Details of Model Architecture and Implementations

Anonymous Author(s)

1 TEXT-TO-SQL PARSING

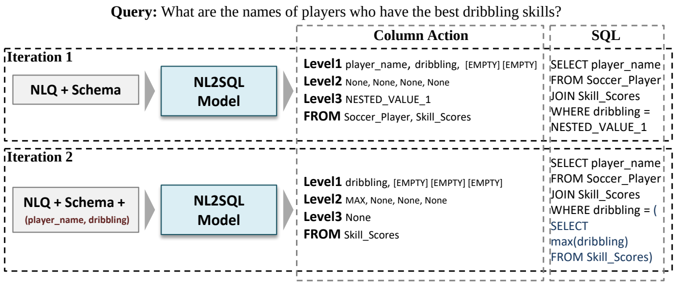
The network architecture of *CatSQL* is shown in Figure 2, which consists of three major components, an encoder, a column-action decoder and a from-clause decoder. The model runs recursively to generate the elements of each *Column Action Template* and the join paths of each atomic SQL query. The generation process terminates when no label for nesting or set operation ever exists in the generated blocks. To maintain the consistency between steps in the recursive process, we store the column names in the generated blocks at each iteration, called *Generation History*, and feed append them to the encoder input in the next iteration as shown in Table 3. An example of the complete process of translating a natural language query (NLQ) to SQL statement is shown in Table ??.

1.1 Encoder

Similar to existing models, the encoder utilizes a pretrained language model (*e.g.* BERT[1, 10]) to extract semantic features. We combine the user question, the selected schema, and the column names in the generation history as input. Given the question $\mathbf{q}$ with D tokens, S columns $\mathbf{C}$ in the schema, and L columns in the generation history, the input sequence is organized as follows:

$$[\mathsf{CLS}], q_1, \cdots, q_D, [\mathsf{SEP}], c_1^1, c_2^1, \cdots, [\mathsf{SEP}], c_1^2, \cdots, [\mathsf{SEP}], c_1^S, c_2^S, \\ \cdots, [\mathsf{SEP}], [\mathsf{EMPTY}], [\mathsf{SEP}], hc_1^1, hc_2^1, \cdots, hc_1^L, \cdots, hc_{K^L}^L, [\mathsf{SEP}],$$

where hc_k^l refers to the k-th token of the l-th column in the generation history, and K^L is the number of words in the L-th column name. Here, `[EMPTY]` is a special token, representing the end of a clause. We treat this symbol as an independent column, and append it to the end of the input sequence. The question, domain schema, and generation history is simply concatenated and the cross-dependencies can be learned by the self-attention mechanism of BERT. We use different type embeddings to distinguish question, schema and generation history. Specially, the type embedding of a column token in the generation history is a concatenation of two sub-vectors. The former part chosen from (`ORI`, `FROM`, `VALUE`, `UNION`, `INTERSECT`, `EXCEPT`) refers to the position of current sub-SQL, where `ORI` represents the first iteration. The latter part is an element from the set {`SELECT`, `FILTER`, `GROUPBY`, `ORDERBY`}.


Query: What are the names of players who have the best dribbling skills?
Column Action
SQL
Iteration 1
NLQ + Schema
NL2SQL Model
Level1 player_name, dribbling, [EMPTY] [EMPTY]
Level2 None, None, None, None
Level3 NESTED_VALUE_1
FROM Soccer_Player, Skill_Scores
Iteration 2
NLQ + Schema + (player_name, dribbling)
NL2SQL Model
Level1 dribbling, [EMPTY] [EMPTY] [EMPTY]
Level2 MAX, None, None, None
Level3 None
FROM Skill_Scores

```
SELECT player_name
FROM Soccer_Player
JOIN Skill_Scores
WHERE dribbling =
NESTED_VALUE_1
```

```
SELECT player_name
FROM Soccer_Player
JOIN Skill_Scores
WHERE dribbling = (
SELECT
max(dribbling)
FROM Skill_Scores)
```

Figure 1: NL2SQL parsing at runtime phase.

The BERT output is denoted as follows:

$$\mathbf{h}_{[\mathtt{CLS}]}, \mathbf{h}_1, \mathbf{h}_2, \cdots, \mathbf{h}_{D+1}, \mathbf{h}_{D+1}, \mathbf{h}_{c_1^1}, \mathbf{h}_{c_2^1}, \cdots, \mathbf{h}_{c_1^S}, \mathbf{h}_{c_2^S}, \cdots, \\ \mathbf{h}_{c_1^{S+1}}, \mathbf{h}_{c_2^{S+1}}, \mathbf{h}_{hc_1^1}, \cdots, \mathbf{h}_{hc_1^L}, \cdots, \mathbf{h}_{hc_{K^L}^L}, \mathbf{h}_{hc_{K^L+1}^L}$$

where the superscript c^{S+1} refers to `[EMPTY]` for writing convenience.

To obtain the embedding of each column, we feed the token embedding of each column name $\mathbf{c}^j$, including `[SEP]`, to an attention pooling layer as:

$$\mathbf{h}_{c^j} = \sum_{k=1}^{K_j+1} \alpha_k^c \mathbf{h}_{c_k^j}. \tag{1}$$

The computation of attention score α_k^c is as follows:

$$\alpha_k^c = \operatorname{softmax}(\mathbf{v}_\alpha^\top \tanh(\mathbf{W}_\alpha \mathbf{h}_{[\mathtt{CLS}]} + \mathbf{U}_\alpha \mathbf{h}_{c_k^j})), \tag{2}$$

where $\mathbf{W}_\alpha$, $\mathbf{U}_\alpha$, and $\mathbf{v}_\alpha$ are trainable matrices and vectors. The final embeddings of question and schema are calculated by filling the information of global context $\mathbf{h}_{[\mathtt{CLS}]}$ as:

$$\mathbf{h}_i^q = f_q(\mathbf{h}_{[\mathtt{CLS}]}, \mathbf{h}_{q_i}), \tag{3}$$

$$\mathbf{h}_j^c = f_c(\mathbf{h}_{[\mathtt{CLS}]}, \mathbf{h}_{c^j}), \tag{4}$$

where f_q, f_c are nonlinear activations.

Finally, we feed the token representations of question, column names, and generation history into another three attention pooling layers respectively, to obtain the final representations of question $\mathbf{h}_Q$, schema $\mathbf{h}_C$ and generation history $\mathbf{h}_H$.

1.2 Schema Linking

Schema Linking [2, 8] is proved to be essential for NL2SQL tasks. The idea is to learn the alignments between words in the question and column names in the database schema. Existing words (*i.e.RAT-SQL*) incorporate multiple relational transformer layers([8]) to capture such alignments. Though

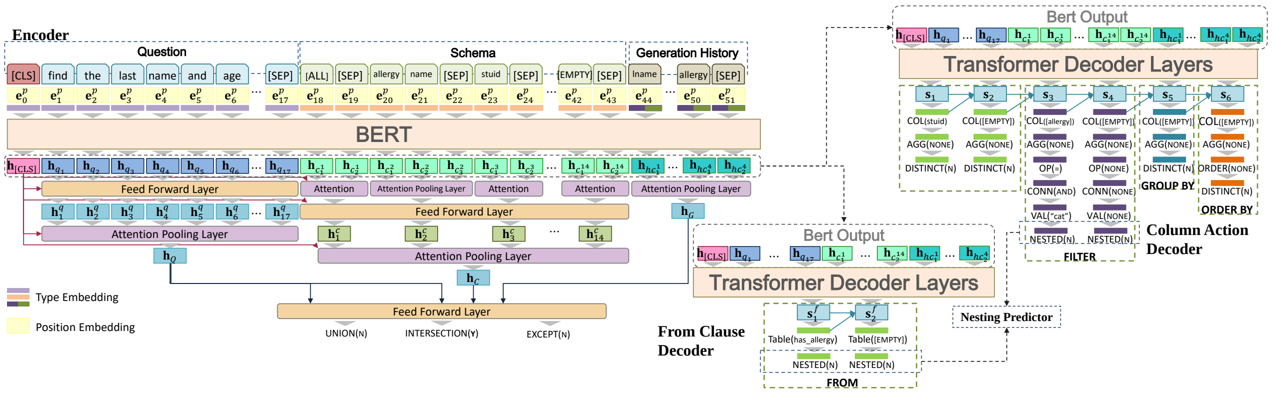
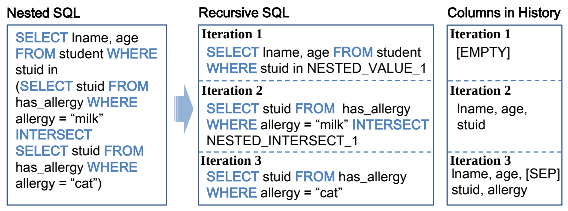


Figure 2: The conceptual architecture of *CatSQL*.



Nested SQL	Recursive SQL	Columns in History
SELECT lname, age FROM student WHERE stuid in (SELECT stuid FROM has_allergy WHERE allergy = "milk" INTERSECT SELECT stuid FROM has_allergy WHERE allergy = "cat")	Iteration 1 SELECT lname, age FROM student WHERE stuid in NESTED_VALUE_1	Iteration 1 [EMPTY]
	Iteration 2 SELECT stuid FROM has_allergy WHERE allergy = "milk" INTERSECT NESTED_INTERSECT_1	Iteration 2 lname, age, stuid
	Iteration 3 SELECT stuid FROM has_allergy WHERE allergy = "cat"	Iteration 3 lname, age, [SEP] stuid, allergy

Figure 3: Converting a nested structure into a non-nested form.

powerful, this implementation brings a large amount of parameters and significantly slows down the training/inference speed of the neural network models. In this work, we feed schema linking alignments to the pretrained model as the type embeddings, which are simply added to the input token embeddings. We extend the definition of schema linking alignments of RAT-SQL, but consider the alignments between column names in the generation history, words in the question and the columns in the database schema. The added linking types are listed in Table 1. It should be noted that, for each link from element x to y where $x \neq y$, there also exists a reversed link from y to x with the opposite type.

1.3 CAT Decoder

The column action decoder is composed of *transformer blocks* [6], and each block comprises two attention layers. The first attention layer is applied to learn semantic dependencies between the current state and previous outputs in a self-attention manner, while the other one utilizes context attention conditioned on source contextual states to refine the self-attention output. In our model, we use BERT outputs as the contextual features. The decoding process follows the order of (`SELECT`, `FILTER`, `GROUP BY`, `ORDER BY`), and the state changes when a `[EMPTY]` token occurs. At each decoding step, the decoder is fed with the concatenation of the previous column's representation and the type embedding of the current clause as input.

The output states of the Transformer decoder corresponds to the column sequence $\mathbf{y}^c = (y_1^c, y_2^c, \cdots, y_{D_c}^c)$ of *Column Action*. The hidden states of the last layer of the Transformer decoder is denoted by $(\mathbf{s}_1, \mathbf{s}_2, \cdots .\mathbf{s}_{D_c})$, where D_c is the length of column sequence. We leverage the pointer network [7] to identify the column to be selected. Concretely, at each step t, the probability of choosing column $\mathbf{c}^i$ as a *Column Action* element is estimated by:

$$p(y_t^C = i|\mathbf{q}, \mathrm{C}) = \mathrm{softmax}(\mathbf{v}_c^\top \tanh(\mathbf{W}_c\mathbf{s}_t + \mathbf{U}_c\mathbf{h}_i^c)), \quad (5)$$

where $\mathbf{W}_c$ and $\mathbf{U}_c$ are weight matrices, and $\mathbf{v}_c$ is a weight vector.

For the output elements in **Layer 2** of *Column Action*, excluding values, we formulate the element prediction task as a classification problem. Formally, the probability of selecting the j-th element at step t is computed as:

$$p(y_t^I = j|\mathbf{q}, \mathrm{C}) = \mathrm{softmax}(g(\mathbf{s}_t)), \quad (6)$$

where g is a non-linear transformation function to transform the hidden states to output-sized vectors. The computation of each element is executed independently with its own parameters. Predicting values for `WHERE` and `HAVING` clause is formulated as a task to determine the starting and ending positions of values in the question. Assuming the decoding timestep t is in the span of `WHERE` or `HAVING` clause, the probability of choosing position k of question $\mathbf{q}$ as the starting or ending position of a value is calculated as follows.

$$\begin{aligned} p(y_t^{Vstart} = k|\mathbf{q}, \mathrm{C}) &= \mathrm{softmax}(\mathbf{v}_{vs}^\top(\mathbf{W}_{vs}\mathbf{s}_t + \mathbf{U}_{vs}\mathbf{h}_k^q)), \\ p(y_t^{Vend} = k|\mathbf{q}, \mathrm{C}) &= \mathrm{softmax}(\mathbf{v}_{ve}^\top(\mathbf{W}_{ve}\mathbf{s}_t + \mathbf{U}_{ve}\mathbf{h}_k^q)), \end{aligned} \quad (7)$$

where $\mathbf{v}_{vs}$, $\mathbf{W}_{vs}$, $\mathbf{W}_{ve}$, $\mathbf{v}_{vs}$ $\mathbf{W}_{ve}$, and $\mathbf{U}_{ve}$ are trainable parameters. Checking the existence of a nested structure could be treated as a binary classification problem. And we feed $\mathbf{s}_t$

Table 1: Description of linking types. The links between elements with different types are symmetric and some reversed links are omitted here.

Type of x	Type of y	Link Type	Description
Column in Generation History	Column	SAME-COLUMN	x and y are the same column.
		SAME-TABLE	x and y belong to the same table.
		FOREIGN-KEY-C-F	x is a foreign key for y
		FOREIGN-KEY-C-R	y is a foreign key for x
Column in Generation History	Token in Question	EXACT-MATCH	y matches the name of x exactly.
		PARTIAL-MATCH	y matches the name of x partially.
		EXACT-COLUMN-VALUE	y matches a value in column x exactly.
		PARTIAL-COLUMN-VALUE	y matches a value in column x partially.
Column in Generation History	Column in Generation History	SAME-CLAUSE	x and y belong to the same clause in the same iteration.
		DIFF-CLAUSE	x and y belong to different clauses in the same iteration.
		DIFF-ITERATION	x and y belong to different iterations.

to a feed-forward layer with sigmoid activation function to estimate the probability.

The other clauses (**INTERSECT**/**EXCEPT**/**UNION** and **LIMIT**) are predicted based on the global representation $\mathbf{h}_G$ of the entire input, which is computed as:

$$\mathbf{h}^G = f_g(\mathbf{h}_Q, \mathbf{h}_C, \mathbf{h}_H), \tag{8}$$

where f_g is a feed-forward layer with non-linear activation. With the usage of the global representation, predicting the limit number is formulated as a classification problem. The probability of choosing k as the limit number is computed as:

$$p(y^l = k|\mathbf{q}, \mathrm{C}) = \mathrm{softmax}(g_l(\mathbf{h}_G)_k), \tag{9}$$

where g_l is a transformation function to generate an output-sized vector and superscript k refers to the k-th element of the output vector. We also feed $\mathbf{h}_G$ to another 3 feed-forward layers with sigmoid activation to estimate the probability whether the output SQL contains a **INTERSE CT** / **EXCEPT** / **UNION** clause of not.

1.4 From Clause Decoder

With regards to the `FROM` clause, we use another transformer decoder to predict the sequence of involved tables as the join path. Concretely, the from clause decoder takes the BERT outputs as source contextual information. At each step, the table embedding obtained at the previous step is also used for decoding. Similar to the column action decoder, we feed the decoder hidden states and table embeddings to a pointer network to identify the table to be selected at each timestep. The nesting in the `FROM` clause is also predicted with a binary classification layer at each decoding step.

1.5 Training and Inference

Before training, we preprocess the training samples to convert the nested structures into non-nested blocks as shown in Table 3 by traversing the AST tree of SQL with the depth-first order. Each non-nested block is regarded as an independent training sample.

The final training objective is a summation over all sub-task losses, which are calculated by the cross-entropy loss. It should be noted that the proposed model is designed for generic SQL grammar. When applied to simple dataset that only contains select-projection queries such as WikiSQL, we simply remove the from clause decoder and nesting predictor to achieve higher efficiency.

At runtime phase, the NL2SQL model sequentially generates the elements of *Column Action* with beam search [4], which is a greedy search algorithm. Facing multiple sub-tasks for various SQL elements, we extend the original beam search to support multiple beams. Concretely, we first apply the traditional beam search to the output of each component. Once the hypotheses of all component are generated, we merge all beams to construct the final hypothesis. The hypothesis score is computed by the summation over all sub-tasks.

2 TABLE SELECTION

Here, we introduce the approach to obtain representations of the query and tables. The architecture of the table selection model is shown in Figure 4.

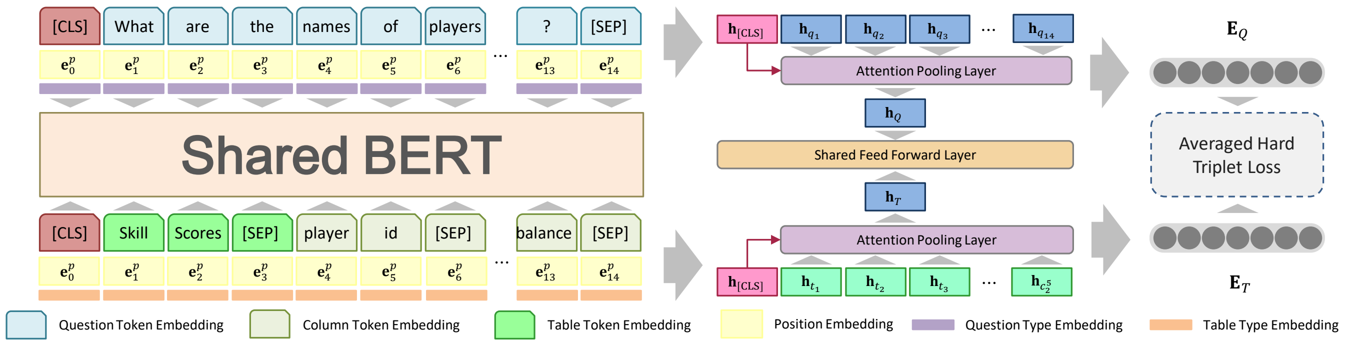


Figure 4: The architecture of the Table Selection Model.

2.1 Question Encoder.

Given a sequence of tokens $\mathbf{q} = (q_1, q_2, \cdots, q_D)$ of question Q, we first convert each token to an embedding vector with fixed length. Considering the high linguistic variability, we utilize BERT [1] as the backbone to transfer external linguistic knowledge into our encoder model. Specifically, the input question tokens are represented as: $([\mathrm{CLS}], q_1, q_2, \cdots, q_D, [\mathrm{SEP}])$, where [CLS] and [SEP] are special symbols of BERT added in the head and end of every input sequence. [CLS] is usually used for sentence classification, while we treat this token as the representation of the whole sequence. [SEP] is often used as the end symbol or a separator for multi-sequence inputs. BERT comprises multiple multi-head attention layers [6], capturing the correlations across words with bi-directional modeling. The output sequence of BERT is represented as: $(\mathbf{h}_{[\mathrm{CLS}]}, \mathbf{h}_{q_1}, \mathbf{h}_{q_2}, \cdots, \mathbf{h}_{q_{D+1}})$, where $\mathbf{h}_{q_{D+1}}$ is the alias of $\mathbf{h}_{[\mathrm{SEP}]}$ for the sake of convenience.

To generate the feature representation capturing the global contextual information of the entire question, we use an Attention Pooling Layer to get the feature representation $\mathbf{h}_Q$ of question Q. Concretely, $\mathbf{h}_Q$ is calculated as a weighted sum of all token annotations as:

$$\mathbf{h}_Q = \sum_{i=1}^{D+1} \alpha_i \mathbf{h}_{q_i}, \tag{10}$$

where α_i is the weight score of $\mathbf{h}_{q_i}$ computed by Equation 11.

$$\alpha_i = \mathrm{softmax}(\mathbf{v}_\alpha^\top \tanh(\mathbf{W}_\alpha \mathbf{h}_{[\mathrm{CLS}]} + \mathbf{U}_\alpha \mathbf{h}_{q_i})), \tag{11}$$

where $\mathbf{W}_\alpha$, $\mathbf{U}_\alpha$, and $\mathbf{v}_\alpha$ are trainable matrices and vectors. This mechanism guides the model to pay more attention to important words. Here, the context vector $\mathbf{h}_{[\mathrm{CLS}]}$ can be regarded as a highly abstract query, such as "what is the meaning of this question" [9].

2.2 Table Encoder.

Similar to the question encoder, the table encoder is also based on BERT, and the distinction lies in the organization of input. Let $(t_1, t_2, \cdots, t_{K_t})$ denote the token list of a table name where K_t is the number of words. The list of column names of this table is denoted by $\mathbf{C} = (\mathbf{c}^1, \mathbf{c}^2, \cdots, \mathbf{c}^S)$, where $\mathbf{c}^i$ is the token list of column name c_i as $\mathbf{c}^i = (c_1^i, c_2^i, \cdots, c_{K_i}^i)$, we flatten the token sequence as follows:

$$\mathbf{c} = (c_1^1, c_2^1, \cdots, c_{K_1}^1, c_1^2, c_2^2, \cdots, c_{K_2}^2, \cdots, c_1^S, c_2^S, \cdots, c_{K_S}^S).$$

Then the input of BERT is the concatenation of the table name and column names as:

$$[\mathrm{CLS}], t_1, \cdots, t_{K_t}, [\mathrm{SEP}], c_1^1, \cdots, [\mathrm{SEP}], c_1^S, \cdots, c_{K_S}^S, [\mathrm{SEP}].$$

Here, the symbol [SEP] is used to separate each column. The output of BERT is represented as:

$$\mathbf{h}_{[\mathrm{CLS}]}, \mathbf{h}_{t_1}, \cdots, \mathbf{h}_{t_{K_t}} \mathbf{h}_{[\mathrm{SEP}]}, \mathbf{h}_{c_1^1}, \cdots, \mathbf{h}_{[\mathrm{SEP}]}, \mathbf{h}_{c_1^S}, \cdots, \mathbf{h}_{[\mathrm{SEP}]}.$$

Both question encoder and table encoder share the same BERT parameters. We use different type embeddings to distinguish natural language queries and tables. The rest computation is similar to the question encoder, where the table representation $\mathbf{h}_T$ is calculated with the self-attention pooling mechanism (Equation 10).

The retrieval learning component receives the two high-level feature representations $\mathbf{h}_Q$ and $\mathbf{h}_T$ as input and computes the retrieval loss. Primarily, both feature vectors are fed to a shared feed-forward layer with tanh activation to generate the final embeddings, denoted by $\mathbf{E}_Q$ and $\mathbf{E}_T$.

2.3 Model Training.

The training process is formulated as a supervised representation learning task. The training object is constructed by hierarchically considering the coarse-grained distinctions over connected schema graphs and fine-grained semantics between tables in the same connected schema graph. Concretely, the model parameters are optimized by minimizing the following function:

$$\mathcal{L}_r = \mathcal{L}_r^c + \mathcal{L}_r^f. \tag{12}$$

where $\mathcal{L}_r^c$ (or $\mathcal{L}_r^f$) refers to the coarse-grained (or find-grained) loss.

$\mathcal{L}_r^c$ is developed to learn item representations ($\mathbf{E}_Q$ and $\mathbf{E}_T$) by constraining the latent space to gather matching query-table pairs, and discriminate irrelevant ones. It could be solved by the usage of triplet loss [5]. A triplet consists of an anchor $\mathbf{E}^a$, its matching counterpart $\mathbf{E}^p$ and a negative instance $\mathbf{E}^n$. The basic triplet loss is defined as:

$$\mathcal{L}_{tri}(\mathbf{E}^a, \mathbf{E}^p, \mathbf{E}^n) = \mathbf{ReLU}(s(\mathbf{E}^a, \mathbf{E}^n) - s(\mathbf{E}^a, \mathbf{E}^p) + m_c), \quad (13)$$

where the similarity score $s(x_1, x_2)$ is measured by cosine similarity, superscripts a, p, and n refer to anchor, positive, and negative samples respectively, and m_c is a predefined margin. Here, the anchor and positive items refer to the question embedding $\mathbf{E}_Q$ and the representation of a matched table $\mathbf{E}_T$. A table sampled from remaining connected schema graphs is used as the negative instance $\mathbf{E}_{T^n}$. Given the fact that triplet loss is hard to train and suffers from the gradient vanishing problem [3], we leverage hard-mining to accelerate convergence. To be specific, the loss is defined on a batch of matching question-table pairs $\mathbf{B} = \{(\mathbf{E}_{Q,1}, \mathbf{E}_{T,1}), \cdots, (\mathbf{E}_{Q,|\mathbf{B}|}, \mathbf{E}_{T,|\mathbf{B}|})\}$ and any two tables in the batch are not from the same connected schema graph. The training object is defined as the average of all non-zero losses as:

$$\mathcal{L}_r^c = \frac{1}{N_c} \sum_{i=1}^{|\mathbf{B}|} \sum_{j=1, j\neq i}^{|\mathbf{B}|} \mathcal{L}_{tri}(\mathbf{E}_{Q,i}, \mathbf{E}_{T,i}, \mathbf{E}_{T,j}), \quad (14)$$

where N_c is the number of non-zero triplets.

The fine-grained loss $\mathcal{L}_r^f$ is designed to distinguish tables in the same connected schema graph. Given a query Q^* and a connected schema graph, the tables are divided into two groups. $\mathcal{T}_p$ denotes the set of referenced tables and $\mathcal{T}_n$ represents the set of rest tables. The triplet of $\mathcal{L}_r^f$ is constructed as:

$$\mathcal{L}_{tri}^f(Q^*) = \sum_{T_p^* \in \mathcal{T}_p} \sum_{T_n^* \in \mathcal{T}_n} [s(\mathbf{E}_{Q^*}, \mathbf{E}_{T_n^*}) - s(\mathbf{E}_{Q^*}, \mathbf{E}_{T_p^*}) + m]. \quad (15)$$

The fine-grained loss $\mathcal{L}_r^f$ is calculated on another sampled batch $\mathbf{B}^*$ which is independent to $\mathbf{B}$ as:

$$\mathcal{L}_r^f = \frac{1}{N_f} \sum_{k=1}^{|\mathbf{B}^*|} \mathcal{L}_{tri}^f(Q_k^*) \quad (16)$$

where N_f is the total number of non-zero triplets in $\mathcal{L}_r^f$.

REFERENCES

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

[2] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *ACL*.

[3] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[4] Mark F. Medress, Franklin S Cooper, Jim W. Forgie, CC Green, Dennis H. Klatt, Michael H. O'Malley, Edward P Neuburg, Allen Newell, DR Reddy, B Ritea, et al. 1977. Speech understanding systems: Report of a steering committee. *Artificial Intelligence* 9, 3 (1977), 307–316.

[5] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *CVPR*.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*.

[7] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS*.

[8] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*.

[9] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[10] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Caiming Xiong, et al. 2021. GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing. In *International Conference on Learning Representations*.