methods sample a small percent of data and collect the feature statistics, and predict the performance based on these features. However, the accuracy is highly related to samples and it is not easy to many high-quality samples. The second uses traditional machine learning [4]. It first selects most useful query features ranked by their correlations to the performance metrics, and then uses Support Vector Machines (SVMs) to predict the performance in the query level and proposes a hierarchical model in the operator level. With these hybrid ML models, it can balance between accuracy and adaptability.

However, these methods predict on single queries and mainly have two limitations. First, they do not consider concurrency features, which may affect the performance significantly under high-concurrency scenarios. Second, they take a long time to adapt to new scenarios and workloads without encoding the concurrency features.

Performance Prediction for Concurrent Queries. There are some traditional methods predicting performance for concurrent queries. Duggan et al. [8] proposed Buffer Access Latency (BAL) to capture the joint effects of disk and memory contention on query performance. And it used a linear regression model to predict query performance. However it takes a long time (several days) to train the model. Wu et al. [28] proposed an analytic model to predict dynamic workloads, which models the underlying database system as a queuing network for I/O and CPU requests. They predicted the execution time based on the merge result of the I/O and CPU requirements in concurrent queries. The third uses deep learning [20]. For a single query, it builds tree-structured network to predict query latency, which encodes interactions between child/parent operators.

However, these methods are all designed for analytical workloads. For transactional scenarios, the concurrency level is much higher than analytical workloads, and the queriers may have lock conflicts. Moreover, the workload may change with time more frequently, which are hard to predict.

There are multi-query optimization techniques (MQO) that generate a global query plan for the concurrent queries to share the intermediate results, which make the performance more predictable [23, 26, 2]. As traditional MQO identifies common sub-expressions greedily and may fail into local optimum [23], Unterbrunner et al [26] propose to index query predicates with probabilistic counting and join concurrent queries with *Clock Scan* to ensure predictable query latency. Zahid et al [2] propose to use regular path queries (RPQs) to explore path patterns in the query graph, which uses multi-join operations (e.g., ancestor-descendant) to estimate the intermediate costs.

Resource Usage Estimation. Resource usage estimation (e.g., CPU time, memory size) is crucial to various database tasks, e.g., cost estimation, performance prediction, and task scheduling [17, 10]. We classify existing resource usage prediction technologies into two classes. The first is the statistics-based resource models [1, 3, 27], which use combinations of weighted resource-related features, like memory requirements and the I/O times. However, these methods annot provide accurate resource estimation for complex operators like nested loop [15]. The second is machine-learning based methods. Li et al [17] proposed machine learning models (e.g., regression trees) to estimate resource usage. It can learn complex dependencies between query features and resources with machine learning. However, it did not consider many other factors that affect the resource usage (e.g., resource-related parameter, lock conflict). However these methods cannot be used to predict the query performance for concurrent queries.

Cost Estimation. Cost estimation is important to estimate query performance. We can classify cost estimation techniques into two classes. The first is traditional cost models [29, 18]. However, these methods cannot provide high-quality estimation for queries with multiple columns and multiple tables. The second is learning-based method [25]. It estimates query cost with a tree-structured LSTM model, which can embed the plan structures and gain higher accuracy with the cardinality estimated together. However these methods cannot be used to predict the query performance for concurrent queries.

Graph Neural Networks. Graph Neural Networks (`GNNs`) can be broadly classified into primitive `GNNs` [22], recurrent graph neural networks (`RecGNNs`) [9], graph auto-encoders (`GAEs`) [30, 5], graph convolution networks (`GCNs`) [14, 19]. Primitive `GNNs` [22] only support static graphs with fixed vertices and have to re-train when graph changes. `RecGNNs` [9] are computationally expensive for performance prediction, because they require to recurrently process the neighbor sequence to gain global features. `GAEs` [30, 5] embed the whole graph or generate new graph (decoder) by mapping vertices into latent representations (encoder), while in performance prediction we embed local graphs on each vertex. `GCNs` [14, 19] extract localized features and construct expressive representations for vertices, and thus are best suit for performance prediction. Traditional GCNs [14] use Laplacian matrix (symmetric) and only work for undirected graphs. Ma et al [19] propose to use a directed Laplacian matrix for directed graphs. But they only consider out-degree matrix and may cause information loss. So we propose to use the in/out-degree neighbor matrices to embed vertices in non-Euclidean domain. We have conducted experiments to compare with other GNN models, and our method outperforms them. To the best of our knowledge, this is the *first work* that uses `GNNs` to predict query performance.

8. CONCLUSION

In this paper we proposed a query performance prediction method for concurrent queries using a graph-embedding-based model. We used a graph-structured model to encode query features as vertices and the correlations between operators as edges. Then we proposed a prediction model to embed the performance-related graph features with a graph embedding network and predict the query performance using a three-layer deep learning model. We also proposed a graph update and compaction algorithm for supporting dynamic workloads and reducing graph size. Experimental results on real-world datasets showed that our method significantly outperformed the state-of-the-art approaches.

9. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, pages 181–192, 1999.

[2] Z. Abul-Basher. Multiple-query optimization of regular path queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1426–1430, 2017.

[3] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. The case for predictive database systems: Opportunities and challenges. In *CIDR*, pages 167–174, 2011.

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

[5] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann. Netgan: Generating graphs via random walks. In *ICML*, pages 609–618, 2018.

[6] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.

[7] J. A. Cadzow. Application of the L1 norm in signal processing. In *NSIP*, pages 15–18, 1999.

[8] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[9] C. Gallicchio and A. Micheli. Graph echo state networks. In *IJCNN*, pages 1–8, 2010.

[10] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[11] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, pages 357–368, 2009.

[12] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *CoRR*, abs/0912.3848, 2009.

[13] H. Harmouch and F. Naumann. Cardinality estimation: An experimental survey. *PVLDB*, 11(4):499–512, 2017.

[14] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[15] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[16] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.

[17] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[18] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, pages 153–166, 2015.

[19] Y. Ma, J. Hao, Y. Yang, H. Li, J. Jin, and G. Chen. Spectral-based graph convolutional network for directed graphs. *CoRR*, abs/1907.08990, 2019.

[20] R. C. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *PVLDB*, 12(11):1733–1746, 2019.

[21] C. Mishra and N. Koudas. The design of a query monitoring system. *ACM Trans. Database Syst.*, 34(1):1:1–1:51, 2009.

[22] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE TNN*, 20(1):61–80, 2009.

[23] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, 2(2):262–266, 1990.

[24] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*, pages 451–462, 2014.

[25] J. Sun and G. Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[26] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[27] V. Vapnik. *Statistical learning theory*. Wiley, 1998.

[28] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 6(10):925–936, 2013.

[29] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

[30] W. Yu, C. Zheng, W. Cheng, C. C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang. Learning deep network representations with adversarially regularized autoencoders. In *KDD*, pages 2663–2671, 2018.

[31] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *VLDB*, pages 289–300, 2005.