RECPARSER: A Recursive Semantic Parsing Framework for Text-to-SQL Task

Yu Zeng\textsuperscript{1*}, Yan Gao\textsuperscript{2}, Jiaqi Guo\textsuperscript{3*}, Bei Chen\textsuperscript{2}, Qian Liu\textsuperscript{4*}, Jian-Guang Lou\textsuperscript{2}, Fei Teng\textsuperscript{11}, Dongmei Zhang\textsuperscript{2}

\textsuperscript{1}Southwest Jiaotong University, Chengdu, China
\textsuperscript{2}Microsoft Research Asia, Beijing, China
\textsuperscript{3}Xi’an Jiaotong University, Xi’an, China
\textsuperscript{4}Beihang University, Beijing, China

\textsuperscript{1}zengyutk@my.swjtu.edu.cn, 2\{Yan.Gao, beichen, jlou, dongmeiz\}@microsoft.com, 3jasperguo2013@stu.xjtu.edu.cn, 4qian.liu@buaa.edu.cn, 1fteng@swjtu.edu.cn

Abstract

Neural semantic parsers usually fail to parse long and complicated utterances into nested SQL queries, due to the large search space. In this paper, we propose a novel recursive semantic parsing framework called RECPARSER to generate the nested SQL query layer-by-layer. It decomposes the complicated nested SQL query generation problem into several progressive non-nested SQL query generation problems. Furthermore, we propose a novel Question Decomposer module to explicitly encourage RECPARSER to focus on different components of utterance when predicting SQL queries in different layers. Experiments on Spider dataset show that our approach is more effective compared to the previous works at predicting the nested SQL queries. In addition, we obtain an overall accuracy that is comparable with the state-of-the-art approaches.

1 Introduction

Text-to-SQL is one of the most important sub-tasks of semantic parsing in natural language processing (NLP). It maps natural language utterances to corresponding SQL queries. By helping non-experts to interact with ever increasing databases, the task has many important potential applications in real life, and thus receives a great deal of interest from both industry and academia [Li and Jagadish, 2016; Zhong et al., 2017; Affolter et al., 2019].

Composing nested SQL queries is a challenging problem in Text-to-SQL task. Several works [Guo et al., 2019; Zhang et al., 2019b; Bogin et al., 2019; Yu et al., 2018a] have attempted to deal with this problem on the recently proposed dataset Spider [Yu et al., 2018b], which contains nested SQL queries over different databases with multiple tables. However, due to the large search space, existing neural semantic parsers do not perform well on composing nested SQL queries [Zhang et al., 2019a; Affolter et al., 2019].

\textsuperscript{*}Work done during an internship at Microsoft Research.
\textsuperscript{†}Contact Author

Utterance: list all song names by singers age above the average singer age.

SQL Query: SELECT song_name FROM singers 
WHERE age > (SELECT avg(age) FROM singers)

Figure 1: An example of the component mapping relationship between the nested SQL query and its corresponding utterance.

In this paper, we focus on the problem of composing nested SQL queries in Text-to-SQL task. Firstly, as we know the nested SQL query representation is formalized as a kind of recursive structure, thus a nested SQL query generation problem could be decomposed into several progressive non-nested SQL query generation problems by nature. Furthermore, we found that there is a strong component mapping relationship between the nested SQL query and its corresponding utterance. Figure 1 shows an example of the component mapping relationship between the nested SQL query and its corresponding utterance. As we can see, for this nested SQL query, the outside layer “SELECT song_name FROM singers WHERE age >” and the inside layer “SELECT avg(age) FROM singers” correspond to different utterance components “list all song names by singers age above” and “the average singer age” respectively.

Inspired by these observations, we propose a recursive semantic parsing framework called RECPARSER to generate the nested SQL query layer-by-layer from outside to inside. Furthermore, we propose a novel Question Decomposer module to explicitly encourage RECPARSER to focus on different parts of utterance when predicting SQL queries in different layers. RECPARSER consists of four modules: Initial Encoder, Iterative Encoder, Question Decomposer and SQL Generator. In the initialization stage, we encode the user utterance and the database schema with the Initial Encoder. Then, by recursively calling the Question Decomposer, the Iterative Encoder and the SQL Generator modules, we generate the nested SQL query layer-by-layer. Finally, we compose them into the whole SQL query when encountering a recursive termination condition. Concretely, in each recursion round, the Question Decomposer updates the utterance representation with a soft-attention-mask mechanism and a recursive-attention-divergency loss proposed by us for next
recursion; The Iterative Encoder contextualizes the schema representation with the updated utterance representation; The SQL Generator takes the contextualized schema representation as input and generates the non-nested SQL query of the current layer. Our SQL Generator is a simple yet effective multi-task classification model.

We argue that RECPARSER has two advantages. Firstly, in this divide-and-conquer faction, we simplify the difficulty of the nested query generation problem. Our SQL Generator only learns to generate non-nested SQL queries instead of the whole nested SQL query, thus largely reducing the search space and alleviating the training difficulties. Secondly, through updating the utterance representation recursively with the soft-attention-mask mechanism and the recursive-attention-divergency loss, RECPARSER could focus on different parts of utterance when predicting different SQL query layers, thus minimizing the interference of irrelevant parts.

On Spider benchmark [Yu et al., 2018b], RECPARSER achieves a state-of-the-art 39.5% accuracy on the nested SQL query and a comparable 54.3% accuracy on the overall SQL query. When augmented with BERT [Devlin et al., 2018], RECPARSER reaches up to a state-of-the-art 46.8% accuracy on the nested SQL query and a state-of-the-art 63.1% accuracy on the overall SQL query. Our contributions are summarized as follows.

- We propose a recursive semantic parsing framework called RECPARSER to decompose a complicated nested SQL query generation problem into several progressive non-nested SQL query generation problems.
- We propose a novel Question Decomposer module with a soft-attention-mask mechanism and a recursive-attention-divergency loss to explicitly encourage RECPARSER to focus on different parts of the utterance when predicting SQL queries in different layers.
- Our approach is more effective compared to previous works at predicting nested SQL queries. In addition, we obtain an overall accuracy that is comparable with state-of-the-art approaches.

2 Related Works

Composing Nested SQL Query. The problem of composing nested SQL queries has been studied for decades [Androutsopoulos et al., 1995]. Most of the early proposed systems are rule-based methods [Popescu et al., 2003; Li and Jagadish, 2014]. Recently, with the development of advanced neural approaches and the release of the complex and cross-domain Text-to-SQL dataset Spider [Yu et al., 2018b], several neural semantic parsers [Guo et al., 2019; Zhang et al., 2019b; Bogin et al., 2019; Lee, 2019] have attempted to generate the nested SQL queries with a complicated grammar-based decoder. However applying grammar-based decoders to general programming languages such as SQL query is very challenging [Lin et al., 2019] and difficult to generalize to other semantic parsing tasks. Also, Finegan-Dollak et al. [2018] shows that the sequence-to-tree approach was inefficient when generating complicated SQL queries from an utterance. Our work differs from them in our use of a divide-and-conquer generation procedure.

Recursive Mechanism in Semantic Parsing. Recursive mechanism has been successfully used in complicated semantic parsing tasks [Andreas et al., 2016; Rabinovich et al., 2017]. Recently, several works [Yu et al., 2018a; Lee, 2019] also employ the recursive mechanism to generate nested SQL queries in Text-to-SQL task. SyntaxSQL-Net [Yu et al., 2018a] employs a SQL specific syntax tree-based decoder that calls a collection of recursive modules for decoding. RCSQL [Lee, 2019] proposes a SQL clause-wise decoding architecture. It recursively calls different SQL clause decoders to predict nested SQL queries. Compared with their approaches, we recursively call both encoder module and SQL generator module for each recursion, while they only call a set of SQL generator modules. In addition, we propose a novel Question Decomposer module to capture the mapping relationship between different SQL query layers and their corresponding components in utterance.

Decomposing Complicated Question. Decomposing complicated question is widely used in many semantic parsing works [Iyyer et al., 2016; Talmor and Berant, 2018; Zhang et al., 2019a]. They utilize the decompositionality of complicated questions to help question understanding. Inspired by them, we use the soft-attention-mask mechanism and the recursive-attention-divergency loss to model the decomposing process of the utterance and update the utterance representation for next recursion round. In this way, we don’t need to make hard decisions and it could be learned in an end-to-end fashion without extra labeling work.

Multi-Task Learning in Text-to-SQL Task. In multi-task learning, the Text-to-SQL task is decomposed into several sub-tasks, each predicting a part of the final SQL program. Compared with sequence-to-sequence-style models, multi-task learning does not require the SQL queries to be serialized and thus avoid the “ordering issue” [Xu et al., 2018].

Existing state-of-the-art multi-task learning methods [Hwang et al., 2019; He et al., 2019] have already surpassed human performance on WikiSQL [Zhong et al., 2017] dataset. However, existing methods are limited to the specific SQL sketch of WikiSQL, which only supports very simple queries. In this work, we propose a simple yet effective multi-task classification model to generate arbitrary non-nested SQL query as the SQL Generator module in our framework.

3 Methodology

In this section, we will describe our RECPARSER framework in detail. As described in Figure 2, RECPARSER consists of four modules: Initial Encoder, Iterative Encoder, Question Decomposer and SQL Generator. At first, we get the initial representation of utterance and database schema with the Initial Encoder. Then, by recursively calling the Iterative Encoder, the Question Decomposer, and the SQL Generator, we generate the nested SQL query layer-by-layer from outside to inside. Finally, we compose them into the whole SQL query when encountering a recursive termination condition.
3.1 Initial Encoder

Given a user input utterance $U$ and its corresponding database schema $S$, the Initial Encoder generates the initial representation $h_{U}^{\text{init}}$ and $h_{S}^{\text{init}}$. Let $U = (u_{1},\ldots,u_{l})$ and $S = \{t_{1},c_{11},\ldots,t_{j},c_{ij},\ldots\}$, where $u_{k}$ denotes the $k$-th utterance token, $l$ is the length of utterance, $t_{i}$ denotes the $i$-th table name in database schema, and $c_{ij}$ denotes the $j$-th column name in the $i$-th table. Following Hwang et al. [2019], we concatenate the utterance $U$ and the database schema $S$ as a single sequence separated by token $[CLS],[TAB], [COL]$, and $[SEP]$ as follows.

$$[CLS],[TAB],t_{1},[COL],c_{11},[COL],c_{1j},[SEP],u_{1},\ldots,u_{l}, (1)$$

where we put $[CLS]$ token at the beginning of the sequence to capture the contextualized representation of the whole sequence. Then we employ a Bidirection LSTM [Hochreiter and Schmidhuber, 1997] to get the $h_{U}^{\text{init}}$ and the $h_{S}^{\text{init}}$, where $h_{U}^{\text{init}} = (h_{u_{1}},\ldots,h_{u_{l}})$ and $h_{S}^{\text{init}} = (h_{t_{1}},\ldots,h_{t_{j}},h_{c_{1j}},\ldots)$. Concretely, for each token and column in database schema, we use the representation of the separator [TAB] and [COL] before each table and column as their corresponding representations. Moreover, we use $h_{CLS}$ denotes the representation of token $[CLS]$. By concatenating the utterance and the database schema together, we could capture the relationship between the utterance and the corresponding database schema and get the contextualized representations.

3.2 Iterative Encoder

Let $n$ denote the $n$-th recursion round. Given $h_{U}^{(n)}, h_{CLS}^{(n)}$ and $h_{S}^{(n)}$ as inputs, the goal of the Iterative Encoder is to generate the database schema representation $h_{S}^{(n)}$ and the $[CLS]$ representation $h_{CLS}^{(n)}$ of the $n$-th recursion round. Then, $h_{U}^{(n)}$ and $h_{CLS}^{(n)}$ will be used as the inputs of the SQL Generator to generate the current layer SQL query. Note that if it is the 0-th recursion round, we use $h_{U}^{\text{init}}$ as $h_{U}^{(0)}$. Specifically, following Hwang et al. [2019], we concatenate $h_{CLS}^{(n)}, h_{S}^{(n)}$ and $h_{U}^{(n)}$ in a single representation $h^{(n)}$. Then we apply self-attention mechanism [Vaswani et al., 2017] to update $h^{(n)}$ as follows.

$$\alpha^{(n)} = \text{softmax} \left( \frac{QK^{T}}{\sqrt{d}} \right) \tag{2}$$

$$h^{(n)} = V\alpha^{(n)};$$

where $Q = w_{q}h^{(n)},K = w_{k}h^{(n)}, V = w_{v}h^{(n)}$ and $w_{q}, w_{k}, w_{v} \in \mathbb{R}^{d}, w_{k} \in \mathbb{R}^{d}, w_{v} \in \mathbb{R}^{d}$ are trainable parameters.

3.3 SQL Generator

Given the database schema representation $h_{S}^{(n)}$ and the $[CLS]$ representation $h_{CLS}^{(n)}$, the goal of the SQL Generator is to generate the non-nested SQL query at the current recursion round $n$. According to the syntactical grammar of the SQL query, we transform the non-nested SQL query generation problem to a slot-filling problem. Concretely, we design a local template that indicates the property of each column in each SQL clause (e.g., SELECT clause) and a global template that indicates the property of the SQL query.

Firstly, we generate $N \times M$ local templates where $N$ is the number of columns in the given database schema $S$ and $M$ is the number of SQL clauses. Each template has three local slots, i.e., Aggregator, Operator and Nest. Notice that the slot Nest is used to indicate whether the column in corresponding clause has nested SQL query or not. Figure 3 presents all candidates of the Aggregator, the Operator and the Nest. Notice that EMPTY means that the column is not in the corresponding SQL clause. As shown in Figure 3, they are predicted by three classifiers, i.e., AGG-CLS, OP-CLS and NES-CLS respectively. Taking the OP-CLS as an example, the probability of the operator $op$ for col$_{ij}$ in clause$_{m}$ is computed as following:

$$p_{op}(\text{col}_{ij},m) = \text{softmax}(W_{op}h_{\text{col}_{ij}}^{(n)} + b_{op});$$

where $e_{\text{clause}_{m}}$ is the $m$-th clause embedding. $W_{op} \in \mathbb{R}^{n_{op} \times d}, b_{op} \in \mathbb{R}^{n_{op}}, W_{\text{clause}_{m}} \in \mathbb{R}^{d}$ are trainable parameters.
and \( n_{\text{op}} \) is the number of operator types. The same way can be also applied to the AGG-CLS and NES-CLS to calculate \( P_{\text{agg}}^{(\text{col}, i, m)} \) and \( P_{\text{nes}}^{(\text{col}, i, m)} \). Specially, we have 7 kinds of SQL clauses (e.g., SELECT, WHERE_AND, WHERE_OR, ORDER_ASC, ORDER_DESC, GROUP, and HAVING), in which the original WHERE clause is divided into \( \text{where} \) and \( \text{or} \), to distinguish two different connectors (i.e., AND and OR) in the WHERE clause. Similarly, the ORDER clause is also divided into \( \text{order}_{\text{asc}} \) and \( \text{order}_{\text{desc}} \) to distinguish two different sort types (i.e., DES and ASC).

Next, we generate one global template which has two global slots, i.e., Limit and IUE. Limit indicates if the SQL query contains keyword LIMIT. IUE indicates whether the SQL query contains structures such as Intersect, Union, Except or not. We use another two classifiers, i.e., LIMIT-CLS and IUE-CLS to predict them respectively. In IUE-CLS, the probability of the candidate \( \text{iue} \) is calculated as follows.

\[
P_{\text{iue}} = \text{softmax}(W_{\text{iue}} h_{\text{CLS}}^{(n)} + b_{\text{iue}}),
\]

where \( W_{\text{iue}} \in \mathbb{R}^{n_{\text{iue}} \times d}, b_{\text{iue}} \in \mathbb{R}^{n_{\text{iue}}} \) are trainable parameters, \( n_{\text{iue}} \) is the number of IUE types. \( P_{\text{limit}} \) is calculated in the same way by LIMIT-CLS. After filling all the slots of all the templates, we use a heuristic method to compose these templates to a SQL query according to the SQL grammar.

At last, after predicting all the other clauses, similar to [Lee, 2019], we use a heuristics to generate the FROM clause of the current SQL query layer. We first collect all the columns that appear in the predicted SQL, and then we JOIN tables that include these predicted columns.

### 3.4 Question Decomposer

The goal of the Question Decomposer is to update the utterance representation for next recursion round. Intuitively, we hope that RECPARSER could learn to decompose the utterance layer-by-layer and focus on different components of utterance when predicting SQL queries in different layers. We achieve this target by two simple yet effective methods: (1) **Soft Attention Mask**. Inspired by Kim et al. [2018] which leverages a soft attention mask mechanism to do feature selection, we propose a novel method to soft mask the parts of utterance that has already been focused by previous recursion rounds. (2) **Recursive Attention Divergence Loss**. Inspired by Wei et al. [2019] which use JS divergence loss to minimize the attention divergence in code generation task, we proposed a novel regularization loss to encourage the model to minimize the attention divergence between the adjacent recursion rounds. In this way, we explicitly encourage RECPARSER to focus on different components of utterance when predicting different SQL query layers, thus minimizing the interference of irrelevant components of the utterance.

**Soft Attention Mask**

Let \( \alpha_{U}^{(n)} \in \mathbb{R}^{l \times 1} \) denote the attention weights of token [CLS] on utterance \( U \), and \( l \) is the length of \( U \), which is produced by Equation 2. Intuitively, to help the model focus on different components of utterance in the next recursion, we want to mask the component that has been focused in the current recursion. Instead of directly masking the token of the utterance, we generate the utterance representation of next recursion by a soft-attention-mask method as follows.

\[
P_{\text{mask}}^{(n)} = \alpha_{U} W_{\text{mask}}^{(n)} + W_{m} E_{\text{mask}}^{(n)} - \lambda_{\text{div}} \alpha_{U} W_{\text{mask}}^{(n)} - \lambda_{\text{div}} W_{m} E_{\text{mask}}^{(n)},
\]

where \( W_{\text{mask}} \in \mathbb{R}^{l \times d} \) is the soft mask embedding, \( l \) is the length of utterance and \( W_{m} \in \mathbb{R}^{d} \), \( W_{m} \in \mathbb{R}^{d} \) are trainable parameters.

**Recursive Attention Divergence Loss**

To encourage the model to focus on different components of utterance in each recursion round, we design the Recursive Attention Divergence Loss \( L_{\text{div}} \) to regularize the attention divergence between the attention weights of the adjacent recursion rounds.

\[
L_{\text{div}} = -\frac{1}{2}(D_{\text{KL}}(P||Q) + D_{\text{KL}}(Q||P)) + \lambda_{\text{div}} \alpha_{U} W_{\text{mask}}^{(n)} + \lambda_{\text{div}} W_{m} E_{\text{mask}}^{(n)} - \lambda_{\text{div}} \alpha_{U} W_{\text{mask}}^{(n)} - \lambda_{\text{div}} W_{m} E_{\text{mask}}^{(n)},
\]

where \( D_{\text{KL}} \) is the Kullback–Leibler divergence, defined as

\[
D_{\text{KL}}(p||q) = \sum_{x} p(x) \log \left( \frac{p(x)}{q(x)} \right)
\]

which measures how one probability distribution \( q \) diverges from another probability distribution \( p \).

### 3.5 Composing the Final SQL Query

The final SQL query is obtained through backtracking. The SQL Generator will return a SQL query segment after the end of the recursive calling. The recursive termination condition is that the IUE slot and all NEST slots are NO. Specifically, if the slot NEST is YES, we return a SQL query with a placeholder token [SUBQUERY] after its corresponding column that is in its clause (e.g., SELECT name FROM singers WHERE age > [SUBQUERY] and sex = ‘male’). Similarly, if the IUE slot is not NO, we put corresponding placeholder token after the returned SQL query result (e.g., SELECT name FROM singers [EXCEPT]). Finally, we construct the final SQL by replacing the placeholder token with the returned SQL query segment.

### 3.6 Loss function

The loss function of RECPARSER is defined as follows.

\[
L = \lambda_{\text{agg}} L_{\text{agg}} + \lambda_{\text{op}} L_{\text{op}} + \lambda_{\text{nes}} L_{\text{nes}} + \lambda_{\text{iue}} L_{\text{iue}} + \lambda_{\text{limit}} L_{\text{limit}} + \lambda_{\text{div}} L_{\text{div}},
\]

where \( L_{\text{agg}}, L_{\text{op}}, L_{\text{nes}}, L_{\text{iue}}, L_{\text{limit}}, L_{\text{div}} \) are the loss function of AGG-CLS, OP-CLS, NES-CLS, IUE-CLS and LIMIT-CLS in the SQL Generator respectively. All loss function of the classifiers are cross-entropy loss. \( L_{\text{div}} \) is the Recursive Attention Divergence Loss described in Equation 6. \( \lambda_{\text{agg}}, \lambda_{\text{op}}, \lambda_{\text{nes}}, \lambda_{\text{iue}}, \lambda_{\text{limit}}, \lambda_{\text{div}} \) are loss weights and all of them are hyper parameters.
4 Experiments

In this section, we evaluate the effectiveness of RECPARSER on both nested SQL query and all SQL query by comparing it to state-of-the-art approaches and ablating several design choices in RECPARSER to understand their contributions.

4.1 Experiment Setup

Dataset. We conduct our experiments on Spider [Yu et al., 2018b]. It is a large-scale, human annotated and cross-domain Text-to-SQL benchmark, which contains 7,000/1,034 question-SQL query pairs in train and development set. Especially, there are 503/79 question and nested-SQL query pairs in train and development set1. We evaluate RECPARSER and other approaches using SQL Exact Matching and Component Matching accuracy metrics proposed by Yu et al. [2018b].

Baselines. We use the following methods for comparison evaluation: SyntaxSQLNet [Yu et al., 2018a], RCSQL [Lee, 2019], Global GNN [Bogin et al., 2019], EditSQL [Zhang et al., 2019b], and IRNET [Guo et al., 2019]. All of them employ an encoder-decoder architecture, but with different design choices: SyntaxSQLNet is a sequence-to-set model with a SQL specific syntax tree-based decoder; RCSQL employs a SQL clause-wise decoding network with recursive mechanism; Global GNN reasons over the DB structure and questions to make global decision; EditSQL proposes an editing mechanism-based network; IRNET uses schema linking information and predicts an intermediate representation instead of the SQL query. Note that SyntaxSQLNet, RCSQL, EditSQL and IRNET do not use the database content, while Global GNN and RECPARSER use the database content.

Implementations. We implement RECPARSER and the baseline approaches with PyTorch. The dropout rate is 0.2. We use Adam with 1e-3 learning rate for optimization. Batch size is set to 64. Word embeddings are initialized with Glove. The dimensions of word embedding, type embedding and hidden vectors are set to 300. λ_{AGG}, λ_{OP}, λ_{NES}, λ_{IUE}, λ_{LIMIT}, λ_{DIV} in Equation 7 are set as 1, 1, 1, 0.1, 0.1, and 0.2 respectively. We use the database content to find the column whose value is exactly mentioned in the utterance and put a special token [VALUE] in front of that column.

1 We don’t use the test set of Spider, since the test set is blind to us and we can’t obtain the nested SQL query of the test set to evaluate the effectiveness of our approach.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Accuracy</th>
<th>Accuracy (BERT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyntaxSQLNet</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Global-GNN</td>
<td>33.0%</td>
<td>-</td>
</tr>
<tr>
<td>EditSQL</td>
<td>-</td>
<td>40.4%</td>
</tr>
<tr>
<td>IRNET</td>
<td>34.2%</td>
<td>41.8%</td>
</tr>
<tr>
<td>RECPARSER(no DB)</td>
<td>38.0%</td>
<td>46.8%</td>
</tr>
<tr>
<td>RECPARSER</td>
<td>39.3%</td>
<td>46.8%</td>
</tr>
</tbody>
</table>

Table 1: Nested SQL Exact Matching accuracy on Spider development set. “no DB” means the database content is not used; “BERT” means the results are obtained with BERT enhanced approaches and “-” means the methods are not proposed in corresponding settings, e.g., with or without BERT.

4.2 Model Comparison

Table 1 presents the nested SQL Exact Matching accuracy of RECPARSER and various baselines on the development set. The result shows that RECPARSER clearly outperforms all the baselines with or without DB content used on the nested SQL query accuracy. Comparing to the methods that use various techniques to predict nested SQL query, such as complicated intermediate representation (e.g., IRNet) and other grammar-based decoders (e.g., EditSQL and Global-GNN), RECPARSER achieves a prominent improvement by employing a novel recursive generation framework. For example, it obtains 5.0% absolute improvement than the state-of-art method IRNET when both of them incorporate with BERT.

Next, we evaluate the overall exact matching accuracy of RECPARSER and various baselines on the development set of Spider. As shown in Table 2 RECPARSER(BERT, no DB) achieves a 60.5% exact matching accuracy without using any rule-based schema linking techniques or intermediate representation that are used in IRNet(BERT). It demonstrates that RECPARSER(BERT) could obtain a comparable overall accuracy than the state-of-the-art approaches at the same setting with minimum efforts. When incorporating the database con-
Table 3: Ablation study results. Base-framework doesn’t use the Question Decomposer (QD) and the database content (DB). ‘QD (-RAD loss)’ means that the QD does not use the recursive-attention-divergency loss.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Base-Framework</td>
<td>57.4%</td>
</tr>
<tr>
<td>+ QD (-RAD loss)</td>
<td>58.5%</td>
</tr>
<tr>
<td>+ QD</td>
<td>60.8%</td>
</tr>
<tr>
<td>+ QD + DB</td>
<td>63.1%</td>
</tr>
</tbody>
</table>

4.3 Ablation Study

We conduct ablation studies on RECPARSER(BERT) to analyze the contribution of each design choice. In detail, we first evaluate a base framework that does not use the Question Decomposer (QD) and the database content (DB). Then, we gradually apply each component on the base model. The ablation study is conducted on the development set. Table 3 shows the results of ablation study.

Firstly, it is clear that our base model significantly outperforms the well designed complicated architecture methods (e.g., Global-GNN and SyntaxSQLNet) by using the recursive framework with the simple yet effective multi-task SQL Generator module. Secondly, using the Question Decomposer without the recursive-attention-divergency loss improves the performance of RECPARSER about 1.1%. It demonstrates the effectiveness of using the soft-mask-attention to update the utterance representation. Thirdly, adding the recursive-attention-divergency loss further brings a 2.3% improvement. It demonstrates the effectiveness of our loss function. We observed that for those nested SQL queries, the base framework has the problem that tends to put the columns in the outside layer to the inside layer. The number of examples suffering from this problem decreases by 53% when using the Question Decomposer. It means that the Question Decomposer helps to minimize the interference between different SQL layers. At last, using the database content helps to recognize the value mentioned in the utterance, which brings a 2.3% improvement of performance.

Figure 5: Some examples where Question Decomposer (QD) for RECPARSER leads to correct prediction. In the first example, RECPARSER without QD selects the wrong column in the outside SQL layer which belong to the inner SQL layer. In the second example, RECPARSER without QD selects the wrong column in the inner SQL layer which belong the outside SQL layer.


[Yu et al., 2018b] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In EMNLP, 2018.

