Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation

Jiaqi Guo[1]*, Zecheng Zhan[2]*, Yan Gao[3], Yan Xiao[3], Jian-Guang Lou[3]
Ting Liu[1], Dongmei Zhang[3]
[1]Xi'an Jiaotong University, Xi'an, China
[2]Beijing University of Posts and Telecommunications, Beijing, China
[3]Microsoft Research Asia, Beijing, China
jasperguo2013@stu.xjtu.edu.cn, zhanzecheng@bupt.edu.cn
{Yan.Gao,Yan.Xiao,jlou,dongmeiz}@microsoft.com
tingliu@mail.xjtu.edu.cn

Abstract

We present a neural approach called IRNet for complex and cross-domain Text-to-SQL. IRNet aims to address two challenges: 1) the mismatch between intents expressed in natural language (NL) and the implementation details in SQL; 2) the challenge in predicting columns caused by the large number of out-of-domain words. Instead of end-to-end synthesizing a SQL query, IRNet decomposes the synthesis process into three phases. In the first phase, IRNet performs a schema linking over a question and a database schema. Then, IRNet adopts a grammar-based neural model to synthesize a SemQL query which is an intermediate representation that we design to bridge NL and SQL. Finally, IRNet deterministically infers a SQL query from the synthesized SemQL query with domain knowledge. On the challenging Text-to-SQL benchmark Spider, IRNet achieves 46.7% accuracy, obtaining 19.5% absolute improvement over previous state-of-the-art approaches. At the time of writing, IRNet achieves the first position on the Spider leaderboard.

NL: Show the names of students who have a grade higher than 5 and have at least 2 friends.

SQL:
```
SELECT T1.name
FROM friend AS T1 JOIN highschooler AS T2
ON T1.student_id = T2.id WHERE T2.grade > 5
GROUP BY T1.student_id HAVING count(*) >= 2
```

Figure 1: An example from the Spider benchmark to illustrate the mismatch between the intent expressed in NL and the implementation details in SQL. The column *'student_id'* to be grouped by in the SQL query is not mentioned in the question.

1 Introduction

Recent years have seen a great deal of renewed interest in Text-to-SQL, i.e., synthesizing a SQL query from a question. Advanced neural approaches synthesize SQL queries in an end-to-end manner and achieve more than 80% exact matching accuracy on public Text-to-SQL benchmarks (e.g., ATIS, GeoQuery and WikiSQL) (Krishnamurthy et al., 2017; Zhong et al., 2017; Xu et al., 2017; Yaghmazadeh et al., 2017; Yu et al., 2018a; Dong and Lapata, 2018; Wang et al., 2018; Hwang et al., 2019). However, Yu et al. (2018c) presents unsatisfactory performance of state-of-the-art approaches on a newly released, cross-domain Text-to-SQL benchmark, Spider.

The Spider benchmark brings new challenges that prove to be hard for existing approaches. Firstly, the SQL queries in the Spider contain nested queries and clauses like GROUPBY and HAVING, which are far more complicated than that in another well-studied cross-domain benchmark, WikiSQL (Zhong et al., 2017). Considering the example in Figure 1, the column *'student_id'* to be grouped by in the SQL query is never mentioned in the question. In fact, the GROUPBY clause is introduced in SQL to facilitate the implementation of aggregate functions. Such implementation details, however, are rarely considered by end users and therefore rarely mentioned in questions. This poses a severe challenge for existing end-to-end neural approaches to synthesize SQL queries in the absence of detailed specification. The challenge in essence stems from the fact that SQL is designed for effectively querying relational databases instead of for representing the meaning of NL (Kate, 2008). Hence, there inevitably exists a mismatch between intents expressed in natural language and the implementation details in SQL. We regard this challenge as a mismatch problem.

Secondly, given the cross-domain settings of Spider, there are a large number of out-of-domain

* Equal Contributions. Work done during an internship at MSRA.

(OOD) words. For example, 35% of words in database schemas on the development set do not occur in the schemas on the training set in Spider. As a comparison, the number in WikiSQL is only 22%. The large number of OOD words poses another steep challenge in predicting columns in SQL queries (Yu et al., 2018b), because the OOD words usually lack of accurate representations in neural models. We regard this challenge as a lexical problem.

In this work, we propose a neural approach, called IRNet, towards tackling the mismatch problem and the lexical problem with intermediate representation and schema linking. Specifically, instead of end-to-end synthesizing a SQL query from a question, IRNet decomposes the synthesis process into three phases. In the first phase, IRNet performs a schema linking over a question and a schema. The goal of the schema linking is to recognize the columns and the tables mentioned in a question, and to assign different types to the columns based on how they are mentioned in the question. Incorporating the schema linking can enhance the representations of question and schema, especially when the OOD words lack of accurate representations in neural models during testing. Then, IRNet adopts a grammar-based neural model to synthesize a SemQL query, which is an intermediate representation (IR) that we design to bridge NL and SQL. Finally, IRNet deterministically infers a SQL query from the synthesized SemQL query with domain knowledge.

The insight behind IRNet is primarily inspired by the success of using intermediate representations (e.g., lambda calculus (Carpenter, 1997), FunQL (Kate et al., 2005) and DCS (Liang et al., 2011)) in various semantic parsing tasks (Zelle and Mooney, 1996; Berant et al., 2013; Pasupat and Liang, 2015; Wang et al., 2017), and previous attempts in designing IR to decouple meaning representations of NL from database schema and database management system (Woods, 1986; Alshawi, 1992; Androutsopoulos et al., 1993).

$$
\begin{aligned}
Z &::= intersect\ R\ R \mid union\ R\ R \mid except\ R\ R \mid R \\
R &::= Select \mid Select\ Filter \mid Select\ Order \\
&\quad \mid Select\ Superlative \mid Select\ Order\ Filter \\
&\quad \mid Select\ Superlative\ Filter \\
Select &::= A \mid A\ A \mid A\ A\ A \mid A\ A\ A\ A \mid A\ A \cdots A \\
Order &::= asc\ A \mid desc\ A \\
Suerlative &::= most\ A \mid least\ A \\
Filter &::= and\ Filter\ Filter \mid or\ Filter\ Filter \\
&\quad \mid > A \mid > A\ R \mid < A \mid < A\ R \\
&\quad \mid \geq A \mid \geq A\ R \mid = A \mid = A\ R \\
&\quad \mid \neq A \mid \neq A\ R \mid between\ A \\
&\quad \mid like\ A \mid not\ like\ A \mid in\ A\ R \mid not\ in\ A\ R \\
A &::= \max C\ T \mid \min C\ T \mid count\ C\ T \\
&\quad \mid sum\ C\ T \mid avg\ C\ T \mid none\ C\ T \\
C &::= column \\
T &::= table
\end{aligned}
$$

Figure 2: The context-free grammar of SemQL. *column* ranges over distinct column names in a schema. *table* ranges over tables in a schema.

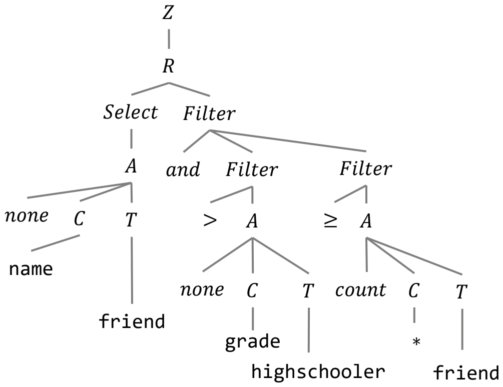


Figure 3: An illustrative example of SemQL. Its corresponding question and SQL query are shown in Figure 1.

On the challenging Spider benchmark (Yu et al., 2018c), IRNet achieves 46.7% exact matching accuracy, obtaining 19.5% absolute improvement over previous state-of-the-art approaches. At the time of writing, IRNet achieves the first position on the Spider leaderboard. When augmented with BERT (Devlin et al., 2018), IRNet reaches up to 54.7% accuracy. In addition, as we show in the experiments, learning to synthesize SemQL queries rather than SQL queries can substantially benefit other neural approaches for Text-to-SQL, such as SQLNet (Xu et al., 2017), TypeSQL (Yu et al., 2018a) and SyntaxSQLNet (Yu et al., 2018b). Such results on the one hand demonstrate the effectiveness of SemQL in bridging NL and SQL. On the other hand, it reveals that designing an effective intermediate representation to bridge NL and SQL is a promising direction to being there for complex and cross-domain Text-to-SQL.

2 Approach

In this section, we present IRNet in detail. We first describe how to tackle the mismatch problem and the lexical problem with intermediate representation and schema linking. Then we present the neural model to synthesize SemQL queries.

2.1 Intermediate Representation

To eliminate the mismatch, we design a domain specific language, called SemQL, which serves as an intermediate representation between NL and SQL. Figure 2 presents the context-free grammar of SemQL. An illustrative SemQL query is shown in Figure 3. We elaborate on the design of SemQL in the following.

Inspired by lambda DCS (Liang, 2013), SemQL is designed to be tree-structured. This structure, on the one hand, can effectively constrain the search space during synthesis. On the other hand, in view of the tree-structure nature of SQL (Yu et al., 2018b; Yin and Neubig, 2018), following the same structure also makes it easier to translate to SQL intuitively.

The mismatch problem is mainly caused by the implementation details in SQL queries and missing specification in questions as discussed in Section 1. Therefore, it is natural to hide the implementation details in the intermediate representation, which forms the basic idea of SemQL. Considering the example in Figure 3, the `GROUPBY`, `HAVING` and `FROM` clauses in the SQL query are eliminated in the SemQL query, and the conditions in `WHERE` and `HAVING` are uniformly expressed in the subtree of *Filter* in the SemQL query. The implementation details can be deterministically inferred from the SemQL query in the later inference phase with domain knowledge. For example, a column in the `GROUPBY` clause of a SQL query usually occurs in the `SELECT` clause or it is the primary key of a table where an aggregate function is applied to one of its columns.

In addition, we strictly require to declare the table that a column belongs to in SemQL. As illustrated in Figure 3, the column *'name'* along with its table *'friend'* are declared in the SemQL query. The declaration of tables helps to differentiate duplicated column names in the schema. We also declare a table for the special column '$\star$' because we observe that '$\star$' usually aligns with a table mentioned in a question. Considering the example in Figure 3, the column '$\star$' in essence aligns with the table *'friend'*, which is explicitly mentioned in the question. Declaring a table for '$\star$' also helps infer the `FROM` clause in the next inference phase.

When it comes to inferring a SQL query from a SemQL query, we perform the inference based on an assumption that the definition of a database schema is precise and complete. Specifically, if a column is a foreign key of another table, there should be a foreign key constraint declared in the schema. This assumption usually holds as it is the best practice in database design. More than 95% of examples in the training set of the Spider benchmark hold this assumption. The assumption forms the basis of the inference. Take the inference of the `FROM` clause in a SQL query as an example. We first identify the shortest path that connects all the declared tables in a SemQL query in the schema (A database schema can be formulated as an undirected graph, where vertex are tables and edges are foreign key relations among tables). Joining all the tables in the path eventually builds the `FROM` clause. Supplementary materials provide detailed procedures of the inference and more examples of SemQL queries.

2.2 Schema Linking

The goal of schema linking in IRNet is to recognize the columns and the tables mentioned in a question, and assign different types to the columns based on how they are mentioned in the question. Schema linking is an instantiation of entity linking in the context of Text-to-SQL, where entity is referred to columns, tables and cell values in a database. We use a simple yet effective string-match based method to implement the linking. In the followings, we illustrate how IRNet performs schema linking in details based on the assumption that the cell values in a database are not available.

As a whole, we define three types of entities that may be mentioned in a question, namely, *table*, *column* and *value*, where *value* stands for a cell value in the database. In order to recognize entities, we first enumerate all the n-grams of length 1-6 in a question. Then, we enumerate them in the descending order of length. If an n-gram exactly matches a column name or is a subset of a column name, we recognize this n-gram as a *column*. The recognition of *table* follows the same way. If an n-gram can be recognized as both *column* and *table*, we prioritize *column*. If an n-gram begins and ends with a single quote, we recognize it as *value*. Once an n-gram is recognized, we will remove other n-grams that overlap with it. To this end, we can recognize all the entities mentioned in a question and obtain a non-overlap n-gram sequence of the question by joining those recognized n-grams and the remaining 1-grams. We refer each n-gram in the sequence as a span and assign each

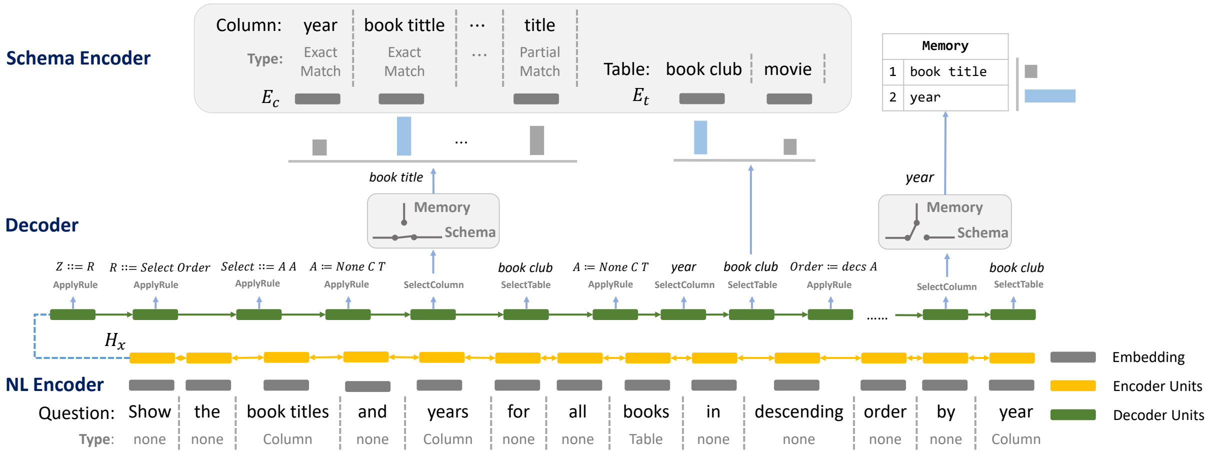


Figure 4: An overview of the neural model to synthesize SemQL queries. Basically, IRNet is constituted by an NL encoder, a schema encoder and a decoder. As shown in the figure, the column 'book title' is selected from the schema, while the second column 'year' is selected from the memory.

span a type according to its entity. For example, if a span is recognized as *column*, we will assign it a type COLUMN. Figure 4 depicts the schema linking results of a question.

For those spans recognized as *column*, if they exactly match the column names in the schema, we assign these columns a type EXACT MATCH, otherwise a type PARTIAL MATCH. To link the cell value with its corresponding column in the schema, we first query the *value* span in ConceptNet (Speer and Havasi, 2012) which is an open, large-scale knowledge graph and search the results returned by ConceptNet over the schema. We only consider the query results in two categories of ConceptNet, namely, 'is a type of' and 'related terms', as we observe that the column that a cell value belongs to usually occurs in these two categories. If there exists a result exactly or partially matches a column name in the schema, we assign the column a type VALUE EXACT MATCH or VALUE PARTIAL MATCH.

2.3 Model

We present the neural model to synthesize SemQL queries, which takes a question, a database schema and the schema linking results as input. Figure 4 depicts the overall architecture of the model via an illustrative example.

To address the lexical problem, we consider the schema linking results when constructing representations for the question and columns in the schema. In addition, we design a memory augmented pointer network for selecting columns during synthesis. When selecting a column, it makes a decision first on whether to select from memory or not, which sets it apart from the vanilla pointer network (Vinyals et al., 2015). The motivation behind the memory augmented pointer network is that the vanilla pointer network is prone to selecting same columns according to our observations.

NL Encoder. Let $x{=}[(x_1,\tau_1),\cdots,(x_L,\tau_L)]$ denote the non-overlap span sequence of a question, where x_i is the i^{th} span and τ_i is the type of span x_i assigned in schema linking. The NL encoder takes x as input and encodes x into a sequence of hidden states $\boldsymbol{H}_x$. Each word in x_i is converted into its embedding vector and its type τ_i is also converted into an embedding vector. Then, the NL encoder takes the average of the type and word embeddings as the span embedding $\boldsymbol{e}_x^i$. Finally, the NL encoder runs a bi-directional LSTM (Hochreiter and Schmidhuber, 1997) over all the span embeddings. The output hidden states of the forward and backward LSTM are concatenated to construct $\boldsymbol{H}_x$.

Schema Encoder. Let $s{=}(c,t)$ denote a database schema, where $c{=}\{(c_1,\phi_i),\cdots,(c_n,\phi_n)\}$ is the set of distinct columns and their types that we assign in schema linking, and $t{=}\{t_1,\cdots,t_m\}$ is the set of tables. The schema encoder takes s as input and outputs representations for columns $\boldsymbol{E}_c$ and tables $\boldsymbol{E}_t$. We take the column representations as an example below. The construction of table representations follows the same way except that we do not assign a type to a table in schema linking.

Concretely, each word in c_i is first converted into its embedding vector and its type ϕ_i is also converted into an embedding vector $\boldsymbol{\varphi}_i$. Then, the schema encoder takes the average of word embeddings as the initial representations $\hat{\boldsymbol{e}}_c^i$ for the column. The schema encoder further performs an attention over the span embeddings and obtains a context vector $\boldsymbol{c}_c^i$. Finally, the schema encoder takes the sum of the initial embedding, context vector and the type embedding as the column representation $\boldsymbol{e}_c^i$. The calculation of the representations for column c_i is as follows.

$$
\begin{aligned}
g_k^i &= \frac{(\hat{\boldsymbol{e}}_c^i)^\mathsf{T} \boldsymbol{e}_x^k}{\|\hat{\boldsymbol{e}}_c^i\| \|\boldsymbol{e}_x^k\|} \\
\boldsymbol{c}_c^i &= \sum_{k=1}^{L} g_k^i \boldsymbol{e}_x^k \\
\boldsymbol{e}_c^i &= \hat{\boldsymbol{e}}_c^i + \boldsymbol{c}_c^i + \boldsymbol{\varphi}_i,
\end{aligned}
$$

Decoder. The goal of the decoder is to synthesize SemQL queries. Given the tree structure of SemQL, we use a grammar-based decoder (Yin and Neubig, 2017, 2018) which leverages a LSTM to model the generation process of a SemQL query via sequential applications of actions. Formally, the generation process of a SemQL query y can be formalized as follows.

$$
p(y|x, s) = \prod_{i=1}^{T} p(a_i|x, s, a_{<i}),
$$

where a_i is an action taken at time step i, $a_{<i}$ is the sequence of actions before i, and T is the number of total time steps of the whole action sequence.

The decoder interacts with three types of actions to generate a SemQL query, including APPLYRULE, SELECTCOLUMN and SELECTTABLE. APPLYRULE(r) applies a production rule r to the current derivation tree of a SemQL query. SELECTCOLUMN(c) and SELECTTABLE(t) selects a column c and a table t from the schema, respectively. Here, we detail the action SELECTCOLUMN and SELECTTABLE. Interested readers can refer to Yin and Neubig (2017) for details of the action APPLYRULE.

We design a memory augmented pointer network to implement the action SELECTCOLUMN. The memory is used to record the selected columns, which is similar to the memory mechanism used in Liang et al. (2017). When the decoder is going to select a column, it first makes a decision on whether to select from the memory or not, and then selects a column from the memory or the schema based on the decision. Once a column is selected, it will be removed from the schema and be recorded in the memory. The probability of selecting a column c is calculated as follows.

$$
\begin{aligned}
p(a_i = \textsc{SelectColumn}[c]|x, s, a_{<i}) = \\
p(\textsc{Mem}|x, s, a_{<i}) p(c|x, s, a_{<i}, \textsc{Mem}) \\
+ p(\mathrm{S}|x, s, a_{<i}) p(c|x, s, a_{<i}, \mathrm{S})
\end{aligned}
$$

$$
\begin{aligned}
p(\textsc{Mem}|x, s, a_{<i}) &= \mathrm{sigmod}(\boldsymbol{w}_m^\mathsf{T} \boldsymbol{v}_i) \\
p(\mathrm{S}|x, s, a_{<i}) &= 1 - p(\textsc{Mem}|x, s, a_{<i}) \\
p(c|x, s, a_{<i}, \textsc{Mem}) &\propto \exp(\boldsymbol{v}_i^\mathsf{T} \boldsymbol{E}_c^m) \\
p(c|x, s, a_{<i}, \mathrm{S}) &\propto \exp(\boldsymbol{v}_i^\mathsf{T} \boldsymbol{E}_c^s),
\end{aligned}
$$

where S represents selecting from schema, MEM represents selecting from memory, $\boldsymbol{v}_i$ denotes the context vector that is obtained by performing an attention over $\boldsymbol{H}_x$, $\boldsymbol{E}_c^m$ denotes the embedding of columns in memory and $\boldsymbol{E}_c^s$ denotes the embedding of columns that are never selected. $\boldsymbol{w}_m$ is trainable parameter.

When it comes to SELECTTABLE, the decoder selects a table t from the schema via a pointer network:

$$
p(a_i = \textsc{SelectTable}[t]|x, s, a_{<i}) \propto \exp(\boldsymbol{v}_i^\mathsf{T} \boldsymbol{E}_t).
$$

As shown in Figure 4, the decoder first predicts a column and then predicts the table that it belongs to. To this end, we can leverage the relations between columns and tables to prune the irrelevant tables.

Coarse-to-fine. We further adopt a coarse-to-fine framework (Solar-Lezama, 2008; Bornholt et al., 2016; Dong and Lapata, 2018), decomposing the decoding process of a SemQL query into two stages. In the first stage, a skeleton decoder outputs a skeleton of the SemQL query. Then, a detail decoder fills in the missing details in the skeleton by selecting columns and tables. Supplementary materials provide a detailed description of the skeleton of a SemQL query and the coarse-to-fine framework.

3 Experiment

In this section, we evaluate the effectiveness of IRNet by comparing it to the state-of-the-art approaches and ablating several design choices in IRNet to understand their contributions.

3.1 Experiment Setup

Dataset. We conduct our experiments on the Spider (Yu et al., 2018c), a large-scale, human-annotated and cross-domain Text-to-SQL benchmark. Following Yu et al. (2018b), we use the database split for evaluations, where 206 databases are split into 146 training, 20 development and 40 testing. There are 8625, 1034, 2147 question-SQL query pairs for training, development and testing. Just like any competition benchmark, the test set of Spider is not publicly available, and our models are submitted to the data owner for testing. We evaluate IRNet and other approaches using SQL Exact Matching and Component Matching proposed by Yu et al. (2018c).

Baselines. We also evaluate the sequence-to-sequence model (Sutskever et al., 2014) augmented with a neural attention mechanism (Bahdanau et al., 2014) and a copying mechanism (Gu et al., 2016), SQLNet (Xu et al., 2017), TypeSQL (Yu et al., 2018a), and SyntaxSQLNet (Yu et al., 2018b) which is the state-of-the-art approach on the Spider.

Implementations. We implement IRNet and the baseline approaches with PyTorch (Paszke et al., 2017). Dimensions of word embeddings, type embeddings and hidden vectors are set to 300. Word embeddings are initialized with Glove (Pennington et al., 2014) and shared between the NL encoder and schema encoder. They are fixed during training. The dimension of action embedding and node type embedding are set to 128 and 64, respectively. The dropout rate is 0.3. We use Adam (Kingma and Ba, 2014) with default hyperparameters for optimization. Batch size is set to 64.

BERT. Language model pre-training has shown to be effective for learning universal language representations. To further study the effectiveness of our approach, inspired by SQLova (Hwang et al., 2019), we leverage BERT (Devlin et al., 2018) to encode questions, database schemas and the schema linking results. The decoder remains the same as in IRNet. Specifically, the sequence of spans in the question are concatenated with all the distinct column names in the schema. Each column name is separated with a special token *[SEP]*. BERT takes the concatenation as input. The representation of a span in the question is taken as the average hidden states of its words and type. To construct the representation of a column, we first run a bi-directional LSTM (BI-LSTM) over the hidden states of its words. Then, we take the sum of its type embedding and the final hidden state of the BI-LSTM as the column representation. The construction of table representations follows the same way. Supplementary material provides a figure to illustrate the architecture of the encoder. To establish baseline, we also augment SyntaxSQLNet with BERT. Note that we only use the base version of BERT due to the resource limitations.

Approach	Dev	Test
Seq2Seq	1.9%	3.7%
Seq2Seq + Attention	1.8%	4.8%
Seq2Seq + Copying	4.1%	5.3%
TypeSQL	8.0%	8.2%
SQLNet	10.9%	12.4%
SyntaxSQLNet	18.9%	19.7%
SyntaxSQLNet(augment)	24.8%	27.2%
IRNet	**53.2%**	**46.7%**
BERT		
SyntaxSQLNet(BERT)	25.0%	25.4%
IRNet(BERT)	**61.9%**	**54.7%**

Table 1: Exact matching accuracy on SQL queries.

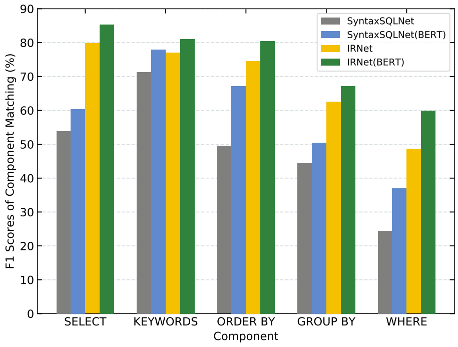


Figure 5: F1 scores of component matching of SyntaxSQLNet, SyntaxSQLNet(BERT), IRNet and IRNet(BERT) on test set.

We do not perform any data augmentation for fair comparison. All our code are publicly available. [1]

3.2 Experimental Results

Table 1 presents the exact matching accuracy of IRNet and various baselines on the development set and the test set. IRNet clearly outperforms all the baselines by a substantial margin. It obtains 27.0% absolute improvement over SyntaxSQLNet on test set. It also obtains 19.5% absolute improvement over SyntaxSQLNet(augment) that performs large-scale data augmentation. When incorporating BERT, the performance of both SyntaxSQLNet and IRNet is substantially improved and the accuracy gap between them on both the development set and the test set is widened.

To study the performance of IRNet in detail, following Yu et al. (2018b), we measure the average F1 score on different SQL components on the test

[1] https://github.com/zhanzecheng/IRNet

set. We compare between SyntaxSQLNet and IRNet. As shown in Figure 5, IRNet outperforms SyntaxSQLNet on all components. There are at least 18.2% absolute improvement on each component except KEYWORDS. When incorporating BERT, the performance of IRNet on each component is further boosted, especially in WHERE clause.

We further study the performance of IRNet on different portions of the test set according to the hardness levels of SQL defined in Yu et al. (2018c). As shown in Table 2, IRNet significantly outperforms SyntaxSQLNet in all four hardness levels with or without BERT. For example, compared with SyntaxSQLNet, IRNet obtains 23.3% absolute improvement in Hard level.

Approach	Easy	Medium	Hard	Extra Hard
SyntaxSQLNet	38.6%	17.6%	16.3%	4.9%
SyntaxSQLNet (BERT)	42.9%	24.9%	21.9%	8.6%
IRNet	**70.1%**	**49.2%**	**39.5%**	**19.1%**
IRNet(BERT)	**77.2%**	**58.7%**	**48.1%**	**25.3%**

Table 2: Exact matching accuracy of SyntaxSQLNet, SyntaxSQLNet(BERT), IRNet and IRNet(BERT) on test set by hardness level.

To investigate the effectiveness of SemQL, we alter the baseline approaches and let them learn to generate SemQL queries rather than SQL queries. As shown in Table 3, there are at least 6.6% and up to 14.4% absolute improvements on accuracy of exact matching on the development set. For example, when SyntaxSQLNet is learned to generate SemQL queries instead of SQL queries, it registers 8.6% absolute improvement and even outperforms SyntaxSQLNet(augment) which performs large-scale data augmentation. The relatively limited improvement on TypeSQL and SQLNet is because their slot-filling based models only support a subset of SemQL queries. The notable improvement, on the one hand, demonstrates the effectiveness of SemQL. On the other hand, it shows that designing an intermediate representations to bridge NL and SQL is promising in Text-to-SQL.

3.3 Ablation Study

We conduct ablation studies on IRNet and IRNet(BERT) to analyze the contribution of each design choice. Specifically, we first evaluate a base model that does not apply schema linking (SL) and the coarse-to-fine framework (CF), and replace the memory augment pointer network (MEM) with the vanilla pointer network (Vinyals et al., 2015). Then, we gradually apply each component on the base model. The ablation study is conducted on the development set.

Approach	SQL	SemQL
Seq2Seq	1.9%	11.4%(**+9.5**)
Seq2Seq + Attention	1.8%	14.7%(**+12.9**)
Seq2Seq + Copying	4.1%	18.5%(**+14.1**)
TypeSQL	8.0%	14.4%(**+6.4**)
SQLNet	10.9%	17.5%(**+6.6**)
SyntaxSQLNet	18.9%	27.5%(**+8.6**)
BERT		
SyntaxSQLNet(BERT)	25.0%	35.8%(**+10.8**)

Table 3: Exact matching accuracy on development set. The header ‘SQL’ means that the approaches are learned to generate SQL, while the header ‘SemQL’ indicates that they are learned to generate SemQL queries.

Technique	IRNet	IRNet(BERT)
Base model	40.5%	53.9%
+SL	48.5%	60.3%
+SL + MEM	51.3%	60.6%
+SL + MEM + CF	53.2%	61.9%

Table 4: Ablation study results. Base model means that we does not perform schema linking (SL), memory augmented pointer network (MEM) and the coarse-to-fine framework (CF) on it.

Table 4 presents the ablation study results. It is clear that our base model significantly outperforms SyntaxSQLNet, SyntaxSQLNet(augment) and SyntaxSQLNet(BERT). Performing schema linking (‘+SL’) brings about 8.5% and 6.4% absolute improvement on IRNet and IRNet(BERT). Predicting columns in the WHERE clause is known to be challenging (Yavuz et al., 2018). The F1 score on the WHERE clause increases by 12.5% when IRNet performs schema linking. The significant improvement demonstrates the effectiveness of schema linking in addressing the lexical problem. Using the memory augmented pointer network (‘+MEM’) further improves the performance of IRNet and IRNet(BERT). We observe that the vanilla pointer network is prone to selecting same columns during synthesis. The number of examples suffering from this problem decreases by 70%, when using the memory augmented pointer network. At last, adopting the coarse-to-fine framework (‘+CF’) can further boost performance.

3.4 Error Analysis

To understand the source of errors, we analyze 483 failed examples of IRNet on the development set. We identify three major causes for failures:

Column Prediction. We find that 32.3% of failed examples are caused by incorrect column predictions based on cell values. That is, the correct column name is not mentioned in a question, but the cell value that belongs to it is mentioned. As the study points out (Yavuz et al., 2018), the cell values of a database are crucial in order to solve this problem. 15.2% of the failed examples fail to predict correct columns that partially appear in questions or appear in their synonym forms. Such failures may can be further resolved by combining our string-match based method with embedding-match based methods (Krishnamurthy et al., 2017) to improve the schema linking in the future.

Nested Query. 23.9% of failed examples are caused by the complicated nested queries. Most of these examples are in the Extra Hard level. In the current training set, the number of SQL queries in Extra Hard level (∼20%) is the least, even less than the SQL queries in Easy level (∼23%). In view of the extremely large search space of the complicated SQL queries, data augmentation techniques may be indispensable.

Operator. 12.4% of failed examples make mistake in the operator as it requires common knowledge to predict the correct one. Considering the following question, 'Find the name and membership level of the visitors whose membership level is higher than 4, and sort by their age from old to young', the phrase 'from old to young' indicates that sorting should be conducted in descending order. The operator defined here includes aggregate functions, operators in `WHERE` clause and the sorting orders (ASC and DESC).

Other failed examples cannot be easily categorized into one of the categories above. A few of them are caused by the incorrect `FROM` clause, because the ground truth SQL queries join those tables without foreign key relations defined in the schema. This violates our assumption that the definition of a database schema should be precise and complete.

When incorporated with BERT, 30.5% of failed examples are fixed. Most of them are in the category Column Prediction and Operator, but the improvement on Nested Query is quite limited.

4 Discussion

Performance Gap. There exists a performance gap on IRNet between the development set and the test set, as shown in Table 1. Considering the explosive combination of nested queries in SQL and the limited number of data (1034 in development, 2147 in test), the gap is probably caused by the different distributions of the SQL queries in Hard and Extra level. To verify the hypothesis, we construct a pseudo test set from the official training set. We train IRNet on the remaining data in the training set and evaluate them on the development set and the pseudo test set, respectively. We find that even though the pseudo set has the same number of complicated SQL queries (Hard and Extra Hard) with the development set, there still exists a performance gap. Other approaches do not exhibit the performance gap because of their relatively poor performance on the complicated SQL queries. For example, SyntaxSQLNet only achieves 4.6% on the SQL queries in Extra Hard level on test set. Supplementary material provides detailed experimental settings and results on the pseudo test set.

Limitations of SemQL. There are a few limitations of our intermediate representation. Firstly, it cannot support the self join in the `FROM` clause of SQL. In order to support the self join, the variable mechanism in lambda calculus (Carpenter, 1997) or the scope mechanism in Discourse Representation Structure (Kamp and Reyle, 1993) may be necessary. Secondly, SemQL has not completely eliminated the mismatch between NL and SQL yet. For example, the `INTERSECT` clause in SQL is often used to express disjoint conditions. However, when specifying requirements, end users rarely concern about whether two conditions are disjointed or not. Despite the limitations of SemQL, experimental results demonstrate its effectiveness in Text-to-SQL. To this end, we argue that designing an effective intermediate representation to bridge NL and SQL is a promising direction to being there for complex and cross-domain Text-to-SQL. We leave a better intermediate representation as one of our future works.

5 Related Work

Natural Language Interface to Database. The task of Natural Language Interface to Database (NLIDB) has received significant attention since the 1970s (Warren and Pereira, 1981; Androutsopoulos et al., 1995; Popescu et al., 2004; Hallett, 2006; Giordani and Moschitti, 2012). Most of the early proposed systems are hand-crafted to a specific database (Warren and Pereira, 1982; Woods, 1986; Hendrix et al., 1978), making it challenging to accommodate cross-domain settings. Later work focus on building a system that can be reused for multiple databases with minimal human efforts (Grosz et al., 1987; Androutsopoulos et al., 1993; Tang and Mooney, 2000). Recently, with the development of advanced neural approaches on Semantic Parsing and the release of large-scale, cross-domain Text-to-SQL benchmarks such as WikiSQL (Zhong et al., 2017) and Spider (Yu et al., 2018c), there is a renewed interest in the task (Xu et al., 2017; Iyer et al., 2017; Sun et al., 2018; Gur et al., 2018; Yu et al., 2018a,b; Wang et al., 2018; Finegan-Dollak et al., 2018; Hwang et al., 2019). Unlike these neural approaches that end-to-end synthesize a SQL query, IRNet first synthesizes a SemQL query and then infers a SQL query from it.

Intermediate Representations in NLIDB. Early proposed systems like as LUNAR (Woods, 1986) and MASQUE (Androutsopoulos et al., 1993) also propose intermediate representations (IR) to represent the meaning of questions and then translate it into SQL queries. The predicates in these IRs are designed for a specific database, which sets SemQL apart. SemQL targets a wide adoption and no human effort is needed when it is used in a new domain. Li and Jagadish (2014) propose a query tree in their NLIDB system to represent the meaning of a question and it mainly serves as an interaction medium between users and their system.

Entity Linking. The insight behind performing schema linking is partly inspired by the success of incorporating entity linking in knowledge base question answering and semantic parsing (Yih et al., 2016; Krishnamurthy et al., 2017; Yu et al., 2018a; Herzig and Berant, 2018; Kolitsas et al., 2018). In the context of semantic parsing, Krishnamurthy et al. (2017) propose a neural entity linking module for answering compositional questions on semi-structured tables. TypeSQL (Yu et al., 2018a) proposes to utilize type information to better understand rare entities and numbers in questions. Similar to TypeSQL, IRNet also recognizes the columns and tables mentioned in a question. What sets IRNet apart is that IRNet assigns different types to the columns based on how they are mentioned in the question.

6 Conclusion

We present a neural approach SemQL for complex and cross-domain Text-to-SQL, aiming to address the lexical problem and the mismatch problem with schema linking and an intermediate representation. Experimental results on the challenging Spider benchmark demonstrate the effectiveness of IRNet.

Acknowledgments

We would like to thank Bo Pang and Tao Yu for evaluating our submitted models on the test set of the Spider benchmark. Ting Liu is the corresponding author.

References

Hiyan Alshawi. 1992. *The core language engine*. MIT press.

I. Androutsopoulos, G. Ritchie, and P. Thanisch. 1993. Masque/sql: An efficient and portable natural language query interface for relational databases. In *Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 327–330. Gordon & Breach Science Publishers.

Ion Androutsopoulos, Graeme D Ritchie, and Peter Thanisch. 1995. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1:29–81.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*. Version 7.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544. Association for Computational Linguistics.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 775–788. ACM.

Bob Carpenter. 1997. *Type-logical semantics*. MIT press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. Version 1.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 731–742. Association for Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 351–360. Association for Computational Linguistics.

Alessandra Giordani and Alessandro Moschitti. 2012. Generating sql queries using natural language syntactic dependencies and metadata. In *Proceedings of the 17th International Conference on Applications of Natural Language Processing and Information Systems*, pages 164–170. Springer-Verlag.

Barbara J. Grosz, Douglas E. Appelt, Paul A. Martin, and Fernando C. N. Pereira. 1987. Team: An experiment in the design of transportable natural-language interfaces. *Artificial Intelligence*, 32:173–243.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1631–1640. Association for Computational Linguistics.

Izzeddin Gur, Semih Yavuz, Yu Su, and Xifeng Yan. 2018. Dialsql: Dialogue based structured query generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 1339–1349. Association for Computational Linguistics.

Catalina Hallett. 2006. Generic querying of relational databases using natural language generation techniques. In *Proceedings of the Fourth International Natural Language Generation Conference*, pages 95–102. Association for Computational Linguistics.

Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3:105–147.

Jonathan Herzig and Jonathan Berant. 2018. Decoupling structure and lexicon for zero-shot semantic parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1619–1629. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9:1735–1780.

Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *arXiv preprint arXiv:1902.01069*. Version 1.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 963–973. Association for Computational Linguistics.

Hans Kamp and U. Reyle. 1993. *From Discourse to Logic Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*.

Rohit Kate. 2008. Transforming meaning representation grammars to improve semantic parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 33–40. Coling 2008 Organizing Committee.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1062–1068. AAAI Press.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. Version 9.

Nikolaos Kolitsas, Octavian-Eugen Ganea, and Thomas Hofmann. 2018. End-to-end neural entity linking. In *Proceedings of the 22nd Conference on Computational Natural Language Learning*, pages 519–529. Association for Computational Linguistics.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526. Association for Computational Linguistics.

Fei Li and H. V. Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8:73–84.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. 2017. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 23–33. Association for Computational Linguistics.

Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408*. Version 2.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599. Association for Computational Linguistics.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, pages 1470–1480. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543. Association for Computational Linguistics.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th International Conference on Computational Linguistics*. Association for Computational Linguistics.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. thesis, Berkeley, CA, USA. AAI3353225.

Robert Speer and Catherine Havasi. 2012. Representing general relational knowledge in conceptnet 5. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation*, pages 3679–3686. European Language Resources Association.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax- and table-aware sql generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 361–372. Association for Computational Linguistics.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, pages 3104–3112. MIT Press.

Lappoon R. Tang and Raymond J. Mooney. 2000. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics*, pages 133–141. Association for Computational Linguistics.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100*. Version 3.

David H. D. Warren and Fernando Pereira. 1981. Easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8:110–122.

David H. D. Warren and Fernando C. N. Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8:110–122.

W A Woods. 1986. Readings in natural language processing. chapter Semantics and Quantification in Natural Language Question Answering, pages 205–248. Morgan Kaufmann Publishers Inc.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*. Version 1.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1:63:1–63:26.

Semih Yavuz, Izzeddin Gur, Yu Su, and Xifeng Yan. 2018. What it takes to achieve 100 percent condition accuracy on wikisql. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1702–1711. Association for Computational Linguistics.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 201–206. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 440–450. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12. Association for Computational Linguistics.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018a. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 588–594. Association for Computational Linguistics.

Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018b. Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1653–1663. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018c. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1050–1055. AAAI Press.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

7 Supplemental Material

7.1 Examples of SemQL Query

Figure 10 presents more examples of SemQL queries.

7.2 Inference of SQL Query

To infer a SQL query from a SemQL query, we traverse the tree-structured SemQL query in pre-order and map each tree node to the corresponding SQL query components according to the production rule applied to it.

The production rule applied to the *Z* node denotes whether the SQL query has one of the following components, `UNION`, `EXCEPT` and `INTERSECT`. The *R* node stands for the start of a single SQL query. The production rule applied to R denotes whether the SQL query has a `WHERE` clause and `ORDERBY` clause. The production rule applied to a *Select* node denotes how many columns does the `SELECT` clause has. Each *A* node denotes a column/aggregate function pair. Specifically, nodes under *A* denote the aggregate function, the column name and the table name of the column. The subtrees under nodes *Superlative* and *Order* are mapped to the `ORDERBY` clause in the SQL query. The production rules applied to *Filter* denote different condition operators in SQL query, e.g. `and`, `or`, `>`, `<`, `=`, `in`, `not in` and so on. If there is a *A* node under the *Filter* node and its aggregate function is not $None$, it will be filled in the `HAVING` clause, otherwise in the `WHERE` clause. If there is a *R* node under the *Filter* node, we will repeat the process recursively on the *R* node and return a nested SQL query. The `FROM` clause is generated from the selected tables in the SemQL query by identifying the shortest path that connects these tables in the schema (Database schema can be formulated as an undirected graph, where vertex are tables and edges are relations among tables). At last, if there exists an aggregate function applied on a column in the SemQL query, there should be `GROUPBY` clause in the SQL query. The column to be grouped by occurs in the `SELECT` clause in most cases, or it is the primary key of a table where an aggregate function is applied on one of its columns.

7.3 Transforming SQL to SemQL

To generate a SemQL query from a SQL query, we first initialize a *Z* node. If the SQL query has one of the components `UNION`, `EXCEPT` and `INTERSECT`, we attach the corresponding keywords and two *R* nodes under *Z*, otherwise a single *R* node. Then, we attach a *Select* node under *R*, and the number of columns in `SELECT` clause determines the number of *A* nodes under the *Select* node. If an `ORDERBY` clause in a SQL query contains a `LIMIT` keyword, it will be transformed into a *Superlative* node, otherwise a *Order* node. Next, the sub-tree of *Filter* node is determined by the condition in `WHERE` and `HAVING` clause. If it has a nested query in `WHERE` clause or `HAVING` clause, we process the subquery recursively. For each column in a SQL query, we attach its aggregate function node, a *C* node and a *T* node under *A*.

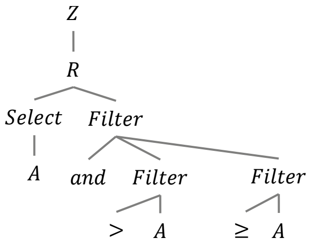


Figure 6: The skeleton of the SemQL query presented in Figure 3

Node C attaches the column name and node T attaches its table name. For the special column '$\star$', if there is only one table in the `FROM` clause that does not belongs to any column, we assign it the column '$\star$', otherwise, we label the table name of '$\star$' manually. If a table in `FROM` clause is not assigned to any column, it will be transformed into a subtree under a $Filter$ node with in condition. In this way, a SQL query can be successfully transformed into a SemQL query.

7.4 Coarse-to-Fine Framework

The skeleton of a SemQL query is obtained by removing all nodes under each A node. Figure 6 shows the skeleton of the SemQL query presented in Figure 3.

Figure 7 depicts the coarse-to-fine framework to synthesize a SemQL query. In the first stage, a skeleton decoder outputs the skeleton of a SemQL query. Then, a detail decoder fills in the missing details in the skeleton by selecting columns and tables. The probability of generating a SemQL query y in the coarse-to-fine framework is formalized as follows.

$$\begin{aligned} p(y|x,s) &= p(q|x,s)p(y|x,s,q) \\ p(q|x,s) &= \prod_{i=1}^{T_s} p(a_i = \textsc{ApplyRule}[r]|x,s,a_{<i}) \\ p(y|x,s,q) &= \prod_{i=1}^{T_c} [\lambda_i p(a_i = \textsc{SelectColumn}[c]|x,s,q,a_{<i}) \\ &\quad + (1-\lambda_i)p(a_i = \textsc{SelectTable}[t]|x,s,q,a_{<i})] \end{aligned}$$

where q denotes the skeleton. $\lambda_i = 1$ when the ith action type is SelectColumn, otherwise $\lambda_i = 0$.

At training time, our model is optimized by maximizing the log-likelihood of the ground true action sequences:

$$max \sum_{(x,s,q,y)\in\mathcal{D}} \log p(y|x,s,q) + \gamma \log p(q|x,s)$$

where $\mathcal{D}$ denotes the training data and γ represents the scale between $\log p(y|x,s,q)$ and $\log p(q|x,s)$. γ is set to 1 in our experiment.

7.5 BERT

Figure 8 depicts the architecture of the BERT encoder.

7.6 Analysis on the Performance Gap between the Development set and the Test set

Dataset	Easy	Medium	Hard	Extra Hard
Pseudo Test A	24.2%	44.5%	14.4%	16.9%
Pseudo Test B	22.7%	44.1%	16.7%	16.5%
Pseudo Test C	24.7%	37.1%	22.9%	15.3%
Development	**24.1%**	**42.5%**	**16.8%**	**16.4%**

Table 5: The hardness distribution of the pseudo test A, the pseudo test B, the pseudo test C and the development set.

To test our hypothesis that the performance gap is caused by the different distribution of the SQL queries in Hard and Extra Hard level, we first construct a pseudo test set from the official training set of Spider benchmark. Then, we conduct further experiment on the pseudo test set and the official development set. Specifically, we sample 20 databases from the training set to construct a pseudo test set, which has the same hardness distributions with the development set. Then, we train IRNet on the remaining training set, and evaluate it on the development set and the pseudo test set, respectively. We sample the pseudo test set from the training set for three times and obtain three pseudo test sets, namely, pseudo test A, pseudo test B and pseudo test C. They contain 1134, 1000 and 955 test data respectively.

Table 5 presents the hardness distribution of the three pseudo test sets and the official development set. Figure 9 presents the exact matching accuracy of SQL on the development set and three pseudo tests set after each epoch during training. IRNet performs competitively on the development set and the pseudo set C (Figure 9c), but there exists a clear performance gap on the pseudo test A and B (Figure 9a and Figure 9b). Although the hardness distributions among the development set and the three pseudo sets are nearly the same, the data distribution still has some difference, which results in the performance gap.

We further study the performance gap of SyntaxSQLNet on the development set and the pseudo test A. As shown in Table 6, SyntaxSQLNet

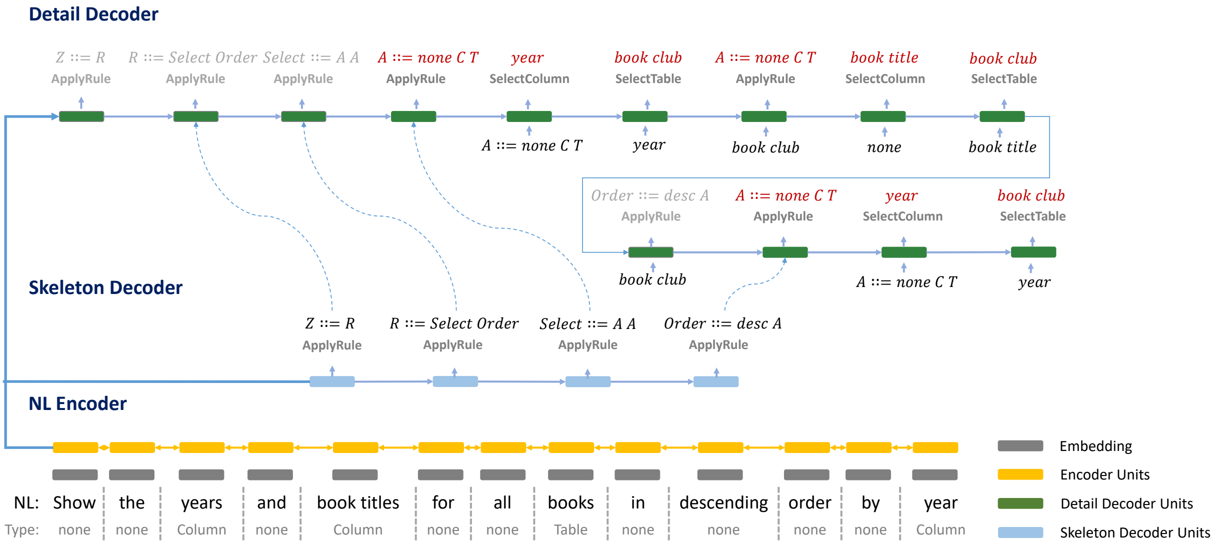


Figure 7: An overview of the coarse-to-fine framework to synthesize SemQL.

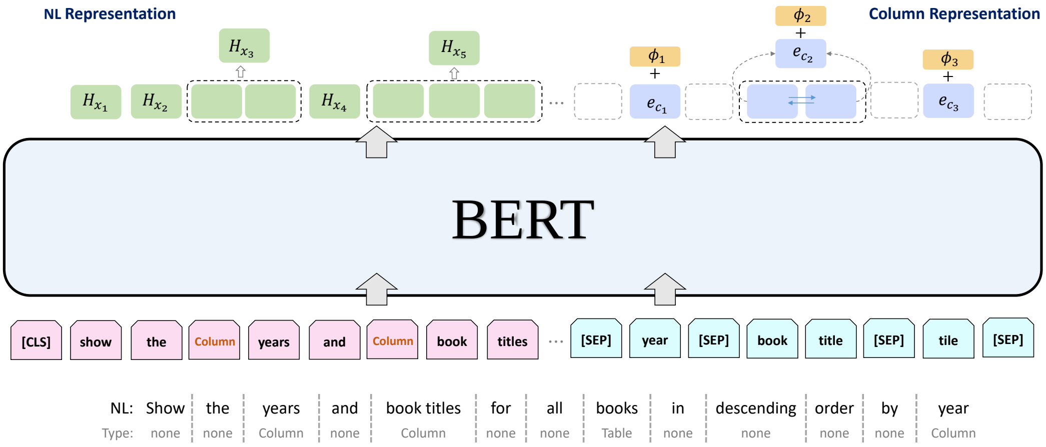
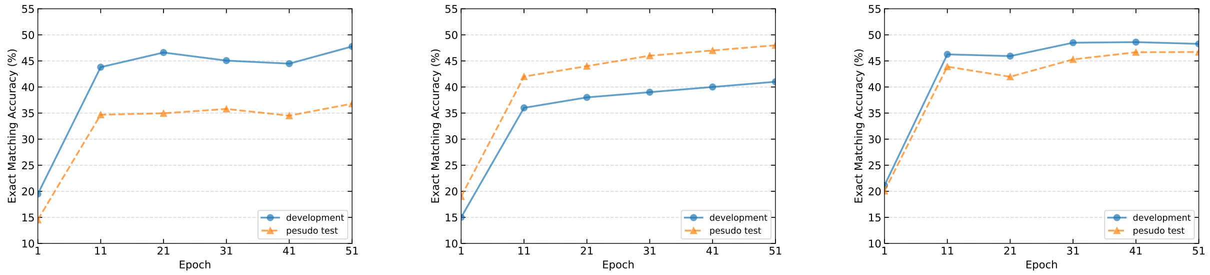


Figure 8: Encoding a question and column names with BERT.

Approach	Dev			Pseudo Test A		
	All	Hard	Extra Hard	All	Hard	Extra Hard
SyntaxSQLNet	17.4%	15.5%	2.9%	16.9%	11.0%	2.6%
+BERT +SemQL	34.5%	30.5%	17.6%	30.2%	24.5%	15.7%

Table 6: Exact matching accuracy on the development set and the pseudo test A set.



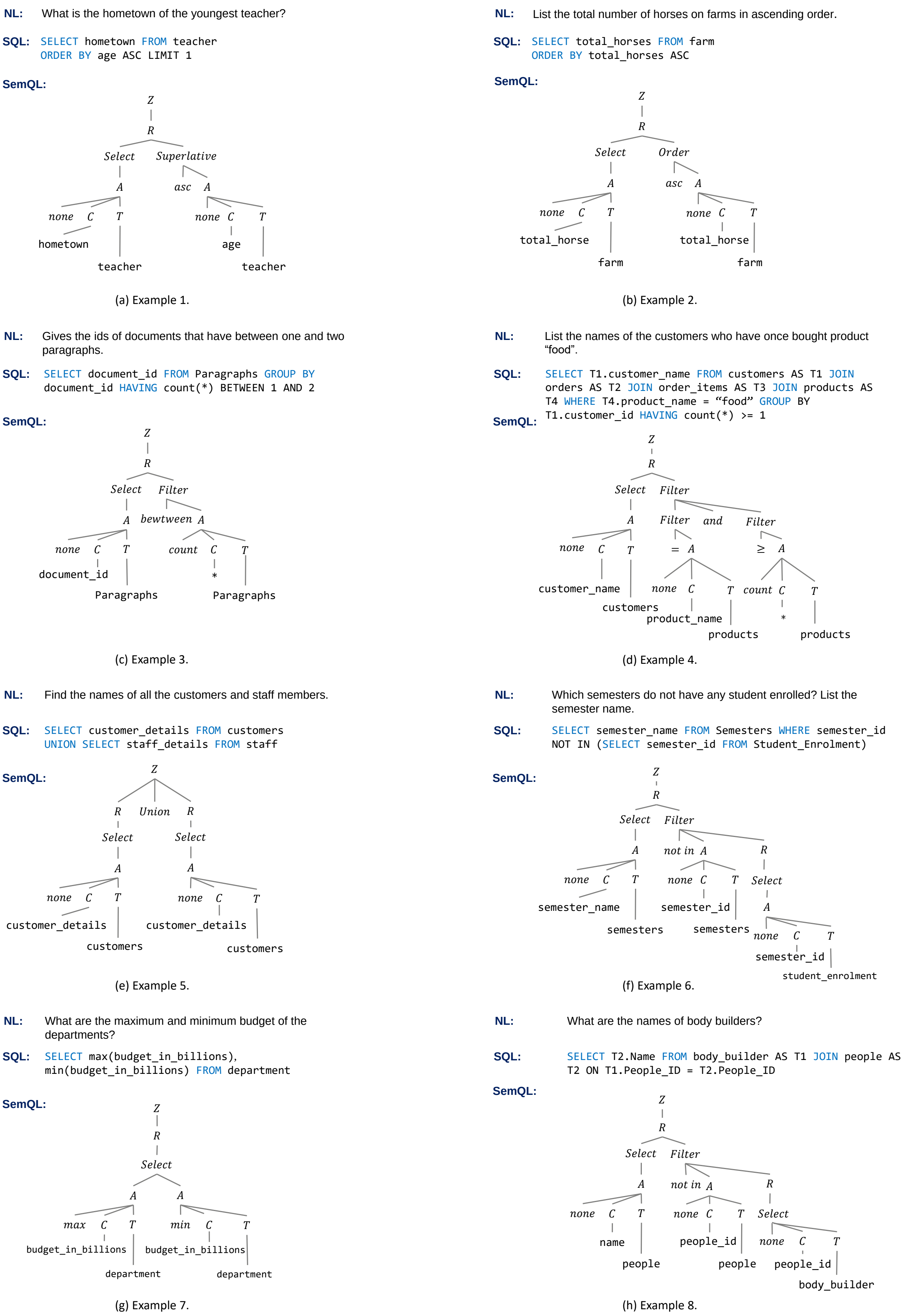
(a) Pseudo Test A

(b) Pseudo Test B

(c) Pseudo Test C

Figure 9: Exact matching accuracy of IRNet on development set and pseudo test sets.

achieves 16.9% on the development set and 17.4% on the pseudo test A. When incorporating BERT and learning to synthesizing SemQL, SyntaxSQL-Net(BERT,SemQL) achieves 34.5% on the development set and 30.2% on the pseudo test A, exhibiting a clear performance gap (4.3%). SyntaxSQLNet(BERT, SemQL) significantly outperforms SyntaxSQLNet in the Hard and Extra Hard level. The experimental results show that when SyntaxSQLNet performs better in the Hard and Extra Hard level, the performance gap will be larger, since that the performance gap is caused by the different data distributions.



NL: What is the hometown of the youngest teacher?

SQL:

```
SELECT hometown FROM teacher
ORDER BY age ASC LIMIT 1
```

SemQL:

(a) Example 1.

NL: List the total number of horses on farms in ascending order.

SQL:

```
SELECT total_horses FROM farm
ORDER BY total_horses ASC
```

SemQL:

(b) Example 2.

NL: Gives the ids of documents that have between one and two paragraphs.

SQL:

```
SELECT document_id FROM Paragraphs GROUP BY
document_id HAVING count(*) BETWEEN 1 AND 2
```

SemQL:

(c) Example 3.

NL: List the names of the customers who have once bought product “food”.

SQL:

```
SELECT T1.customer_name FROM customers AS T1 JOIN
orders AS T2 JOIN order_items AS T3 JOIN products AS
T4 WHERE T4.product_name = “food” GROUP BY
T1.customer_id HAVING count(*) >= 1
```

SemQL:

(d) Example 4.

NL: Find the names of all the customers and staff members.

SQL:

```
SELECT customer_details FROM customers
UNION SELECT staff_details FROM staff
```

SemQL:

(e) Example 5.

NL: Which semesters do not have any student enrolled? List the semester name.

SQL:

```
SELECT semester_name FROM Semesters WHERE semester_id
NOT IN (SELECT semester_id FROM Student_Enrolment)
```

SemQL:

(f) Example 6.

NL: What are the maximum and minimum budget of the departments?

SQL:

```
SELECT max(budget_in_billions),
min(budget_in_billions) FROM department
```

SemQL:

(g) Example 7.

NL: What are the names of body builders?

SQL:

```
SELECT T2.Name FROM body_builder AS T1 JOIN people AS
T2 ON T1.People_ID = T2.People_ID
```

SemQL:

(h) Example 8.

Figure 10: Examples of SemQL Query.