PreQR: Pre-training Representation for SQL Understanding

Xiu Tang
Zhejiang University
tangxiu@zju.edu.cn

Sai Wu*
Zhejiang University
wusai@zju.edu.cn

Mingli Song
Zhejiang University
brooksong@zju.edu.cn

Shanshan Ying
Alibaba Group
shanshan.ying@alibaba-inc.com

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

Gang Chen
Zhejiang University
cg@zju.edu.cn

ABSTRACT

Recently, the learning-based models are shown to outperform the conventional methods for many database tasks such as cardinality estimation, join order selection and performance tuning. However, most existing learning-based methods adopt the one-hot encoding for SQL query representation, unable to catch complicated semantic context, e.g. structure of query, database schema definition and distribution variance of columns. To address such above problem, we propose a novel pre-trained SQL representation model, called PreQR, which extends the language representation approach to SQL queries. We propose an automaton to encode the query structures, and apply a graph neural network to encode database schema information conditioned on the query. A new SQL encoder is then established by adopting the attention mechanism to support on-the-fly query-aware schema linking. Experimental results on real datasets show that replacing the one-hot encoding with our query representation can significantly improve the performances of existing learning-based models on several database tasks.

CCS CONCEPTS

• **Information systems** → *Structured Query Language*; • **Computing methodologies** → *Semantic networks.*

KEYWORDS

SQL query, pre-training representation, database schema

ACM Reference Format:
Xiu Tang, Sai Wu, Mingli Song, Shanshan Ying, Feifei Li, and Gang Chen. 2022. PreQR: Pre-training Representation for SQL Understanding. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3514221.3517878

1 INTRODUCTION

Due to the complex and diverse states of modern DBMS (Database Management System), databases are becoming more and more complex to optimize and maintain. Hence, a new line of research emerges which applies learning techniques to support many database tasks, such as cardinality estimation [23], join order selection [52], cost estimation [42] and performance tuning [28, 54]. Commercial DBMSs, such as Oracle and GaussDB [19], have started to redesign their core modules by leveraging various neural models, where SQL queries are normally accepted as one of the inputs. Therefore, almost all existing models need to address the same challenge—how to vectorize the representation of SQL queries.

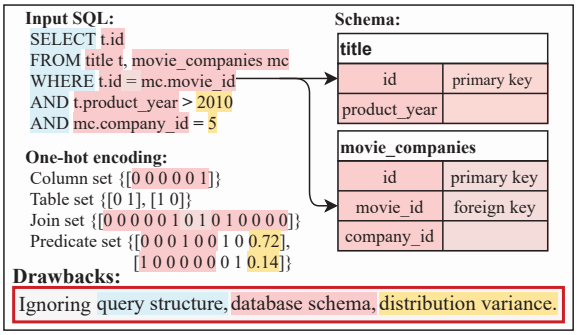


Figure 1: One-hot encoding fails to maintain the semantic information of SQL queries.

Most existing models adopt the one-hot encoding for SQL query representation [23, 42]. For example, MSCN [23] adopts one-hot encoding to represent database table and column in a query to estimate the cardinality. Sun and Li propose a learning-based cost model [42] by adopting the one-hot encoding to represent query plan node. As Figure 1 shows, the one-hot encoding fails to maintain the semantic information of SQL queries. The example depicts three drawbacks of existing one-hot encoding for SQL queries:

- The encoding of an SQL query is generated by simply concatenating the encodings of all clauses in the query. It contradicts the nature of the SQL by neglecting the structure information of the query.
- All tables and columns are encoded independently using a context-free mode, ignoring the definition of database schema (e.g., types and value domains). Moreover, primary keys and foreign keys are not explicitly identified, such as column *t.id* of table *title* and column *mc.movie_id* of table *movie_companies*.
- All values in an SQL query are normalized to [0, 1]. E.g., *2010* and *5* are encoded as 0.72 and 0.14 in our example. This approach ignores the distribution variance of columns *t.product_year* and *mc.movie_id*.

*Sai Wu is the corresponding author.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517878

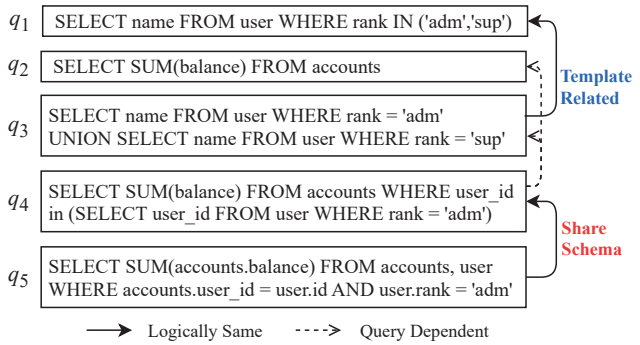


Figure 2: Challenges of SQL query embedding.

The above example demonstrates the key factors that affect the performance of existing database learning tasks. To address the drawbacks, we need a novel SQL encoding model to effectively incorporate semantic and context information into word vectors to improve the representation of structured data.

In fact, the language representation has been well studied by work on the NLP (Natural Language Processing), where generative pre-trained sentence encoders [9, 16, 37, 43] have contributed to significant performance improvements [44]. However, different from natural language, SQL is a structured query language, incurring new challenges as shown in Figure 2. In the example, all queries use different syntax and rules, and hence existing pre-trained language model will consider them to be far from each other. In fact, query q_3 is logically equivalent to query q_1, which can be easily identified by their query structures. Although query q_4 is different from queries q_2 and q_3, they are semantically related. Finally, query q_5 and q_4 are also logically the same, which can be discovered via involved database schema information.

To address the problems, the query representation scheme must have the knowledge on database schema and be sensitive to the query structure. Therefore, in this paper, we propose the PreQR (**Pre**-training **Q**uery **R**epresentation) model. PreQR is built on top of the BERT [9] by integrating the database schema, query structure and other domain knowledge. The pre-trained model only needs to be trained once for a database and can be used in various learning tasks. The major contributions of our paper are summarized as follows:

- We propose the first pretrained representation model for SQL understanding, which outperforms state-of-the-art (SOTA) results after fine-tuned on a series of database learning tasks.
- Our input embeddings can represent the query structure via matching sub-automaton states. Moreover, it can learn the distribution variance of columns during the matching process.
- A graph-structured model is adopted to encode database schema definition and the query-aware sub-graph can extract SQL-related schema information.
- We built an SQL encoder by leveraging the attention mechanism to identify the query-aware structural and schema information in an ad-hoc way.

We evaluate the performance of PreQR on multiple classic database learning tasks. Experimental results show that by replacing the encoding approach with our semantic representation, all models can outperform the SOTA results by a large margin.

2 SCHEME OVERVIEW

Figure 3 shows the architecture of PreQR, consisting of three modules: *Input Embedding*, *Query-Aware Schema*, and *SQLBERT*.

First, *Input Embedding* generates an initial encoding for an input query based on the SQL structure. We build an automaton to represent the SQL structure information. The list of states returned by the automaton is a flat representation of the input SQL query. Queries with similar structures (e.g., q_1 and q_3 in Figure 2) will share similar state representations. We also include the token embedding and position embedding to denote words and syntax used in the query. All three embeddings are concatenated together as the input for the *SQLBERT* model.

Next, *Query-Aware Schema* generates a representation for the database schema involved in the input query. Schema2Graph applies a graph neural network to represent the whole database schema by considering tables and columns as vertices in the graph. Because a query only involves a small portion of the database schema, we build a model to adaptively generate a query-aware embedding for the corresponding schema sub-graph.

Finally, *SQLBERT* merges the structural embedding from the *Input Embedding* module and schema embedding from the *Query-Aware Schema* module to generate the final SQL embedding. In particular, a bidirectional transformer is employed to generate the SQL encoding conditioned on both the schema and structure information. The training process is conducted as the masked language model in an unsupervised manner, which randomly masks some of the tokens from the input for prediction.

After the pre-training is done, PreQR model can be further fine-tuned for any particular database learning tasks. For example, to estimate the cardinality of a query, we can pick any SOTA model and replace the query encoding part with PreQR. The fine-tuning is performed by training the last layer of *SQLBERT* module together with the SOTA model.

3 PRE-TRAINING QUERY REPRESENTATION

In this section, we present the detailed designs of PreQR (Pre-training Query Representation) model.

3.1 Problem Definition

Any input to a neural model must be vectorized as a binary encoding. Using the notations listed in Table 1, the SQL embedding problem is formally defined as:

Definition 3.1. Given a database D and its schema S, we train a model $F : Q \times D \times S \rightarrow Y$. Q denotes the most frequent SQL query set on D and Y represents the 0-1 vectors of queries.

In this paper, to reduce the training overhead, we have the following assumptions. First, the data distribution of a database does not change over time, and hence, F is not affected by data insertion and deletion on the database. So we rewrite our model as $F : Q \times S \rightarrow Y$. Second, updating the schema definition is a rare case. Third, frequent queries can be identified by a limited number of query templates Q_t. This is true in real systems, where queries

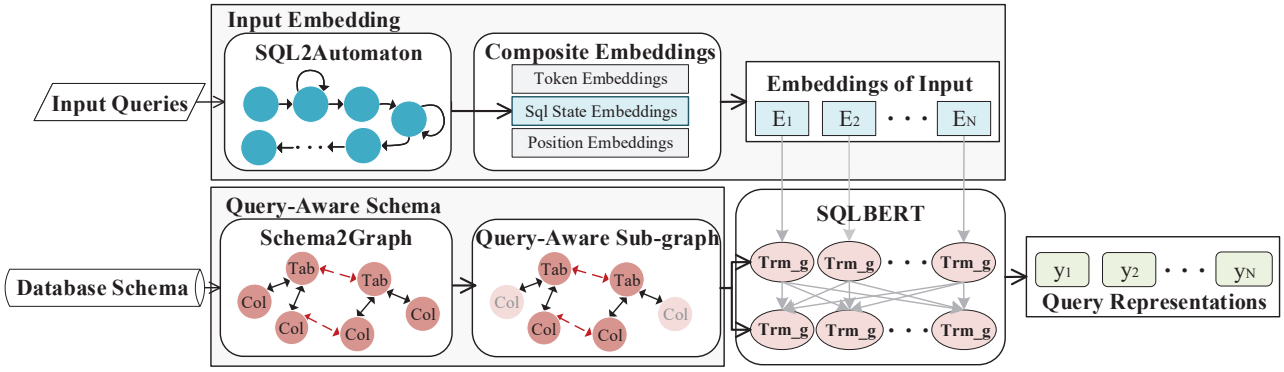


Figure 3: The architecture of PreQR. SQL2Automaton extracts query structure information as SQL state embedding, which is one of the initial embeddings. Schema2Graph represents database schema as a relation graph. SQLBERT extracts query representations by combining Query-Aware Sub-graph with structured content information.

Table 1: Summary of notations.

Notation	Definition
Q	the SQL query set
S	the database schema
Y	the SQL embedding
Trm_g	the Transformer module in SQLBERT
t_i	the tokens of SQL query q
a_i	the encoding of state in automaton
G_s	the database schema graph
v_i	the i-th vertex in schema graph
c_i	the i-th column vertex
t_i	the i-th table vertex
e_q	the BERT encoding of SQL query q
e_G	the global encoding of schema S
e_g	the sub-graph encoding of SQL-related schema

are always submitted via web forms. We also show how PreQR can be updated when the above assumptions are violated in Section 3.6.

3.2 Model Workflow

For an input query, *Input Embedding* generates its structural binary encodings. In particular, SQL2Automaton extracts the structure information of queries by establishing an automaton for query template set Q_t. Given any query q in Q, we can extract its corresponding template from Q_t and then guide the query through the automaton. Let $T = \{t_1, ..., t_n\}$ denote the tokens of SQL query q. We create an initial embedding for q as $\{e(t_1), ..., e(t_n)\}$, where $e(t_i)$ is defined as $(b(t_i), a(t_i), pos(t_i))$. $b(t_i)$ is the word embedding of t_i from BERT model. $a(t_i)$ denotes the 0-1 encoding of t_i's state in the automaton, and $pos(t_i)$ is t_i's position in q.

Different from *Input Embedding* module, *Query-aware Schema* catches the schema information of the query by building a graph convolutional model G to represent the database schema S. It then projects an input query q into the database schema to obtain a relational graph via attention Transformer $Trm' : S \times q \rightarrow e_g$. e_g is the 0-1 encoding for the schema information. Note that as a part of F, G is trained together with F.

Finally, *SQLBERT* tailors the BERT model [9] by accepting the query initial embedding (structured content information) as its input and integrating e_g (query-aware schema information) into its transformer Trm_g. The outputs Y are finalized as the query representations, which can be used as the input for database learning tasks.

In what follows, we show the details of our three modules: *Input Embedding*, *Query-Aware Schema* and *SQLBERT*.

3.3 Input Embedding

3.3.1 SQL-to-Automaton. To unambiguously represent SQL structure, we propose to convert query structure into a finite-state automaton. It was shown that automata can recognize syntactically well-formed strings [15]. Finite-state automata (FA) are machines with finite numbers of states. An FA can transit from one state to another in response to an input. The FA can well define the structure of an SQL query and its states are serializable. Its state transition information can optimize the prediction of mask words in the masked language modeling.

Specifically, the FA has a start state a_0 and a set of final states $\{a_{end}\}$. Let $T = \{t_1, ..., t_n\}$ denote the tokens of SQL query q. T is considered as an accepted sequence for the FA, only if we can find a path starting from a_0 to any final state in $\{a_{end}\}$. Table 2 shows the automaton of example queries in the Figure 2. We observe that queries q_1 and q_3 share a similar state sequence, indicating that the automaton can well represent the SQL structure. Since the automaton has a limited number of states, we use a one-hot encoding vector to denote each state a_i. For token sequence $T = \{t_1, ..., t_n\}$, we can concatenate all state vectors $\{a(t_i)\}_{i=1}^{n}$ as a FA encoding for the query.

In our approach, we cluster popular queries in a system and extract a query template for each group. The query template extraction step is semi-automatically. A hybrid distance metric is adopted to perform the query clustering. In particular, the column and table names are replaced with specific tokens, and the string, number, and category values are represented with different variations. We compute the string similarities between the query clauses and merge the similarities as cosine distance. Then, for each cluster, we create a template to denote most queries.

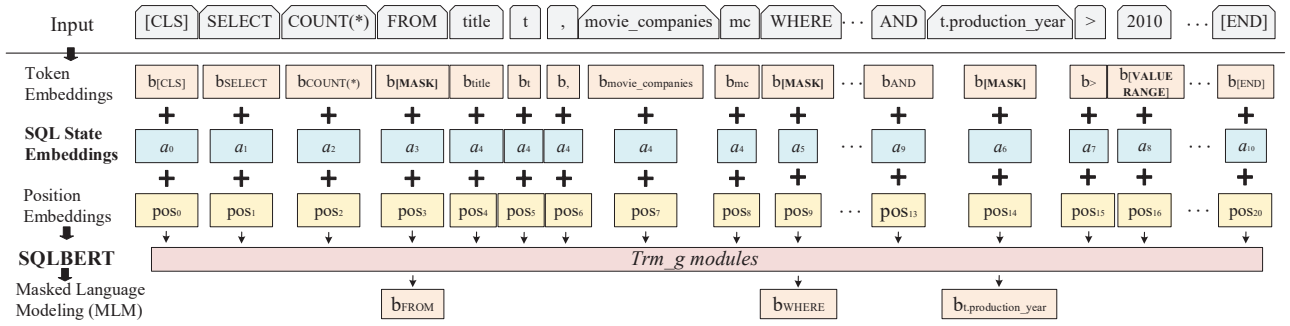


Figure 4: Query representation pre-training. Our query embeddings are refined gradually and finally can represent query words and structure information.

Table 2: SQL state embedding. For an incoming query, a sub-automaton matches the query. The sequence of state vectors of the automaton is the query's SQL State Embedding.

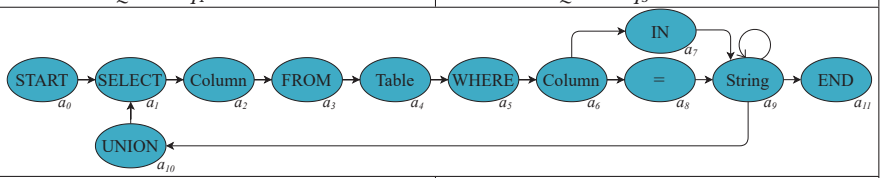
Input	Queries q_1 in Section 1	Queries q_3 in Section 1
Automaton Matching		
SQL State Embedding	$a = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_9, a_9, a_{11})$	$a = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_8, a_9, a_{10}, a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_8, a_9, a_{11})$

Table 3: The number of query templates in datasets.

Dataset	Num	Dataset	Num	Dataset	Num
JOB-light	1	WikiSQL	2	IIT Bombay	4
Synthetic	1	JOB	3	PocketData	4
Scale	1	UB Exam	3	StackOverflow	8

We build a sub-automaton for each template, and the final automaton is constructed by merging all sub-automatons. The merging process adopts the maximal prefix strategy. Table 3 shows the number of query templates required for datasets used in our experiments (for detailed descriptions, please refer to the experiment section). It indicates that a small number of templates can represent most SQL queries and hence, building FA and matching queries against the FA incurs negligible cost.

3.3.2 Composite Embeddings. As mentioned, for each token t_i in the SQL query, we create a composite embedding $e(t_i) = (b(t_i), a(t_i), pos(t_i))$. We visualize the embedding of a example query in Figure 4. The first token of an SQL is always a special classification token ([CLS]). In classification task, the hidden state of token [CLS] can be used as the aggregate representation for the whole token sequence. The end token of an SQL is also a special token ([END]). As shown in Figure 4, the composite embedding is created for each token by concatenating all three embeddings, which is then fed to the *SQLBERT* module to generate the final representation for the SQL.

Vocabularies of SQL are quite different from those of the natural language. To address the problem, we adopt two dictionaries. For input tokens, we use the WorldPiece embedding [47] with a 30,000 vocabulary. In the pre-training task, we randomly mask some percentage of the input markers and then predict these masked markers. For the mask layer, we use database-specific vocabularies, consisting of schema tokens and SQL keywords. In particular, for values (e.g., 2010 and 5 in the example query) in SQL queries, we transform them into discrete ranges and use range tokens to denote them. For example, we partition values of years into three ranges [1900, 2000], [2000, 2010] and [2010, 2020]. 2010 will be replaced by range token $year_3$, as it belongs to the third range.

3.4 Query-Aware Schema

Database schema plays an important role in data management and query optimization. It denotes the relationship between tables and columns. Popular schema structures include star schema, snowflake schema and chain schema. The PreQR model considers the database schema as a context during the query embedding process. The schema context is encoded using a graph convolutional model G, and dynamically updated by the input query.

At a high-level, our model has the following parts: (a) The schema is converted to a graph. (b) A graph embedding algorithm generates node representations in the schema graph. (c) The graph is softly pruned conditioned on the input query. (d) The query-aware schema subgraph is fed to our *SQLBERT* module. We will now elaborate on each part.

3.4.1 Schema-to-Graph. We first represent the database schema as a directed graph $G_s = \{V, E, R\}$ with nodes $v_i \in V$ and labeled

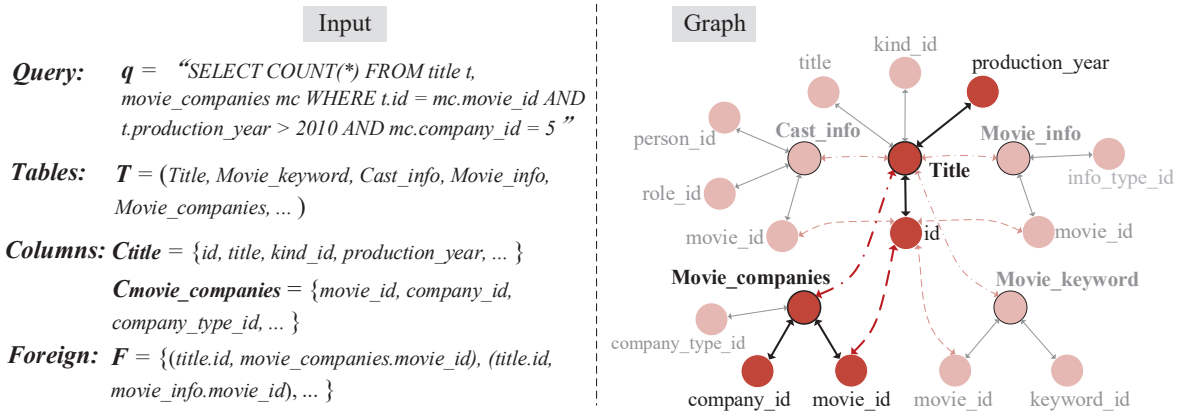


Figure 5: The query-aware sub-graph. Left: DB schema and table definitions. Right: A graph representation of the schema. Bold nodes are tables, other nodes are columns. Dashed red edges are foreign keys edges, black edges are table-column edges.

Table 4: Description of edge types used in the directed schema graph.

Type of (v_x, v_y)	Label of Edge r	Description
(Column, Column)	Same-Table	v_x and v_y belong to the same table.
	Foreign-Key-Column-Left	v_x is a foreign key for v_y.
	Foreign-Key-Column-Right	v_y is a foreign key for v_x.
(Column, Table)	Primary-Key-Left	v_x is the primary key of v_y.
	Belongs-To-Left	v_x is a column of v_y, but is not the primary key.
(Table, Column)	Primary-Key-Right	v_y is the primary key of v_x.
	Belongs-To-Right	v_y is a column of v_x, but is not the primary key.
(Table, Table)	Foreign-Key-Table-Left	Table v_x has a foreign key column in v_y.
	Foreign-Key-Table-Right	Table v_y has a foreign key column in v_x.
	Foreign-Key-Table-Both	v_x and v_y have foreign keys in both directions.

edges $(v_i, r, v_j) \in E$, where $r \in R$ is a relation type. V can be further classified into two types of vertices, table vertex and column vertex. E denotes four types of edges which can be further labeled by 10 tags from R, as shown in Table 4. For each pair of nodes v_x and v_y in the graph, Table 4 describes how edge (v_x, r, v_y) is created, where r is the label for the edge. If both v_x and v_y represent a column vertex, we will create an edge, when both v_x and v_y belongs to the same table or they have the primary-foreign key relationship. For table vertex v_t and one of its column vertex v_x, we create edge (v_t, r, v_x) and (v_x, r, v_t), and indicate whether v_x represents the primary key of the table. For two table vertices v_t and v'_t, an edge is created between them, only if they can be joined using primary-foreign key connection. If the schema definition is updated, our schema graph can be updated by adding or deleting nodes or edges correspondingly.

3.4.2 Graph Embedding Algorithm. The schema graph is further embedded in vertex level before fed to the *SQLBERT* module. We have two types of vertices, column vertices $V_C = \{c_1, ..., c_m\}$ and table vertices $V_T = \{t_1, ..., t_n\}$. For each vertex, we use function ρ to return the corresponding column names and table names. E.g., we have $\rho(c_i) = c_{i,1}, ..., c_{i,|c_i|}$, where $c_{i,j}$ denotes the j-th token of c_i's name. Specifically, for column vertex, the first token of its name is always set as column type (e.g., INT, VARCHAR and BOOL).

We use a bidirectional LSTM (BiLSTM) to generate the encodings for vertices:

$$\begin{aligned} \{(c_{i,j}^{fwd}, c_{i,j}^{rev}) | \forall c_{i,j} \in \rho(c_i)\} &= \mathcal{M}_{Col}(c_i), \\ \{(t_{i,j}^{fwd}, t_{i,j}^{rev}) | \forall t_{i,j} \in \rho(t_i)\} &= \mathcal{M}_{Tab}(t_i). \end{aligned} \tag{1}$$

$\mathcal{M}$ denotes the BiLSTM model, accepting the BERT embedding of each token as its initial input. We concatenate the output of the first and last time steps of $\mathcal{M}$ to form the initial node representation:

$$\begin{aligned} c_i^{init} &= Concat(c_{i,|c_i|}^{fwd}, c_{i,1}^{rev}), \\ t_i^{init} &= Concat(t_{i,|t_i|}^{fwd}, t_{i,1}^{rev}). \end{aligned} \tag{2}$$

At this point, vertex representations have been created as $v^{init} = \{c_i^{init}\}_1^{|C|} \cup \{t_i^{init}\}_1^{|T|}$. Now, we would like to imbue these representations with the information in the schema graph. We formulate the procedure of graph construction using the vertices and relationships between vertices. To learn the structure information of the graph, we apply a relational graph convolutional network G to embed the relationships of vertexes in V. Graph embedding network aims to embed each vertex into a vertex vector that captures both the vertex features and edge features. The idea is to conduct a non-linear mapping, and learn graph representation by training

network weights. We use the following propagation model for calculating the forward-pass update of a vertex denoted by v_i in the relational graph:

$$h_i^{(l+1)} = \sigma(\sum_{\bar{e}\in E}\sum_{j\in N_i^{\bar{e}}}\frac{1}{\lambda_{i,\bar{e}}}W_{\bar{e}}^{(l)}h_j^{(l)}), \tag{3}$$

where $h_i^{(l)}$ is the hidden state of vertex v_i in the l-th layer of the neural network. $N_i^{\bar{e}}$ denotes the set of neighbor vertices of v_i in the schema graph. $\lambda_{i,\bar{e}}$ is a problem-specific normalization constant that can either be learned or chosen in advance (e.g., $\lambda_{i,\bar{e}} = |N_i^{\bar{e}}|$). Eq.(3) accumulates transformed feature vectors of neighboring nodes through a normalized sum. It is a relation-specific transformation, which depends on the type and direction of an edge. We can then define the global representation e_G for schema graph $G_s = (V, E, R)$ as:

$$e_G = avg\_pool(\{h_i^L | \forall v_i \in V\}), \tag{4}$$

L is the last layer of the neural model. We use average pooling to reduce the embedding dimension to d_G.

We also intentionally create a self-connection edge for each vertex in the graph to ensure that the representation of a vertex at layer $l+1$ can also be informed by the corresponding representation at layer l. The neural model is trained by gradually updated weights based on Equation 3. In practice, Equation 3 is computed efficiently using sparse matrix multiplications to avoid explicit summation over neighborhoods.

3.4.3 Query-Aware Sub-graph. Database schema is sometimes very complex, consisting of hundreds of tables and thousands of columns, while one typical SQL query may only involve a few tables and columns. For instance, in Figure 5, only a sub-graph of the schema is involved when processing the example query. Therefore, instead of feeding the whole graph embedding to *SQLBERT*, we use a query-aware sub-graph embedding.

An attention module is adopted to achieve query-aware selections, which can be described as a mapping function for a query and a set of key-value pairs to an output. The architecture of query-aware sub-graph Transformer which includes the attention module is illustrated as the red rectangle in Figure 6. The particular attention in Transformer is called Scaled Dot-Product Attention. The input consists of query embedding, key embedding and value embedding. The output is computed as a weighted sum of the values, where the weight w_i assigned to each value is computed by a compatibility function of the query with the corresponding key.

In our case, the query embedding to the transformer is the BERT encoding of SQL query, denoted as e_q. Both keys and values are the global schema graph representation e_G. Namely, we try to find the correlations between schema and queries $F(e_q, e_G)$. We compute the dot products of the query with all keys, divided by $\sqrt{d_G}$, and apply a softmax function to obtain the weights w_i on values. Hence, the weights w_i represent relevance scores of the graph representation to the query, which is used to create a query-conditioned sub-graph representation e_g (defined in the next subsection). We compute the matrix of attention outputs as:

$$Attention(e_q, e_G) = softmax(\frac{e_q e_G^T}{\sqrt{d_G}})e_G. \tag{5}$$

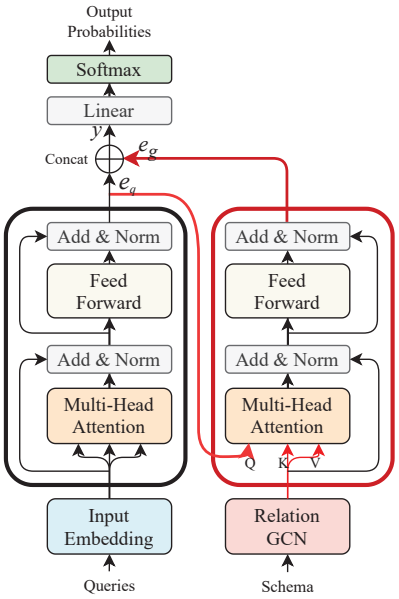


Figure 6: The architecture of Trm_g module. Our Trm_g combines the original transformer (black rectangle) with the query-aware sub-graph transformer (red rectangle).

That is, it is the node embeddings in the schema graph multiplied by the similarity weight w_i between the query and the graph. Figure 5 shows the query-aware sub-graph to our example query.

3.5 SQLBERT

3.5.1 Trm_g Modules. We discuss the Trm_g architecture, a variant of the transformer from BERT in this section. As shown in Figure 6, the Trm_g model includes the original transformer Trm (black rectangle) and our query-aware sub-graph transformer Trm' (red rectangle). The encoder in Transformer has the multi-head self-attention mechanism.

Our transformers are composed of a stack of $N = 4$ identical layers. Each layer has two sub-layers. The first one is a multi-head attention mechanism, and the second one is a position-wise fully connected feed-forward network. We employ a residual connection [13] for the two sub-layers, followed by a layer normalization layer [3]. At each step, the model is auto-regressive, consuming previously generated representation as additional input when generating the next one. We use multi-head attention, which allows the model to jointly attend to information from different representation subspaces at different positions. That is, the output of each sub-layer of original transformer Trm is

$$e_q = LayerNorm(e_q + Sublayer(e_q)). \tag{6}$$

Similarly, the output of each sub-layer of query-aware sub-graph transformer Trm' is

$$\begin{aligned} e_g &= LayerNorm(e_G + Attention(e_q, e_G)), \\ e_g &= LayerNorm(e_g + Forward(e_g)). \end{aligned} \tag{7}$$

We concatenate the sub-graph representation e_g to the output of the original Transformer e_q, so that each word is augmented with

Table 5: Update cost of PreQR model.

Case	Description	Time
Case 1	Incremental learning for the last layer of *SQLBERT*	15min
Case 2	Incremental Learning for the *Schema2Graph* part	3.5h
Case 3	Incremental learning for the *Input Embedding* module	6.7h
Case 4	Train from scratch	18.3h

the graph structure of the schema items that it is linked to. The final output of the *Trm_g* model is

$$y = Concat(e_q, e_g). \tag{8}$$

To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 256$.

3.5.2 Pre-training Process. The three modules are trained together by adopting the "masked LM" (MLM) pre-training approach [9]. In particular, for each incoming SQL query, we mask 15% of all tokens randomly. The obtained vector representation of the query is then fed to a softmax layer to predict the missing (masked) tokens from our pre-defined vocabulary. The training data generator chooses 15% of the token positions at random for prediction. If the i-th token is chosen, we replace the i-th token with: (i) the [MASK] token 80% of the time, (ii) a random token 10% of the time, and (iii) the unchanged i-th token 10% of the time. Then, y_i will be used to predict the original token with cross-entropy loss. After trained, the vector representations of SQL queries can be directly used for down-stream tasks, such as cardinality estimation and join order selection.

3.6 Model Update

Previous solution is based on the assumption that the database scheme does not change and the data distribution remains the same. In this section, we discuss how PreQR updates its model when such assumption is violated. Based on the update costs, we have four cases.

Case 1: The distribution of data changes significantly, but the database schema and query patterns remain the same. This only affects the token embedding part of the *Input Embedding* module. Our solution is to generate a few samples and perform an incremental training for the layer of *SQLBERT*, which can be accomplished within a few minutes.

Case 2: If the database schema is updated (namely, new tables or columns are created), we need to update the schema graph model G_s. G_s adopts a typical graph incremental training process [45], which normally lasts for a few hours.

Case 3: When query patterns change, we may need to update the FA to handle new queries. This requires a full retraining process for the *Input Embedding* module, while the *Query-Aware Schema* module is not affected. The training takes about 5-10 hours.

Case 4: Finally, to train a new embedding model for a database from scratch normally takes less than 20 hours.

We summarize the average update costs of different cases on our experiment datasets in Table 5.

Table 6: Distribution of joins.

Number of Joins	0	1	2	3	4	overall
Synthetic	1636	1407	1957	0	0	5000
Scale	100	100	100	100	100	100
JOB-light	0	3	32	23	12	70

4 EXPERIMENT

In this section, we conduct extensive experiments to evaluate our pre-training method for four downstream tasks, query clustering, query cardinality estimation, cost estimation and SQL-to-Text generation.

4.1 Datasets

4.1.1 Query Clustering. Clustering is an effective way to understand massive query logs [25], because it evaluates the similarity between queries, where query representation schemes will play an important role. In this test, we use two types of query workloads:

The first workload contains queries that are clustered according to logical equality: (i) student-authored queries released by IIT Bombay [6], (ii) student queries gathered at the University at Buffalo[1] (denoted as UB Exam), and (iii) SQL query logs of the Google+ app extracted from PocketData dataset [22].

The second workload contains queries with similarity scores based on the intersection of query result sets. In particular, we generated 600 random queries using the CH-benchmark [7], and classify them into three categories: logically equivalent queries, queries with same templates and irrelevant queries. And we use the ratios of common *row_id* between query result sets as the similarity scores to measure the similarity between queries.

4.1.2 Estimation Tasks. For query cardinality and cost estimation, we use the IMDB dataset, where columns and tables have high correlations, and therefore the dataset proves to be very challenging for all types of database tasks. It includes 22 tables, connected by the primary-foreign key relationships. We use two types of query workloads:

The first workload contains predicates with numeric attributes only [23]. It contains three sub-workloads with only numeric predicates: (i) a Synthetic workload with 5,000 unique queries containing both (conjunctive) equality and range predicates on non-key columns with zero to two joins; (ii) another synthetic workload Scale with 500 queries designed to show how the model generalizes to more joins; (iii) JOB-light, a workload derived from the Join Order Benchmark (JOB) [27] containing 70 queries which does not contain any predicates on strings nor disjunctions and limits to four joins at most. Table 6 shows the distribution of queries with respect to the number of joins in the three query workloads.

The second workload is from the JOB benchmark, where queries contain complex predicates on string attributes. The number of joins for queries in the JOB workload ranges from 4 to 28.

4.1.3 SQL-to-Text Generation. The goal of the SQL-to-Text task is to automatically generate natural language descriptions that explain the meaning of a given SQL query. This task is critical to

[1] http://odin.cse.buffalo.edu/public_data/2016-UB-Exam-Queries.zip

the natural language interface of the database because it helps non-expert users to understand the esoteric SQL queries. For generation task, we evaluate our model on two datasets, WikiSQL [55] and StackOverflow [20]. (i) WikiSQL[2] consists of a corpus of 87,726 hand-annotated SQL query and natural language question pairs. (ii) StackOverflow[3] consists of 32,337 SQL query and natural language question pairs.

4.2 Evaluation Metrics

For query clustering, the similarity metrics are evaluated using BetaCV measure [53] and Normalized Discounted Cumulative Gain (NDCG) [21]. BetaCV is a standard clustering evaluation metric, a smaller value of BetaCV indicates a better clustering result. NDCG is a standard IR evaluation metric to measure ranking accuracy, a larger value of NDCG indicates a better ranking result. Since only CH workload have ground truth of query similarity ranking, we only use NDCG evaluation on CH workload.

For cardinality estimation and cost estimation tasks, we adopt the q-error metric, defined as:

$$qerror(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \frac{max(y_i, \hat{y_i})}{min(y_i, \hat{y_i})}, \tag{9}$$

where y denotes the ground-truth value and $\hat{y}$ is the predicted one.

For SQL-to-Text generation, we adopt the BLEU [33] score, a widely used benchmark for machine translation task, as our evaluation metric, which is defined based on the n-gram precision as follows:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n \log p_n), \tag{10}$$

where BP stands for the brevity penalty, w_n is the weight for the n-gram, p_n is the precision of the predicted n-grams.

4.3 Baseline Approaches

In this work, we denote the number of layers (i.e., Transformer blocks) as L, the hidden state size as H, and the number of self-attention heads as A. By default, we report results with a pre-defined configuration (L=4, H=256, A=4), where the number of total parameters of our neural model is about 40 millions.

4.3.1 Query Clustering. For query clustering task, we use the following five approaches to measure pairwise similarity[4] between queries as comparative baselines:

(i) Aouiche *et al.* [2] is the first work that proposes a pairwise similarity measurement between two SQL queries. Their target is to optimize view selection in data warehouses based on historical queries. They consider different SQL clauses, such as the selection, joins and group by clauses, and generate vectors for each query. The Hamming distance between those vectors is used to measure the similarity between two queries.

(ii) Aligon *et al.* [1] compare session similarity via investigating query similarity measurement. They identify selection and join parts as the most important features in a query followed by the group by terms. Jaccard coefficient is applied to calculate the set similarity between two queries.

(iii) Makiyama *et al.* [29] extract selection, joins, projection, from, group-by and order-by terms from queries separately and record their appearance frequency. They use the frequency of these terms to create a feature vector to compute the pairwise similarity of queries with cosine distance.

(iv) One-hotDis [23]: We use a one-hot encoding model to represent queries, and then compute the pairwise similarity of queries using cosine distance.

(v) Seq2SeqDis [12]: We use an attention-based Seq2Seq model to generate the embedding of queries, and then compute the pairwise similarity of queries with cosine distance.

For comparison, we directly calculate the pairwise cosine similarity of queries with PreQR encoding.

4.3.2 Estimation Tasks. For the two estimation tasks, the following four approaches are used as baselines. (i) PostgreSQL(PG): PostgreSQL estimates the cost and cardinality using statistics and cost models [36]. (ii) MSCN[5]: a CNN-based model is employed to perform cardinality estimation in query level [23]. (iii) LSTM[6]: A LSTM-based model is trained to estimate query cost, and can also be employed to estimate query cardinality [42]. (iv) NeuroCard[7]: A join cardinality estimator that builds a single neural density estimator over an entire database [50]. It's the state-of-the-art neural cardinality estimator for join queries.

Both MSCN and LSTM approaches adopt an optimization technique by employing bitmap samples. Interestingly, using the bitmap sampling as an example, we show that optimization tricks can benefit PreQR embedding approach as well.

To show the advantage of the PreQR model, we adopt a very simple 3-layer fully-connected (FC) model as our prediction model. For each query, its PreQR embedding is generated, concatenated with bitmaps and fed to the FC layers to generate the prediction result. The intuition is to prove that a well pre-trained representation can produce good enough results even with very simple prediction models.

4.3.3 SQL-to-Text Generation. For SQL-to-Text generation, we compare the encoding of PreQR model against the Seq2Seq, Tree2Seq and Graph2Seq:

(i) Seq2Seq[8]: We use three Seq2Seq models as our baselines. The first one is a basic attention-based Seq2Seq model [4]. The second one additionally introduces the copy mechanism (cp) in the decoder side [11] and the third one also applies the latent variable in the decoder side [12]. (ii) Tree2Seq[9]: Tree2Seq is a tree-to-sequence model [10]. It uses the SQL Parser tool to convert a SQL query into the tree structure which is fed to the Tree2Seq model. (iii) Graph2Seq[10]: Graph2Seq is a a graph-to-sequence model [48]. It represents the SQL query as a directed graph and then encodes the global structure information into node embeddings.

[2] https://github.com/salesforce/WikiSQL
[3] https://github.com/sriniiyer/codenn/tree/master/data/StackOverflow
[4] https://github.com/UBOdin/EttuBench
[5] https://github.com/andreaskipf/learnedcardinalities
[6] https://github.com/greatji/Learning-based-cost-estimator
[7] https://github.com/neurocard/
[8] https://github.com/guoday/Question-Generation-VAE
[9] https://github.com/tempra28/tree2seq
[10] https://github.com/IBM/SQL-to-Text

Table 7: Overall performance with the same training settings.

Task	Methods	Encoding Model	Distance Function	BetaCV (smaller is better)			NDCG
				IIT Bombay	UB Exam	PocketData	CH
Query Clustering	Aouiche [2]	Binary code	Hamming distance	0.577	0.923	0.893	0.131
	Aligon[1]	String set	Jaccard coefficient	0.535	0.799	0.898	0.120
	Makiyama [29]	Item frequency	Cosine similarity	0.665	0.897	0.879	0.214
	One-hotDis[23]	One-hot	Cosine similarity	0.565	0.852	0.883	0.191
	Seq2SeqDis[4]	RNN	Cosine similarity	0.459	0.761	0.801	0.584
	PreQRDis	**PreQR**	Cosine similarity	**0.387**	**0.622**	**0.752**	**0.710**

Task	Methods	Encoding Model	Estimation Model	Q-error (smaller is better)			
				JOB-light	Synthetic	Scale	JOB
Cardinality Estimation	PGCard [36]	No	No	174	154	568	10416
	MSCNCard [23]	One-hot	MLP	57.9	2.89	35.1	–
	LSTMCard [42]	LSTM	MLP	24.9	2.87	28.1	53.0
	PreQRCard	**PreQR**	MLP	11.5	2.86	25.8	**48.3**
	NeuroCard [50]	Neuro (Data)	–	2.33	6.25	21.1	–
	NeuroCard+PreQR	**Neuro+PreQR**	MLP	**2.16**	**2.83**	**18.5**	–
Cost Estimation	PGCost [36]	No	No	173	62.7	35.7	105
	MSCNCost [23]	One-hot	MLP	27.4	10.3	8.22	–
	LSTMCost [42]	LSTM	MLP	17	4.45	5.21	9.4
	PreQRCost	**PreQR**	MLP	**5.25**	**1.09**	**4.15**	**8.0**

Task	Methods	Encoder	Decoder	BLEU Score/% (larger is better)	
				WikiSQL	StackOverflow
SQL-to-Text Generation	Seq2Seq [4]	RNN	RNN	20.9	13.3
	Seq2Seq+cp [11]	RNN	RNN+cp	24.1	16.6
	Seq2Seq+cp+lv [12]	RNN	RNN+cp+lv	26.3	18.4
	Tree2Seq [10]	LSTM	RNN	26.7	17.0
	Graph2Seq [48]	Graph	RNN	29.3	19.9
	PreQR2Seq	**PreQR**	RNN	**32.1**	**21.1**

In this experiment, we just replace the query encoding part in the first Seq2Seq by PreQR encoding.

4.4 Results on Clustering Task

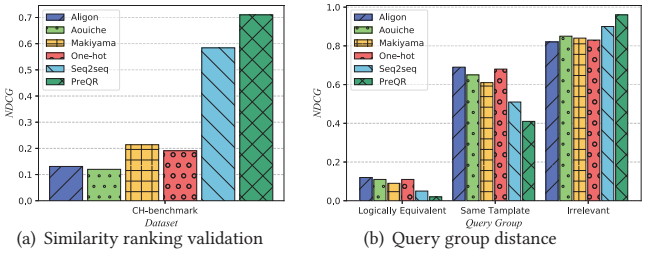
In what follows, we will report our experimental results and give a detailed analysis of the results. We summarize the overall results of PreQR and the baseline models in Table 7.

We report the detailed results on the cluster tasks in this subsection and the results on estimation and SQL-to-Text will be shown in Section 4.5 and 4.6.

In the clustering task, we compare the performance of PreQR with other similarity metrics. The experiment aims to evaluate whether query encoding generated by our method can capture the query similarity.

We first implement the similarity metrics and evaluate them using the BetaCV clustering validation measure. For query clustering, PreQR achieves the best results for all datasets under the BetaCV measure. BetaCV considers all queries in a dataset when computing the measure. So the results can represent the overall capability of representing query semantics. Then, we evaluate the performance of the similarity metrics on CH-benchmark for query similarity ranking.

Figure 7(a) shows the comparison results of query similarity ranking validation, we can see that PreQR outperforms the other

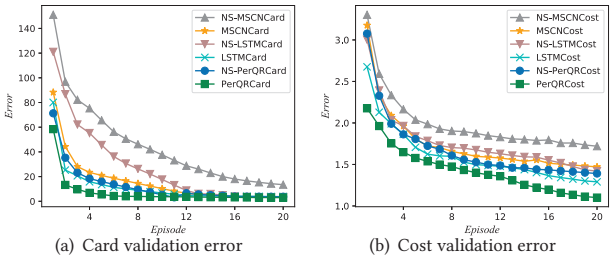


(a) Similarity ranking validation (b) Query group distance

Figure 7: Query similarity validation on CH workload.

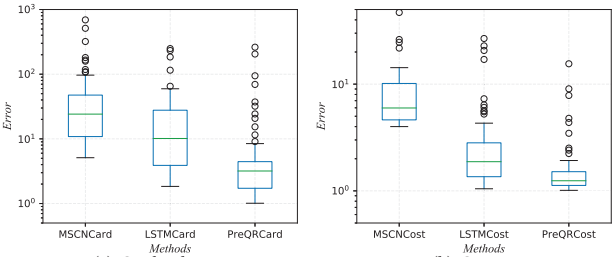
baseline methods in terms of NDCG measure. And we compute the average distances in different groups of CH queries, then report them in Figure 7(b). It is interesting to see that PreQR can identify equivalent queries with different SQL representations, and deliberately increase the distance between irrelevant queries. For queries with the same templates, PreQR returns a proper distance which is larger than the equivalent one. This indicates that PreQR does not only rely on the SQL representation, but also considers the schema information.

Existing approaches, such as Aouiche, Aligon and Makiyama, only focus on the template-based retrieval, which is difficult to distinguish queries with similar templates but different semantics.



(a) Card validation error (b) Cost validation error

Figure 8: Validation error on Synthetic workload.



(a) Cardinality (b) Cost

Figure 9: Estimation errors on JOB-light workload.

One-hot and Seq2Seq ignore the implicit database schema and the structure information of the queries, while PreQR maintains those information to discover the hidden semantics of SQL queries. The results show that our PreQR representation is a better metric to evaluate the semantic distance between SQL queries. Therefore, PreQR encoding can be applied to support many learning tasks on database system, such as query log analysis, recommendation and outlier detection.

4.5 Results on Estimation Tasks

Table 7 summarizes the results of our models and baselines on estimation tasks. We report the detailed results on the estimation tasks in this sub-section.

4.5.1 Effect on Numeric Predicates. We train the models and test them on the workloads with numeric predicates only (namely, Synthetic, Scale and JOB-light), and report the detail results in Table 8 and 9. For both cardinality estimation and cost estimation tasks, PreQR achieves a significant improvement.

On the JOB-light workload, PreQRCard outperforms MSCNCard by 9 times on max error and 5 times on the mean error for cardinality estimation respectively. PreQRCost outperforms MSCNCost by 5 times on mean error for cost estimation. We observe similar results on the Synthetic and Scale workload, indicating that PreQR outperforms the one-hot encoding used in the MSCN approaches by capturing the query structure and database schema information.

Different from the MSCN approach, the LSTM approaches introduce a query encoding technique using the LSTM model, similar to the one used in the language translation model. It is supposed to catch more SQL semantics. However, as shown in Table 8 and 9, on the JOB-light workload, PreQRCard outperforms LSTMCard by 13.4% on mean error for cardinality estimation, and PreQRCost outperforms PreQRCost by 11.75% on mean error for cost estimation.

Table 8: Cardinality errors on numeric workloads.

JOB-light	median	90th	95th	99th	max	mean
PGCard	7.93	164	1104	2912	3477	174
MSCNCard	3.82	78.4	362	927	1110	57.9
LSTMCard	3.73	50.8	157	256	289	24.9
PreQRCard	3.51	25.1	52.7	89.8	119	11.5
NeuroCard	1.49	4.04	5.43	10.5	16.5	2.33
NeuroCard+PreQR	**1.31**	**3.17**	**5.38**	**9.08**	**11.32**	**2.16**
Synthetic	median	90th	95th	99th	max	mean
PGCard	1.69	9.57	23.9	465	373901	154
MSCNCard	1.18	3.32	6.8	30.5	1322	2.89
LSTMCard	1.20	3.21	6.1	25.2	357	2.87
PreQRCard	1.17	3.13	5.6	24.8	316	2.86
NeuroCard	1.30	7.75	19.5	63.5	433	6.25
NeuroCard+PreQR	**1.15**	**3.05**	**5.4**	**24.5**	**298**	**2.83**
Scale	median	90th	95th	99th	max	mean
PGCard	2.59	200	540	1816	233863	568
MSCNCard	1.42	37.4	140	793	3666	35.1
LSTMCard	1.43	38.8	139	469	1892	28.1
PreQRCard	1.34	32.2	114	392	1638	25.8
NeuroCard	2.31	16.9	47.2	201	2034	21.1
NeuroCard+PreQR	**1.31**	**16.7**	**35.8**	**169**	**1542**	**18.5**

Table 9: Cost errors on numeric workloads.

JOB-light	median	90th	95th	99th	max	mean
PGCost	26.8	332	696	2740	3020	173
MSCNCost	4.75	11.3	40.1	563	987	27.4
LSTMCost	3.66	32.1	80.3	445	583	17
PreQRCost	**1.45**	**4.1**	**7.5**	**106**	**123**	**5.25**
Synthetic	median	90th	95th	99th	max	mean
PGCost	15.1	65.1	173	1200	8040	62.7
MSCNCost	3.14	7.43	18.1	65.8	739	10.3
LSTMCost	1.56	4.47	10.7	57.7	689	4.45
PreQRCost	**1.05**	**2.1**	**3.3**	**12.7**	**114**	**1.09**
Scale	median	90th	95th	99th	max	mean
PGCost	13.3	38.9	81.1	718	1473	35.7
MSCNCost	1.79	10.6	27.1	88.8	1027	8.22
LSTMCost	1.58	5.51	14.4	70.1	611	5.21
PreQRCost	**1.06**	**2.24**	**5.2**	**35.5**	**134**	**4.15**

On the other two workloads, PreQR-based approaches also achieve a superior performance than the LSTM-based ones. The strength of PreQR compared to the LSTM-based methods is two-fold. First, LSTM-based approaches consider the SQL queries as plain text, neglecting their structures, while PreQR can encode the structure information using automaton. Second, the PreQR encoding can help the representation model to learn query-related database schema information, which is missing in the LSTM-based approaches.

As mentioned, both MSCN and LSTM adopt an optimization technique based on bitmap sampling. We show that PreQR can also benefit from such tricks. In Figure 8, approaches with prefix "NS" indicates that no bitmap sampling is applied. All approaches can get an improvement by adopting the sampling trick. However, even without the optimization, the PreQR still outperforms others.

Table 10: Cardinality errors on the JOB workload (with strings).

Methods	median	90th	95th	99th	max	mean
PGCard	184	8303	34204	1.06e5	6.70e5	10416
LSTMCard	10.1	130	223	680	901	53.0
PreQRCard	**8.2**	**97**	**185**	**625**	**765**	**45.3**

Table 11: Cost errors on the JOB workload (with strings).

Methods	median	90th	95th	99th	max	mean
PGCost	4.90	80.8	104	3577	4920	105
LSTMCost	4.01	**14.9**	24.5	105	148	9.4
PreQRCost	**3.07**	15.4	**20.8**	**46.7**	**58.6**	**6.5**

Figure 9 shows the q-error variance of PreQR compared to our competitors on the JOB-light workload. The errors of PreQR methods are kept within a small range, while the MSCN-based approaches show a more unstable result. PreQR is capable of providing a consistent encoding performance.

PreQR can also be used in conjunction with data-driven estimators. We use the estimations of NeuroCard as preliminary results, and then train the error-correction model by using PreQR to further correct the results. That is, our prediction model is used to learn the gap between the Necrocard's results and their ground trues. As in Table 8, we can see that PreQR model can further improve the results of NeuroCard.

4.5.2 Effect on Mixed Predicates. We train the PG, LSTM, PreQR models and test them on the JOB workload with both string and numeric predicates. Because current MSCN model does not support string predicates, and NeuroCard does not support MIN operator, they are not included in this comparison.

We train the query representation on 100,000 queries with multiple joins. We take 90% of multi-table join queries as training data for estimation tasks and 10% of them as validation queries. The models are trained until the validation q-error will not decrease anymore, and then the trained models are evaluateed on JOB queries. The results for cardinality estimation are shown in Table 10 and the results for cost estimation are shown in Table 11.

It is interesting to observe that PreQR-based approaches outperform other approaches by a larger gap on the JOB workload than on the other numeric predicates only workloads. This is because LSTM encoding encodes the SQL key words and normal predicates together, fails to identify the structure information. Our PreQR approach adopts the BERT encoding as the basis encoding for string words, and also applies the automaton to catches the query template. Therefore, it maintains both the structure information and semantic information.

4.6 Results on SQL-to-Text Task

Table 7 summarizes the results of our models and baselines on SQL-to-Text Generation. We can see that on both datasets, the PreQR encoding performs significantly better than the Seq2Seq, Tree2Seq and Graph2Seq baselines.

PreQR model outperforms the Seq2Seq by 11.2% and 7.8% in terms of the BLEU score on the WikiSQL and StackOverflow datasets, respectively. Note that there are multiple improvements on the Seq2Seq model, such as copy mechanism (CP) and latent variable (LV). The PreQR does not adopt those optimization techniques. Instead, it only replaces the encoding part of the vanilla Seq2Seq model. PerQR can better model the query structure and database schema information. It can outperforms all Seq2Seq approaches, no matter with or without CP and LV optimizations.

We can see that Tree2Seq and Graph2Seq perform much better than the Seq2Seq models on the two datasets, since they both encode some structure information. Graph2Seq outperforms Tree2Seq, because by modeling the SQL query as a graph, Graph2Seq can precisely describe the correlations between keywords in a query. On the contrary, in the Tree2Seq model, the keyword node embedding aggregates the information of its descendants while losing the knowledge of siblings. However, they ignore the database schema information implicit in SQL query. PreQR maintains the SQL query structure while modeling the database schema information related to query. As shown in the results, our PreQR approach outperforms Graph2Seq by 2.8%, and outperforms Tree2Seq by 5.4% on the WikiSQL dataset. And a similar result is also obtained on the StackOverflow dataset. It demonstrates that PreQR encoding approach is a better representation for the query semantics.

If we delve into the query level, we find the PreQR model is better than others at interpreting two classes of queries: (i) complicated queries that have more than two predicates; and (ii) queries whose tables and columns have implicit relationships. It depicts that the PreQR model can better learn the correlation between the tables and columns by utilizing database schema information.

Table 12: Ablation test on cardinality and cost estimation.

Methods	JOB-light	Synthetic	Scale	JOB
BERTCard	36.5	3.53	39.2	58.4
PreQRNTCard	28.2	3.25	35.4	53.1
PreQRNACard	20.3	2.95	29.8	50.8
PreQRCard	**11.5**	**2.85**	**25.8**	**48.3**

Methods	JOB-light	Synthetic	Scale	JOB
BERTCost	7.50	1.51	9.42	13.1
PreQRNTCost	6.35	1.20	7.38	11.8
PreQRNACost	5.84	1.15	5.23	10.5
PreQRCost	**5.25**	**1.09**	**4.15**	**8.0**

Table 13: Ablation over model size on cost estimation.

Hyperparams			Test Set Q-Error			
#L	#H	#A	JOB-light	Synthetic	Scale	JOB
2	256	4	5.63	1.16	4.52	8.5
4	256	4	5.25	1.09	4.15	8.0
6	256	8	5.03	1.05	4.10	7.8
12	256	8	4.94	1.04	4.07	7.7

4.7 Ablation Studies

In this section, we perform ablation experiments over some facets of PreQR in order to better understand their relative importance.

4.7.1 Effect of Model Composition. We first perform ablation studies over the effect of model composition, and report the results in Table 12. We use "PreQRNA", "PreQRNT" and "BERT" to represent the PreQR model without automaton, Trm_g module and both automaton and Trm_g module, respectively.

Automaton vs No-Automaton. As shown in the results, PreQR is superior to the direct BERT and PreQRNA, indicating the effectiveness of applying automaton to represent query structure. Because the automaton encoding is more capable of representing query structure, it can learn a better structure representation with stronger generalization ability.

Trm_g vs No-Trm_g. We further discuss the necessity of schema graph module by comparing PreQR with PreQRNT. PreQR outperforms PreQRNT by a large margin, which proves the necessity of database schema information for SQL pre-training. One interesting observation is that PreQRNA outperforms PreQRNT, indicating that implicit schema information is more important than the query structure information.

In summary, both automaton module and graph schema module are very important in generating a proper representation for a query. They both contribute to the performance improvement of estimation tasks on database.

4.7.2 Effect of Model Size. Finally, we explore the effect of model size on fine-tuning task accuracy. We trained a number of PreQR models with a varied number of layers (#L), hidden units (#H), and attention heads (#A). The training process always adopts the same hyperparameters and configurations as described previously.

We show the effect of model size on cost estimation task in Table 13. In this table, we report the average accuracy from 5 random restarts of fine-tuning. We can see that larger models lead to a better accuracy improvement across all datasets. This result is consistent with other deep learning tasks, such as the language modeling task, where applying a deeper neural model always returns a better result. However, a deeper model incurs more training overhead, so we choose the configuration (L=4, H=256, A=4). When the model is fine-tuned directly on the downstream tasks and uses only a very small number of randomly initialized parameters, the task-specific models can benefit from a larger, more expressive pre-trained representations even when downstream task data are not enough.

5 RELATED WORK

SQL Representation. Neural networks are better function approximators that map featurized queries to predicted database performance. Most existing models focus on representing query content. MSCN adopts one-hot encoding to represent different words in queries [23]. The operators in query plan can be encoded by LSTM unit to represent the tree structure in queries [42]. The PreQR fine-tuning models have demonstrated superior estimation accuracy to representatives in this family. This is because while PreQR models the query structured content, it also pays attention to the implicit schema relationships in the queries. Moreover, complex predicates, especially string predicates, can also be easily handled through the powerful representation capabilities of the PreQR vocabulary.

Pretraining Model. Learning word representations from a large number of unlabeled text is a well-studied topic. While previous models focused on learning context-free word representations, such as Word2Vec [32], GloVe [34], later works focused more on learning context-sensitive word representations, such as ELMo [35], CoVe [31].

Learning natural language representations has been proven to be useful for various NLP tasks and has been widely used [9, 26, 35, 37, 49]. These pre-trained language models are learned on a large unsupervised text corpus, and then fine-tuned for specific NLP tasks, such as classification or natural language inference [46]. However, these methods cannot be directly applied to SQL embedding. Therefore, we designed a SQL pre-training model to make full use of structure-related information.

Learning table representations has also shown success in data management tasks on tables, such as TURL [8], TaBERT [51], TAPAS [14], ARM-Net [5]. These pre-trained models focused on learning the semantic parsing of database tables, and then fine-tuned for specific tasks, such as table knowledge matching and table expansion through tuning. Although we have different pre-training goals from them, our query representation and table representation can be used in cooperation to improve the accuracy of various downstream tasks.

Graph Neural Networks. Graph neural networks (GNNs) are widely used to represent graph data [24, 38, 39]. Original GNNs [38] only support static graphs with fixed nodes and must be re-trained when graph changes. Graph convolution networks (GCNs) [17, 18, 24] extract local features and construct the node representations.

The database schema can be modeled by using a relational graph model intuitively [40]. However, in our approach, it is shown that the new way to adopt schema information in SQL pre-training. We propose the query-aware sub-graph to obtain the part of SQL-related schema information.

Learned Database Components. A great deal of work has recently applied either classical ML or modern deep learning to various database components, e.g., cardinality estimators [23, 41, 42], and query optimization [30]. Our approach targets at training query embedding, and hence these learning works are orthogonal to our proposal, which can apply our query embedding as well.

6 CONCLUSION

In this paper, we proposed a pre-training representation (PreQR) for SQL understanding. Our embedding approach integrates both the structure of input SQL queries and database schema. Moreover, we present an SQL encoder to support query-aware schema linking by projecting the schema embedding using query context. The effective method we proposed can encode query semantics into the model to improve the model generalization. For a database, PreQR model only needs to be trained once and can be applied in various database learning tasks, such as cardinality estimation, cost estimation and join order selection. We conduct extensive experiments on several real-world datasets and database tasks. The experimental results showed that by replacing the encoders of existing models with our PreQR encoding, performances on various database tasks obtain a significant improvement.

ACKNOWLEDGMENT

This work is supported by the Zhejiang Provincial Natural Science Foundation (Grant No. LZ21F020007), National Natural Science Foundation of China (Grant No. 61872315, 62050099) and the Fundamental Research Funds for Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] Julien Aligon, Matteo Golfarelli, Patrick Marcel, Stefano Rizzi, and Elisa Turricchia. 2014. Similarity measures for OLAP sessions. *Knowledge and information systems* 39, 2 (2014), 463–489.

[2] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. 2006. Clustering-based materialized view selection in data warehouses. In *East European conference on advances in databases and information systems*. Springer, 81–95.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR 2015*.

[5] Shaofeng Cai, Kaiping Zheng, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Meihui Zhang. 2021. ARM-Net: Adaptive Relation Modeling Network for Structured Data. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 207–220. https://doi.org/10.1145/3448016.3457321

[6] Bikash Chandra, Bhupesh Chawda, Biplab Kar, KV Maheshwara Reddy, Shetal Shah, and S Sudarshan. 2015. Data generation for testing and grading SQL queries. *The VLDB Journal* 24, 6 (2015), 731–755.

[7] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[8] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2020. TURL: Table Understanding through Representation Learning. *Proc. VLDB Endow.* 14, 3 (2020), 307–319.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT 2019*. ACL, 4171–4186.

[10] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-Sequence Attentional Neural Machine Translation. In *ACL*.

[11] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. In *ACL*.

[12] Daya Guo, Yibo Sun, Duyu Tang, Nan Duan, Jian Yin, Hong Chi, James Cao, Peng Chen, and Ming Zhou. 2018. Question Generation from SQL Queries Improves Neural Semantic Parsing. In *EMNLP*. 1597–1607.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.

[14] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *ACL*. 4320–4333.

[15] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[16] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *ACL*. 328–339.

[17] Rongzhou Huang, Chuyin Huang, Yubao Liu, Genan Dai, and Weiyang Kong. 2020. LSGCN: Long Short-Term Traffic Prediction with Graph Convolutional Networks. In *IJCAI 2020*. 2355–2361.

[18] Zhichao Huang, Xutao Li, Yunming Ye, and Michael K. Ng. 2020. MR-GCN: Multi-Relational Graph Convolutional Networks based on Generalized Tensor Product. In *IJCAI 2020*. 1258–1264.

[19] Huawei. 2019. GaussDB Distributed Database. https://e.huawei.com/en/solutions/cloud-computing/big-data/gaussdb-distributed-database.

[20] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*. 2073–2083.

[21] Kalervo Järvelin and Jaana Kekäläinen. 2017. IR evaluation methods for retrieving highly relevant documents. In *ACM SIGIR Forum*, Vol. 51. 243–250.

[22] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. 2015. Pocket data: The need for TPC-MOBILE. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 8–25.

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[24] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[25] Gokhan Kul, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2018. Similarity metrics for SQL query clustering. *IEEE Transactions on Knowledge and Data Engineering* 30, 12 (2018), 2408–2420.

[26] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2020. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics* 36, 4 (2020), 1234–1240.

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[28] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.* 12 (2019), 2118–2130.

[29] Vitor Hirota Makiyama, Jordan Raddick, and Rafael DC Santos. 2015. Text Mining Applied to SQL Queries: A Case Study for the SDSS SkyServer.. In *SIMBig*. 66–72.

[30] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[31] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. 2017. Learned in Translation: Contextualized Word Vectors. In *NIPS*. 6294–6305.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS*. 3111–3119.

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*. 311–318.

[34] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *EMNLP 2014*. 1532–1543.

[35] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of NAACL-HLT*. 2227–2237.

[36] Postgresql. 1996. Postgresql. https://www.postgresql.org/.

[37] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

[38] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

[39] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*. Springer, 593–607.

[40] Richard Shin. 2019. Encoding database schemas with relation-aware self-attention for text-to-sql parsers. *arXiv preprint arXiv:1906.11790* (2019).

[41] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[42] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[44] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *ICLR*.

[45] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[46] Adina Williams, Nikita Nangia, and Samuel R. Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *NAACL-HLT*. 1112–1122.

[47] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[48] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. 2018. SQL-to-Text Generation with Graph-to-Sequence Model. In *EMNLP*. 931–936.

[49] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *NeurIPS 2019*. 5754–5764.

[50] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Peter Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2021), 61–73.

[51] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *ACL*. 8413–8426.

[52] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *36th IEEE International Conference on Data Engineering, ICDE 2020*. IEEE, 1297–1308.

[53] Mohammed J Zaki, Wagner Meira Jr, and Wagner Meira. 2014. *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press.

[54] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. ACM, 415–432.

[55] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).