RYANSQL: Recursively Applying Sketch-based Slot Fillings for Complex Text-to-SQL in Cross-Domain Databases

DongHyun Choi
Natural Language Processing Team
Kakao Enterprise and School of Software
Sungkyunkwan University
heuristic.c@kakaoenterprise.com

Myeong Cheol Shin
Natural Language Processing Team
Kakao Enterprise
index.sh@kakaoenterprise.com

EungGyun Kim
Natural Language Processing Team
Kakao Enterprise
jason.ng@kakaoenterprise.com

Dong Ryeol Shin
School of Software
Sungkyunkwan University
drshin@skku.edu

Text-to-SQL is the problem of converting a user question into an SQL query, when the question and database are given. In this article, we present a neural network approach called RYANSQL (Recursively Yielding Annotation Network for SQL) to solve complex Text-to-SQL tasks for cross-domain databases. Statement Position Code (SPC) is defined to transform a nested SQL query into a set of non-nested `SELECT` statements; a sketch-based slot-filling approach is proposed to synthesize each `SELECT` statement for its corresponding SPC. Additionally, two input manipulation methods are presented to improve generation performance further. RYANSQL achieved competitive result of 58.2% accuracy on the challenging Spider benchmark. At the time of submission (April 2020), RYANSQL v2, a variant of original RYANSQL, is positioned at 3rd place among all systems and 1st place among the systems not using database content

Submission received: 12 April 2020; revised version received: 3 January 2021; accepted for publication: 4 March 2021.

https://doi.org/10.1162/COLI_a_00403

with 60.6% exact matching accuracy. The source code is available at `https://github.com/kakaoenterprise/RYANSQL`.

1. Introduction

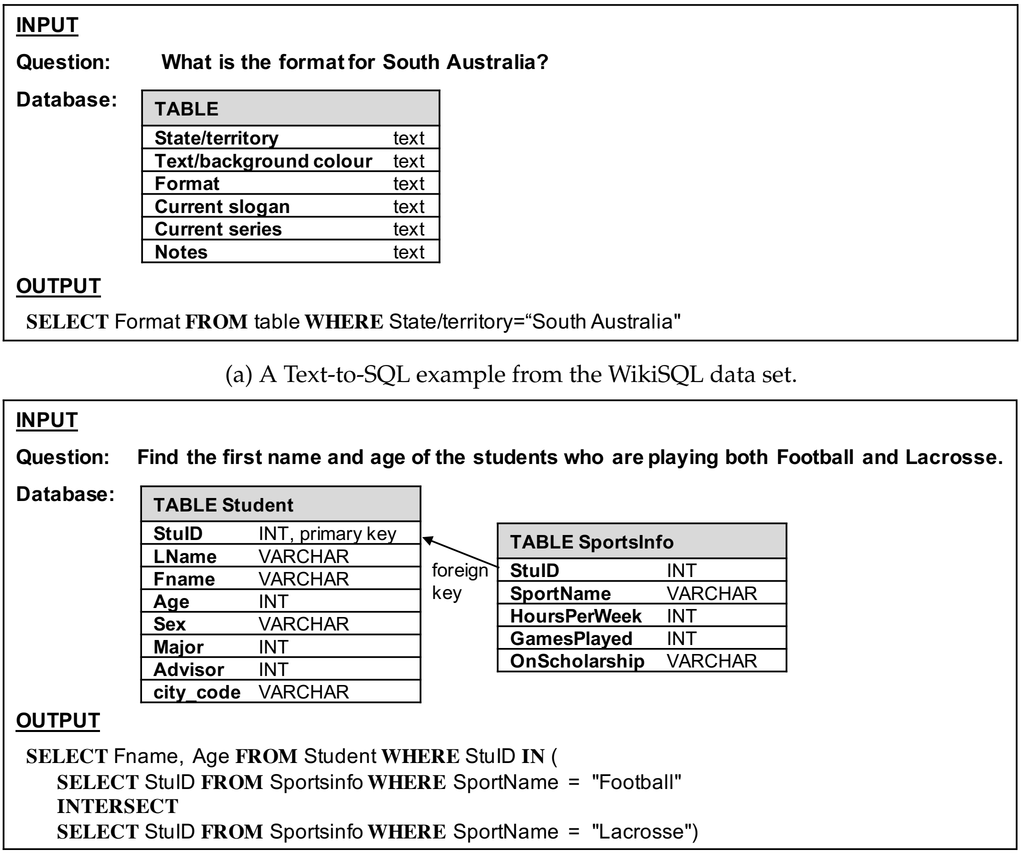
Relational databases are widely used to maintain and query structured data sets in many fields such as healthcare (Hillestad et al. 2005), financial markets (Beck, Demirguc-Kunt, and Levine 2000), or customer relation management (Ngai, Xiu, and Chau 2009). Most relational databases support Structured Query Language (SQL) to access the stored data. Although SQL is expressive and powerful, it is quite difficult to master, especially for non-technical users.

Text-to-SQL is the task of generating an SQL query when a user question and a target database are given. The examples are shown in Figure 1. Recently proposed neural network architectures achieved more than 80% exact matching accuracy on the well-known Text-to-SQL benchmarks such as ATIS (Air Travel Information Service), GeoQuery, and WikiSQL (Xu, Liu, and Song 2017; Yu et al. 2018a; Shi et al. 2018; Dong and Lapata 2018; Hwang et al. 2019; He et al. 2018). However, those benchmarks have shortcomings that restrict their applications. The ATIS (Price 1990) and GeoQuery (Zelle and Mooney 1996) benchmarks assume the same database across the training and test data set, thus the trained systems cannot process a newly encountered database at inference time. The WikiSQL (Zhong, Xiong, and Socher 2017) benchmark assumes cross-domain databases. Cross-domain means that the databases for training and test data sets are different; the system should predict with an unseen database as its input during testing. Meanwhile, the complexity of SQL queries and databases in the WikiSQL benchmark is somewhat limited. WikiSQL assumes that an input database always has only one table. It also assumes that the resultant SQL is non-nested, and contains `SELECT` and `WHERE` clauses only. Figure 1(a) shows an example from the WikiSQL data set.

Different from those benchmarks, the Spider benchmark proposed by Yu et al. (2018c) contains complex SQL queries with cross-domain databases. The SQL queries in Spider benchmark could contain nested queries with multiple table `JOIN`s, and clauses like `ORDERBY`, `GROUPBY`, and `HAVING`. Figure 1(b) shows an example from the Spider benchmark; Yu et al. (2018c) showed that the state-of-the-art systems for the previous benchmarks do not perform well on the Spider data set.

In this article, we propose a novel network architecture called RYANSQL (Recursively Yielding Annotation Network for SQL) to handle such complex, cross-domain Text-to-SQL problems. The proposed approach generates nested queries by recursively yielding their component `SELECT` statements. A sketch-based slot-filling approach is proposed to predict each `SELECT` statement. In addition, two simple but effective input manipulation methods are proposed to improve the overall system performance. Among the systems not using database content, the proposed RYANSQL and its variant RYANSQL v2, with the aid of BERT (Devlin et al. 2019), improve the previous state-of-the-art system SLSQL (Lei et al. 2020) by 2.5% and 4.9%, respectively, in terms of the test set exact matching accuracy. RYANSQL v2 is ranked at 3rd place among all systems including those using database content. Our contributions are summarized as follows:

- We propose a detailed sketch for the complex `SELECT` statements, along with a network architecture to fill the slots.
- Statement Position Code (SPC) is introduced to recursively predict nested queries with sketch-based slot-filling algorithm.



INPUT

Question: **What is the format for South Australia?**

Database:

TABLE	
State/territory	text
Text/background colour	text
Format	text
Current slogan	text
Current series	text
Notes	text

OUTPUT

```
SELECT Format FROM table WHERE State/territory="South Australia"
```

(a) A Text-to-SQL example from the WikiSQL data set.

INPUT

Question: **Find the first name and age of the students who are playing both Football and Lacrosse.**

Database:

TABLE Student	
StuID	INT, primary key
LName	VARCHAR
Fname	VARCHAR
Age	INT
Sex	VARCHAR
Major	INT
Advisor	INT
city_code	VARCHAR

TABLE SportsInfo	
StuID	INT
SportName	VARCHAR
HoursPerWeek	INT
GamesPlayed	INT
OnScholarship	VARCHAR

OUTPUT

```
SELECT Fname, Age FROM Student WHERE StuID IN (
    SELECT StuID FROM Sportsinfo WHERE SportName = "Football"
    INTERSECT
    SELECT StuID FROM Sportsinfo WHERE SportName = "Lacrosse")
```

(b) A Text-to-SQL example from the Spider data set.

Figure 1
Text-to-SQL examples.

- We suggest two simple input manipulation methods to improve performance further.

2. Task Definition

The Text-to-SQL task considered in this article is defined as follows: Given a question with n tokens $Q = \{w_1^Q, \ldots, w_n^Q\}$ and a DB schema with t tables and f foreign key relations $D = \{T_1, \ldots, T_t, F_1, \ldots, F_f\}$, find S, the SQL translation of Q. Each table T_i consists of a table name with t_i words $\{w_1^{T_i}, \ldots, w_{t_i}^{T_i}\}$, and a set of columns $\{C_j, \ldots, C_k\}$. Each column C_j consists of a column name $\{w_1^{C_j}, \ldots, w_{c_j}^{C_j}\}$, and a marker to check if the column is a primary key.

For an SQL query S we define a non-nested form of S, $N(S) = \{(P_1, S_1), \ldots, (P_l, S_l)\}$. In the definition, P_i is the i-th SPC, and S_i is its corresponding `SELECT` statement. Table 1 shows examples of a natural language query Q, corresponding SQL translation S, and non-nested form $N(S)$.

Table 1
Examples of a user question Q, SQL translation S, and non-nested form $N(S)$.

Case 1	**Q**		Find the names of scientists who are not working on the project with the highest hours.
	S		SELECT **name** FROM **scientists** EXCEPT (SELECT **T3.name** FROM **assignedto** AS **T1** JOIN **projects** AS **T2** ON **T1.project** = **T2.code** JOIN **scientists** AS **T3** ON **T1.scientist** = **T3.SSN** WHERE **T2.hours** = (SELECT **max(hours)** FROM **projects**))
	N(S)	P_1	[NONE]
		S_1	SELECT **name** FROM **scientists** EXCEPT S_2
		P_2	[EXCEPT]

		S_2	SELECT **T3.name** FROM **assignedto** AS **T1**
			JOIN **projects** AS **T2** ON **T1.project** = **T2.code**
			JOIN **scientists** AS **T3** ON **T1.scientist** = **T3.SSN**
			WHERE **T2.hours** = S_3

		P_3	[EXCEPT, WHERE]
		S_3	SELECT **max(hours)** FROM **projects**
Case 2	**Q**		Find the names of accounts whose checking balance is above the average checking balance, but savings balance is below the average savings balance.

	S		SELECT **T1.name** FROM **accounts** AS **T1**
			JOIN **checking** AS **T2** ON **T1.custid** = **T2.custid**
			WHERE **T2.balance** > (SELECT **avg(balance)** FROM **checking**)
			INTERSECT
			SELECT **T1.name** FROM **accounts** AS **T1**
			JOIN **savings** AS **T2** ON **T1.custid** = **T2.custid**
			WHERE **T2.balance** < (SELECT **avg(balance)** FROM **savings**)

	N(S)	P_1	[NONE]

		S_1	SELECT **T1.name** FROM **accounts** AS **T1**
			JOIN **checking** AS **T2** ON **T1.custid** = **T2.custid**
			WHERE **T2.balance** > S_2
			INTERSECT S_3

		P_2	[WHERE]
		S_2	SELECT **avg(balance)** FROM **checking**
		P_3	[INTERSECT]

		S_3	SELECT **T1.name** FROM **accounts** AS **T1**
			JOIN **savings** AS **T2** ON **T1.custid** = **T2.custid**
			WHERE **T2.balance** < S_4

		P_4	[INTERSECT, WHERE]
		S_4	SELECT **avg(balance)** FROM **savings**

Each SPC P could be considered as a sequence of p position code elements, $P = [c_1^P, \ldots, c_p^P]$. The possible set of position code elements is {NONE, UNION, INTERSECT, EXCEPT, WHERE, HAVING, PARALLEL}. NONE represents the outermost statement, and PARALLEL means the parallel elements inside a single clause, for example,

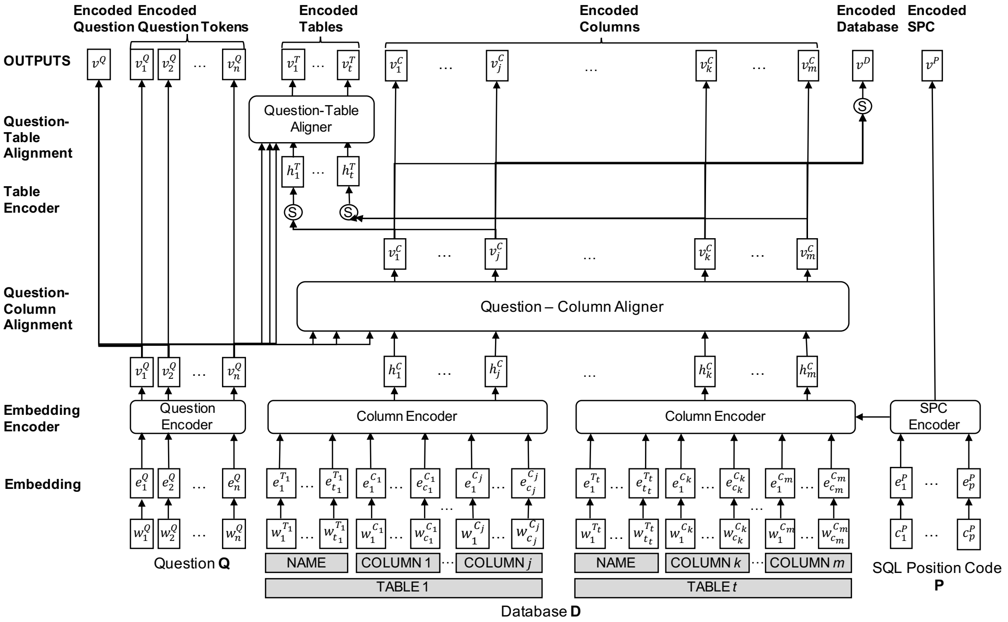

Encoded Question
Encoded Question Tokens
Encoded Tables
Encoded Columns
Encoded Database
Encoded SPC
OUTPUTS
v^Q
v_1^Q v_2^Q … v_n^Q
v_1^T … v_t^T
v_1^C … v_j^C … v_k^C … v_m^C
v^D
v^P
Question-Table Alignment
Question-Table Aligner
Table Encoder
h_1^T … h_t^T
S
Question-Column Alignment
Question – Column Aligner
v_1^C … v_j^C … v_k^C … v_m^C
v_1^Q v_2^Q … v_n^Q
h_1^C … h_j^C … h_k^C … h_m^C
Embedding Encoder
Question Encoder
Column Encoder
Column Encoder
SPC Encoder
Embedding
e_1^Q e_2^Q … e_n^Q
$e_1^{T_1}$ … $e_{t_1}^{T_1}$ $e_1^{C_1}$ … $e_{c_1}^{C_1}$ e_1^{Cj} … e_{cj}^{Cj}
e_1^{Tt} … e_{tt}^{Tt} e_1^{Ck} … e_{ck}^{Ck} e_1^{Cm} … e_{cm}^{Cm}
e_1^P … e_p^P
w_1^Q w_2^Q … w_n^Q
$w_1^{T_1}$ … $w_{t_1}^{T_1}$ $w_1^{C_1}$ … $w_{c_1}^{C_1}$ w_1^{Cj} … w_{cj}^{Cj}
w_1^{Tt} … w_{tt}^{Tt} w_1^{Ck} … w_{ck}^{Ck} w_1^{Cm} … w_{cm}^{Cm}
c_1^P … c_p^P
Question **Q**
NAME
COLUMN 1
COLUMN *j*
TABLE 1
NAME
COLUMN *k*
COLUMN *m*
TABLE *t*
Database **D**
SQL Position Code **P**

Figure 2
Network architecture of the proposed input encoder. Ⓢ represents self-attention.

the second element of the `WHERE` clause. Other position code elements represent corresponding SQL clauses.

Because it is straightforward to construct S from $N(S)$, the goal of the proposed system is to construct $N(S)$ for the given Q and D. To achieve the goal, the proposed system first sets the initial SPC $P_1 = [\texttt{NONE}]$, and predicts its corresponding `SELECT` statement and nested SPCs. The system recursively finds out the corresponding `SELECT` statements for the remaining SPCs, until every SPC has its own corresponding `SELECT` statement.

3. Generating a `SELECT` Statement

In this section, the method to create the `SELECT` statement for the given question Q, database D, and SPC P is described. Section 3.1 describes the input encoder; the sketch-based slot-filling decoder is described in Section 3.2.

3.1 Input Encoder

Figure 2 shows the overall network architecture of the input encoder. The input encoder consists of five layers: Embedding layer, Embedding Encoder layer, Question-Column Alignment layer, Table Encoder layer, and Question-Table Alignment layer.

Embedding Layer. To get the embedding vector for a word w in question, table names, or column names, its word embedding and character embedding are concatenated. The word embedding is initialized with $d_1 = 300$ dimensional pretrained GloVe (Pennington, Socher, and Manning 2014) word vectors, and is fixed during training. For character embedding, each character is represented as a trainable vector of dimension

$d_2 = 50$, and we take maximum value of each dimension of component characters to get the fixed-length vector. The two vectors are then concatenated to obtain the embedding vector $e_w \in \mathbb{R}^{d_1+d_2}$. One layer highway network (Srivastava, Greff, and Schmidhuber 2015) is applied on top of this representation. For SPC P, each code element c is represented as a trainable vector of dimension $d_p = 100$.

Embedding Encoder Layer. A one-dimensional convolution layer with kernel size 3 is applied on top of SPC element embedding vectors $\{e_1^P, \dots, e_p^P\}$. Max-pooling is applied on the output to get the SPC vector $v^P \in \mathbb{R}^{d_p}$. v^P is then concatenated to each question and column word embedding vector.

CNN with dense connection proposed in Yoon, Lee, and Lee (2018) is applied to encode each word of question and columns to capture local information. Parameters are shared across the question and columns. For each column, a max-pooling layer is followed; outputs are concatenated with their table name representations and projected to dimension d. Layer outputs are encoded question word vectors $V^Q = \{v_1^Q, v_2^Q, \dots, v_n^Q\} \in \mathbb{R}^{n \times d}$, and hidden column vectors $H^C = \{h_1^C, \dots, h_m^C\} \in \mathbb{R}^{m \times d}$.

Question-Column Alignment Layer. In this layer, the model tries to update the column vectors with the input question. More precisely, the model first aligns question tokens with column vectors to obtain an attended question vector for each column. The attended question vectors are then fused with corresponding column vectors to get question context-integrated column vectors. Scaled dot-product attention (Vaswani et al. 2017) is used to align question tokens with column vectors:

$$A_{QtoC} = \text{softmax}(\frac{H^C (V^Q)^\intercal}{\sqrt{d}}) V^Q \tag{1}$$

where each i-th row of $A_{QtoC} \in \mathbb{R}^{m \times d}$ is an attended question vector of the i-th column.

The heuristic fusion function *fusion*(x, y), proposed in Hu et al. (2018), is applied to merge A_{QtoC} with H^C:

$$\begin{aligned} \tilde{x} &= \text{relu}(W_r[x; y; x \circ y; x - y]) \\ g &= \sigma(W_g[x; y; x \circ y; x - y]) \\ \mathit{fusion}(x, y) &= g \circ \tilde{x} + (1 - g) \circ x \\ F^C &= \mathit{fusion}(A_{QtoC}, H^C) \end{aligned} \tag{2}$$

where W_r and W_g are trainable variables, σ denotes the sigmoid function, $\circ$ denotes element-wise multiplication, and $F^C \in \mathbb{R}^{m \times d}$ is fused column matrix.

Once the column vectors are updated with the question context, a transformer layer (Vaswani et al. 2017) is applied on top of F^C to capture contextual column information. Layer outputs are the encoded column vectors $V^C = \{v_1^C, \dots, v_m^C\} \in \mathbb{R}^{m \times d}$.

Table Encoder Layer. Column vectors belonging to each table are integrated to get the encoded table vector. For a matrix $M \in \mathbb{R}^{n \times d}$, self-attention function $f_s(M) \in \mathbb{R}^{1 \times d}$ is defined as follows:

$$f_s(M) = \text{softmax}(W_2 \tanh(W_1 M^\intercal)) M \tag{3}$$

where $W_1 \in \mathbb{R}^{d \times d}$, $W_2 \in \mathbb{R}^{1 \times d}$ are trainable parameters. Then, for table t with columns $\{C_j, \ldots, C_k\}$, the hidden table vector h_t^T is calculated as follows:

$$h_t^T = f_s([v_j^C; ...; v_k^C]) \tag{4}$$

Outputs of the layer are the hidden table vectors $H^T = \{h_1^T, h_2^T, \ldots, h_t^T\} \in \mathbb{R}^{t \times d}$.

Question-Table Alignment Layer. In this layer, the same network architecture as the Question-Column Alignment layer is used to model the table vectors with contextual information of the question. Layer outputs are the encoded table vectors $V^T = \{v_1^T, v_2^T, \ldots, v_t^T\} \in \mathbb{R}^{t \times d}$.

Encoder Output. Final outputs of the input encoder are as follows: (1) Encoded question word vectors $V^Q = \{v_1^Q, \ldots, v_n^Q\} \in \mathbb{R}^{n \times d}$, (2) Encoded column vectors $V^C = \{v_1^C, \ldots, v_m^C\} \in \mathbb{R}^{m \times d}$, (3) Encoded table vectors $V^T = \{v_1^T, \ldots, v_t^T\} \in \mathbb{R}^{t \times d}$, and (4) Encoded SPC $v^P \in \mathbb{R}^{d_p}$. Additionally, (5) Encoded question vector $v^Q = f_s(V^Q)$ and (6) Encoded DB schema vector $v^D = f_s(V^C) \in \mathbb{R}^d$ are calculated for later use in the decoder.

3.1.1 BERT-based Input Encoder. Inspired by the work of Hwang et al. (2019) and Guo et al. (2019), BERT (Devlin et al. 2019) is considered as another version of the input encoder. The input to BERT is constructed by concatenating question words, SPC elements, and column words as follows: `[CLS]`, w_1^Q, ..., w_n^Q, `[SEP]`, c_1^P, ..., c_p^P, `[SEP]`, $w_1^{C_1}, \ldots, w_{c_1}^{C_1}$, `[SEP]`, ..., `[SEP]`, $w_1^{C_m}, \ldots, w_{c_m}^{C_m}$, `[SEP]`.

Hidden states of the last layer are retrieved to form V^Q and V^C; for V^C, the state of each column's last word is taken to represent an encoded column vector. Each table vector v_j^T is calculated as a self-attended vector of its containing columns; v^Q, v^D, and v^P are calculated as the same.

3.2 Sketch-based Slot-Filling Decoder

Table 2 shows the proposed sketch for a SELECT statement. The sketch-based slot-filling decoder predicts values for slots of the proposed sketch, as well as the number of slots.

Classifying Base Structure. By the term **base structure** of a SELECT statement, we refer to the existence of its component clauses and the number of conditions for each clause. We first combine the encoded vectors v^Q, v^D, and v^P to obtain the statement encoding vector v^S, as follows:

$$hc(x, y) = concat(x, y, |x - y|, x \circ y) \tag{5}$$

$$v^S = W\, concat(hc(v^Q, v^D), v^P) \tag{6}$$

where $W \in \mathbb{R}^{d \times (4d + d_p)}$ is a trainable parameter, and function $hc(x, y)$ is the concatenation function for the heuristic matching method proposed in Mou et al. (2016).

Eleven values $b_g, b_o, b_l, b_w, b_h, n_g, n_o, n_s, n_w$, n_h, and $c_{\texttt{IUEN}}$ are classified by applying two fully connected layers on v^S. Binary values b_g, b_o, b_l, b_w, b_h represent the existence of GROUPBY, ORDERBY, LIMIT, WHERE, and HAVING, respectively. Note that FROM and SELECT clauses must exist to form a valid SELECT statement. n_g, n_o, n_s, n_w, n_h represent

Table 2
Proposed sketch for a SELECT statement. $TBL and $COL represent a table and a column, respectively. $AGG is one of {**none**, **max**, **min**, **count**, **sum**, **avg**}, $ARI is one of the arithmetic operators {**none**, **-**, **+**, *****, **/** }, and $COND is one of the conditional operators {**between**, **=**, **>**, **<**, **>=**, **<=**, **!=**, **in**, **like**, **is**, **exists**}. $DIST and $NOT are Boolean variables representing the existence of keywords DISTINCT and NOT, respectively. $ORD is a binary value for keywords ASC/DESC, and $CONJ is one of conjunctions {AND, OR}. $VAL is the value for WHERE/HAVING condition; $SEL represents the slot for another SELECT statement.

CLAUSE	**SKETCH**
FROM	($TBL)$^+$
SELECT	$DIST ($AGG ($DIST$_1$ AGG_1$ COL_1$ $ARI $DIST$_2$ AGG_2$ COL_2$)) $^+$
ORDERBY	(($DIST$_1$ AGG_1$ COL_1$ $ARI $DIST$_2$ AGG_2$ COL_2$) $ORD) *
GROUPBY	($COL)*
LIMIT	$NUM
WHERE HAVING	($CONJ ($DIST$_1$ AGG_1$ COL_1$ $ARI $DIST$_2$ AGG_2$ COL_2$) $NOT $COND VAL_1$\|SEL_1$ VAL_2$\|$SEL$_2$)*
INTERSECT UNION EXCEPT	$SEL

the number of conditions in GROUPBY, ORDERBY, SELECT, WHERE, and HAVING clauses, respectively. The maximal numbers of conditions $N_g = 3$, $N_o = 3$, $N_s = 6$, $N_w = 4$, and $N_h = 2$ are defined for GROUPBY, ORDERBY, SELECT, WHERE, and HAVING clauses, to solve the problem as n-way classification problem. The values of maximal condition numbers are chosen to cover all the training cases.

Finally, $c_{\texttt{IUEN}}$ represents the existence of one of INTERSECT, UNION, or EXCEPT, or NONE if no such clause exists. If the value of $c_{\texttt{IUEN}}$ is one of INTERSECT, UNION, or EXCEPT, the corresponding SPC is created, and the SELECT statement for that SPC is generated recursively.

FROM Clause. A list of $TBLs should be decided to predict the FROM clause. For each table i, $P_{\textbf{fromtbl}}(i|Q, D, P)$, the probability that table i is included in the FROM clause, is calculated as follows:

$$\begin{aligned} c_i &= concat(v_i^T, v^Q, v^D, v^P) \\ s_i &= W_2 \tanh(W_1 c_i) \\ P_{\textbf{fromtbl}}(i|Q, D, P) &= \sigma(s)_i \end{aligned} \tag{7}$$

where W_1, W_2 are trainable variables, $s = [s_1, \ldots, s_t] \in \mathbb{R}^t$ represents the scores for tables, and σ denotes the sigmoid function. From now on, we omit the notations Q, D, and P for the sake of simplicity.

Top n_t tables with the highest $P_{\textbf{fromtbl}}(i)$ values are chosen. We set an upper bound $N_t = 6$ on possible number of tables. The formula to get $P_{\textbf{\#tbl}}(n_t)$ for each possible n_t is:

$$\begin{aligned} v^{T'} &= \text{softmax}(s)V^T \\ P_{\textbf{\#tbl}}(n_t) &= \text{softmax}(full_2(v^{T'})) \end{aligned} \tag{8}$$

In the equation, $full_2$ means the application of two fully connected layers, and table score vector s is from Equation (7).

During the inference, the `$TBL`s are classified first, and `$COL`s for other clauses are chosen among the columns of the classified `$TBL`s.

`SELECT` Clause. The decoder first generates N_s conditions to predict the `SELECT` clause. Because each condition depends on different parts of Q, we calculate attended question vector for each condition:

$$\begin{aligned} A^Q_{\textbf{sel}} &= W_3 \tanh(V^Q W_1 + v^P W_2)^\intercal \\ V^Q_{\textbf{sel}} &= \text{softmax}(A^Q_{\textbf{sel}})V^Q \end{aligned} \tag{9}$$

While $W_1, W_2 \in \mathbb{R}^{d\times d}$, $W_3 \in \mathbb{R}^{N_s \times d}$ are trainable parameters, and $V^Q_{\textbf{sel}} \in \mathbb{R}^{N_s \times d}$ is the matrix of attended question vectors for N_s conditions. v^P is tiled to match the row of V^Q.

For m columns and N_s conditions, $P_{\textbf{sel\_col1}} \in \mathbb{R}^{N_s \times m}$, the probability matrix for each column to fill the slot `$COL`$_1$ of each condition, is calculated as follows:

$$\begin{aligned} A^C_{\textbf{sel}}[i] &= W_6 \tanh(V^Q_{\textbf{sel}}[i] W_4 + V^C W_5)^\intercal \\ P_{\textbf{sel\_col1}}[i] &= \text{softmax}(A^C_{\textbf{sel}}[i]) \end{aligned} \tag{10}$$

where $W_4, W_5 \in \mathbb{R}^{d\times d}$ and $W_6 \in \mathbb{R}^{1\times d}$ are trainable parameters. In this and following equations, notation $M[i]$ is used to represent the i-th row of matrix M.

The attended question vectors are then updated with selected column information to get the updated question vector $U^Q_{\textbf{sel\_col1}} \in \mathbb{R}^{N_s \times d}$:

$$\begin{aligned} U^C_{\textbf{sel\_col1}}[i] &= P_{\textbf{sel\_col1}}[i] V^C \\ U^Q_{\textbf{sel\_col1}}[i] &= W_7 hc(V^Q_{\textbf{sel}}[i], U^C_{\textbf{sel\_col1}}[i]) \end{aligned} \tag{11}$$

where W_7 is a trainable variable, and $hc(x, y)$ is defined in Equation (5). The probabilities for `$DIST`$_1$, `$AGG`$_1$, `$ARI`, and `$AGG` are calculated by applying a fully connected layer on $U^Q_{\textbf{sel\_col1}}[i]$.

Equation (10) is reused to calculate $P_{\textbf{sel\_col2}}$, with $V^Q_{\textbf{sel}}[i]$ replaced by $U^Q_{\textbf{sel\_col1}}[i]$; then $U^Q_{\textbf{sel\_col2}}$ is retrieved in the same way as Equation (11), and the probabilities of `$DIST`$_2$ and `$AGG`$_2$ are calculated in the same way as `$DIST`$_1$ and `$AGG`$_1$. Finally, the `$DIST` slot, `DISTINCT` marker for overall `SELECT` clause, is calculated by applying a fully connected layer on v^S.

Once all the slots are filled for N_s conditions, the decoder retrieves the first n_s conditions to predict the `SELECT` clause. This is possible because the CNN with Dense Connection used for question encoding (Yoon, Lee, and Lee 2018) captures relative position information. Due to the SQL consistency protocol of the Spider benchmark (Yu et al. 2018c), we expect that the conditions are ordered in the same way as they are presented in Q. For the data sets without such consistency protocol, the proposed slot-filling method could easily be changed to an LSTM-based model, as shown in Xu, Liu, and Song (2017).

ORDERBY Clause. The same network structure as a SELECT clause is applied. The only difference is the prediction of $ORD slot; this could be done by applying a fully connected layer on $U^Q_{\mathbf{ob\_col1}}$, which is the correspondence of $U^Q_{\mathbf{sel\_col1}}$.

GROUPBY Clause. The same network structure as a SELECT clause is applied. For the GROUPBY case, retrieving only the values of $P_{\mathbf{gb\_col1}}$ is enough to fill the necessary slots.

LIMIT Clause. A question does not explicitly contain the $NUM slot value for LIMIT clause in many cases, if the question is for the top-1 result (For example: "Show the name and the release year of the song by **the youngest** singer"). Thus, the LIMIT decoder first determines if the given Q requests for the top-1 result. If so, the decoder sets the $NUM value to 1; otherwise, it tries to find out the specific token for $NUM among the tokens of Q using pointer network (Vinyals, Fortunato, and Jaitly 2015). LIMIT top-1 probability $P_{\mathbf{limit\_top1}}$ is retrieved by applying a fully-connected layer on v^S. $P^Q_{\mathbf{limit\_num}}[i]$, the probability of i-th question token for $NUM slot value, is calculated as:

$$\begin{aligned} A^Q_{\mathbf{limit\_num}} &= W_3 \tanh(V^Q W_1 + v^P W_2)^\intercal \\ P^Q_{\mathbf{limit\_num}}[i] &= \mathrm{softmax}(A^Q_{\mathbf{limit\_num}})_i \end{aligned} \tag{12}$$

$W_1, W_2 \in \mathbb{R}^{d \times d}$, $W_3 \in \mathbb{R}^{1 \times d}$ are trainable parameters.

WHERE Clause. The same network structure as a SELECT clause is applied to get the attended question vectors $V^Q_{\mathbf{wh}} \in \mathbb{R}^{N_w \times d}$, and probabilities for COL_1$, COL_2$, $DIST$_1$, $DIST$_2$, AGG_1$, AGG_2$, and $ARI. A fully connected layer is applied on $U^Q_{\mathbf{wh\_col1}}$ to get the probabilities for $CONJ, $NOT, and $COND.

A fully connected layer is applied on $U^Q_{\mathbf{wh\_col1}}$ and $U^Q_{\mathbf{wh\_col2}}$ to determine if the condition value for each column is another nested SELECT statement or not. If the value is determined as a nested SELECT statement, the corresponding SPC is generated, and the SELECT statement for the SPC is predicted recursively. If not, the pointer network is used to get the start and end position of the value span from question tokens.

HAVING Clause. The same network structure as a WHERE clause is applied.

4. Two Input Manipulation Methods

In this section, we introduce two input manipulation methods to improve the performance of our proposed system further.

4.1 JOIN Table Filtering

In a FROM clause, some tables (and their columns) are not mentioned explicitly in the given question, but they are still required to make a "link" between other tables to

Table 3
An example SQL query with a link table.

Q: What are the papers of Liwen Xiong in 2015?
SQL:

```
SELECT DISTINCT t3.paperid
 FROM writes AS t2
  JOIN author AS t1 ON t2.authorid = t1.authorid
  JOIN paper AS t3 ON t2.paperid = t3.paperid
 WHERE t1.authorname = “Liwen Xiong”
  AND t3.year = 2015;
```

form a proper SQL query. One such example is given in Table 3. The table **writes** is not explicitly mentioned in *Q*, but it is used in the `JOIN` clause to link between tables **author** and **paper**. Those “link” tables are necessary to create the proper `SELECT` statement, but they work as noise in aligning question tokens and tables because the link tables do not have the corresponding tokens in *Q*.

To reduce the training noises, only the non-link tables are considered as the `$TBL` slot values of `FROM` clause during training. A table of `FROM` clause is considered a link table if (1) all the `$AGG` values of the `SELECT` clause are **none**, and (2) none of its columns appears in other clauses’ slots. During the inference, the link tables could easily be recovered by using the foreign key relations of the extracted tables. More precisely, the system uses a heuristic of finding the shortest joinable foreign key relation “path” between the extracted tables. Once a path is found, tables in the path are added as the `$TBL`s of the `FROM` clause.

The goal of this method is to distinguish the link tables from non-link tables during the training phase. SyntaxSQLNet (Yu et al. 2018b) first predicts all the columns, and then chooses `FROM` tables based on the classified columns. As noted in Yu et al. (2018b), the approach cannot handle **count** queries with additional `JOIN`s, for example, “`SELECT` **T2.name**, **count(*)** `FROM` **singer_in_concert** `AS` **T1** `JOIN` **singer** `AS` **T2** `ON` **T1.singer_id** = **T2.singer_id** `GROUP BY` **T2.singer_id**.” Its corresponding user question is “List singer names and number of concerts for each singer.” GNN (Bogin, Gardner, and Berant 2019) handles the problem by turning the database schema into a graph; foreign key links between nodes help the system to distinguish between two types of tables. IRNet (Guo et al. 2019) and RAT-SQL (Wang et al. 2020) have the separated schema linking processing module to explicitly link columns with question tokens.

4.2 Supplemented Column Names

We supplement the column names with their table names to distinguish between columns with the same name but belonging to different tables and representing different meanings. Table names are concatenated in front of their belonging column names to form supplemented column names (SCNs), but if the stemmed form of a table name is wholly included in the stemmed form of a column name, the table name is not concatenated. Table 4 shows SCN examples; the three columns with the same name *id* are distinguished with their SCNs. We can also expect the SCNs to align better with question tokens, since a SCN contains more information about what the column actually refers to.

The method aims to integrate tables with their columns. To achieve the goal, IRNet (Guo et al. 2019) and RAT-SQL (Wang et al. 2020) separately encode tables and columns,

Table 4
Examples of supplemented column names (SCNs).

Table	Column	SCN
tv channel	id	tv channel id
	series name	tv channel series name
tv series	id	tv series id
cartoon	id	cartoon id

and integrate the two embeddings on the network; GNN (Bogin, Gardner, and Berant 2019) represents database schema as a graph, generating links between tables and their columns, and directly processes the graph using graph neural network; EditSQL (Zhang et al. 2019) concatenates the table names with its column names, using a special character.

5. Related Work

Most recent works on the Text-to-SQL task used the encoder-decoder model. Those works could be classified into three main categories, based on their decoder outputs. Sequence-to-Sequence translation approaches generate SQL query tokens. Dong and Lapata (2016) introduced the hierarchical tree decoder to prevent the model from generating grammatically incorrect semantic representations of the input sentences. Zhong, Xiong, and Socher (2017) used policy-based reinforcement learning to deal with the unordered nature of `WHERE` conditions.

Grammar-based approaches generate a sequence of grammar rules and apply the generated rules sequentially to obtain the resultant SQL query. IRNet (Guo et al. 2019) defined a structural representation of an SQL query and a set of parse actions to handle the WikiSQL data set. IRNet defined the SemQL query, which is an abstraction of a SQL query in tree form. They also proposed a set of grammar rules to synthesize SemQL queries; synthesizing a SQL query from a SemQL tree structure is straightforward. RAT-SQL (Wang et al. 2020) improved the work of Guo et al. (2019) by proposing a relation-aware transformer to effectively encode relations between columns, tables, and question tokens. GNN (Bogin, Gardner, and Berant 2019) focused on the DB constraints selection problem during the grammar decoding process; they applied global reasoning between question words and database columns/tables. SLSQL (Lei et al. 2020) manually annotated link information between user questions and database columns to show the role of schema linking.

Sketch-based slot-filling approaches use a **sketch**, which aligns with the syntactic structure of a SQL query. A sketch should be defined generic enough to handle all SQL queries of interest. Once a sketch is defined, one can simply fill the slots of the sketch to obtain the resultant SQL query. SQLNet (Xu, Liu, and Song 2017) first introduced a sketch to handle the WikiSQL data set, along with attention-based slot-filling algorithms. The proposed sketch for WikiSQL is shown in Table 5. TypeSQL (Yu et al. 2018a) added category information such as named entity to better encode the input question. SQLova (Hwang et al. 2019) introduced BERT (Devlin et al. 2019) to encode the input question and database, and the encoded vectors were used to fill the slots of the sketch.

Table 5
Sketch for WikiSQL data set. `$COL` represents a column, and `$AGG` is one of {**none**, **max**, **min**, **count**, **sum**, **avg**}. `$COND` is one of the conditional operators { =, >, < }. `$VAL` is the value for `WHERE` condition.

CLAUSE	SKETCH
SELECT	$AGG $COL
WHERE	($COL $COND $VAL)*

Table 6
The sketch for a `SELECT` statement proposed by RCSQL (Lee 2019). `$COL` represents a column. `$AGG` is one of {**none**, **max**, **min**, **count**, **sum**, **avg**}, and `$COND` is one of the conditional operators {**between**, =, >, <, >=, <=, !=, **in**, **like**, **is**, **exists**}. `$ORD` is a binary value for keywords `ASC/DESC`, and `$CONJ` is one of conjunctions {`AND`, `OR`}. `$VAL` is the value for `WHERE/HAVING` condition; `$SEL` represents the slot for another `SELECT` statement.

CLAUSE	SKETCH
SELECT	($AGG $COL) +
ORDERBY	($AGG $COL) + $ORD
GROUPBY	($COL)*
LIMIT	$NUM
WHERE	($CONJ $COL $COND $VAL | $SEL) *
HAVING	($CONJ $AGG $COL $COND $VAL | $SEL) *
INTERSECT UNION EXCEPT	$SEL

X-SQL (He et al. 2018) aligned the contextual information with column tokens to better summarize each column. The sketch-based approaches for WikiSQL described here all used the sketch shown in Table 5, which is enough for the WikiSQL queries but oversimplified for general SQL queries, for example, those contained in the Spider benchmark.

The sketch-based approach on the more complex Spider benchmark showed relatively low performance compared to the grammar-based approaches so far. There are two major reasons: (1) It is hard to define a sketch for Spider queries since the allowed syntax of the Spider SQL queries is far more complicated than that of the WikiSQL queries. (2) Because the sketch-based approaches fill values for the predefined slots, the approaches have difficulties in predicting the nested queries. RCSQL (Lee 2019) tried to apply the sketch-based approach on the Spider data set; Table 6 shows the sketch proposed by Lee (2019). To predict a nested `SELECT` statement, RCSQL takes a temporal generated SQL query with a special token [`SUB_QUERY`] in the corresponding location as its input. For example, for Case 1 of Table 1, RCSQL gets a temporal generated query string "SELECT name FROM scientists EXCEPT [`SUB_QUERY`]" as its input to generate the nested statement S_2, along with the user question and database schema.

Our proposed approach has three improvements compared to RCSQL. First, our sketch in Table 2 is more "complete" in terms of expressiveness. For example, because the RCSQL sketch lacks $ARI elements, the RCSQL cannot generate queries with arithmetic operations between columns, for example, "SELECT **T1.name** FROM **accounts** AS **T1** JOIN **checking** AS **T2** ON **T1.custid** = **T2.custid** JOIN **savings** AS **T3** ON **T1.custid** = **T3.custid** ORDER BY **T2.balance + T3.balance** LIMIT 1." Second, while our proposed approach directly predicts for the tables in FROM clause, the RCSQL heuristically predicts the tables using the extracted columns for other clauses. The RCSQL approach cannot generate count queries with additional table JOINs, for example, "SELECT count(*) FROM **institution** AS **T1** JOIN **protein** AS **T2** ON **T1.institution_id** = **T2.institution_id** WHERE **T1.founded** > 1880 OR **T1.type** = 'Private'." Third, RCSQL fails to generate the nested SELECT statements when two or more statements are on the same *depth*, for example, S_2 and S_3 in Case 2 of Table 1. Because RCSQL generates one SELECT statement for an input, it expects only one special token for a query.

In this article, we propose a more completed sketch compared to the WikiSQL (Table 5) and RCSQL (Table 6) sketches for complex SELECT statements, along with the Statement Position Code (SPC) to handle the nested queries more efficiently. Although our proposed sketch is tuned using the Spider data set, the sketch is based on the generic SQL syntax and could be applied to other SQL generation tasks.

6. Experiment

6.1 Experiment Setup

Implementation. The proposed RYANSQL is implemented with Tensorflow (Abadi et al. 2015). Layernorm (Ba, Kiros, and Hinton 2016) and dropout (Srivastava et al. 2014) are applied between layers, with a dropout rate of 0.1. Exponential decay with decay rate 0.8 is applied to the learning rate for every three epochs. On each epoch, the trained classifier is evaluated against the validation data set, and the training stops when the exact match score for the validation data set is not improved for 20 consequent training epochs. Minibatch size is set to 16; learning rate is set to $4e^{-4}$. Loss is defined as the sum of all classification losses from the slot-filling decoder. The trained network has 22M parameters.

For pretrained language model–based input encoding, we downloaded the publicly available pretrained model of BERT, BERT-Large, Uncased (Whole Word Masking), and fine-tuned the model during training. The learning rate is set to $1e^{-5}$, and minibatch size is set to 4. The model with BERT has 445M parameters.

Data sets. The Spider data set (Yu et al. 2018c) is mainly used to evaluate our proposed system. We use the same data split as Yu et al. (2018c); 206 databases are split into 146 train, 20 dev, and 40 test. All questions for the same database are in the same split; there are 8,659 questions for train, 1,034 for dev, and 2,147 for test. The test set of Spider is not publicly available, so for testing our models are submitted to the data owner. For evaluation, we used exact matching accuracy, with the same definition as defined in Yu et al. (2018c).

6.2 Evaluation Results

Table 7 shows comparisons of the proposed system with several state-of-the-art systems; evaluation scores for dev and test data sets are retrieved from the Spider

Table 7
Evaluation results of the proposed systems and other state-of-the-art systems.

System	Dev	Test
Without pretrained language models		
GrammarSQL (Lin et al. 2019)	34.8%	33.8%
EditSQL (Zhang et al. 2019)	36.4%	32.9%
IRNet (Guo et al. 2019)	53.3%	46.7%
RATSQL v2 (Wang et al. 2020)	62.7%	57.2%
RYANSQL (Ours)	43.4%	–
With pretrained language models		
RCSQL (Lee 2019)	28.5%	24.3%
EditSQL + BERT	57.6%	53.4%
IRNet + BERT	61.9%	54.7%
IRNet v2 + BERT	63.9%	55.0%
SLSQL + BERT (Lei et al. 2020)	60.8%	55.7%
RYANSQL + BERT (Ours)	66.6%	58.2%
RYANSQL v2 + BERT (Ours)	**70.6%**	60.6%
With DB content		
Global-GNN (Bogin, Gardner, and Berant 2019)	52.7%	47.4%
IRNet++ + XLNet	65.5%	60.1%
RATSQL v3 + BERT	69.7%	**65.6**%

leaderboard.[1] The proposed system is compared with grammar-based systems GrammarSQL (Lin et al. 2019), Global-GNN (Bogin, Gardner, and Berant 2019), IRNet (Guo et al. 2019), and RATSQL (Wang et al. 2020). Also, we compared the proposed system with RCSQL (Lee 2019), which so far showed the best performance on the Spider data set using a sketch-based slot-filling approach.

Evaluation results are presented in three different groups, based on the use of pretrained language models and database content. Although the use of database content (i.e., cell values) could greatly improve the performance of a Text-to-SQL system (as shown in Wang et al. 2018; Hwang et al. 2019; He et al. 2018), a Text-to-SQL system could rarely have access to database content in real world applications due to various reasons such as personal privacy, business secrets, or legal issues. Because the use of database content improves the system performance but decreases the system availability, we put models using database content in a separated group.

For RYANSQL v2, we trained two networks called **table network** and **slot network**, with the same network architectures as the proposed RYANSQL. The table network is trained to maximize the `$TBL` classification accuracy on dev set; the slot network is trained to maximize the exact match accuracy on dev set as RYANSQL does, but the `$TBL` classification results are fetched from the table network (which is fixed during the training of the slot network). During the inference, the model first classifies `$TBL`s using the table network, and fills other slots using the slot network.

[1] `https://yale-lily.github.io/spider`, as of April 2020.

Table 8
Exact matching accuracy of the proposed system and other state-of-the-art systems for each hardness level. **Med.** means medium hardness.

Approaches	**Easy**	**Med.**	**Hard**	**Extra**	**ALL**
On *dev* set					
RCSQL	53.2%	27.0%	20.1%	6.5%	28.8%
IRNet[2]	70.4%	55.0%	46.6%	30.6%	53.3%
RATSQL v2 *(with DB content)*	80.4%	63.9%	55.7%	40.6%	62.7%
RATSQL v3 + BERT *(with DB content)*	86.4%	73.6%	62.1%	42.9%	69.7%
RYANSQL (Ours)	69.2%	43.0%	28.2%	22.4%	43.4%
RYANSQL + BERT (Ours)	86.0%	70.5%	54.6%	40.6%	66.6%
On *test* set					
IRNet	70.1%	49.2%	39.5%	19.1%	46.7%
IRNet + BERT	77.2%	58.7%	48.1%	25.3%	54.7%
RATSQL v2 *(with DB content)*	74.8%	60.7%	53.6%	31.5%	57.2%
RATSQL v3 + BERT *(with DB content)*	83.0%	71.3%	58.3%	38.4%	65.6%
RYANSQL + BERT (Ours)	81.2%	62.1%	51.9%	28.0%	58.2%

As can be observed from the table, the proposed system RYANSQL improves the previous sketch-based slot-filling system RCSQL by a large margin of 15% on the dev set. Note that the RCSQL fine-tuned another well-known pretrained language model ELMo (Peters et al. 2018). With the use of BERT, among the systems without database content, the proposed systems RYANSQL + BERT and RYANSQL v2 + BERT outperform the previous state-of-the-art by 2.5% and 4.9%, respectively, on the hidden test data set, in terms of exact matching accuracy. The proposed system still shows competitive results compared to the systems using database content; RATSQL v3 + BERT outperforms the proposed system by better aligning user questions and database schemas using database content.

Table 8 compares the exact matching accuracies of the proposed systems and other state-of-the-art systems for each hardness level. The proposed RYANSQL + BERT outperforms the previous sketch-based approach RCSQL in every hardness level on dev set. Additionally, the proposed RYANSQL + BERT showed relatively poor performance for the test set at the **Extra** hardness level, compared to RATSQL v3 + BERT. This suggests that much test data at the **Extra** hardness level require database content to answer, since the two systems showed comparable results for the **Extra** hardness dev set.

Next, ablation studies are conducted on the proposed methods to clarify the contribution of each feature. The results are presented in Table 9. It turns out that the use of SPC greatly improves the performances for **Hard** and **Extra** hardness levels. The result shows that the SPC plays an important role in generating the nested SQL queries. The SPC also slightly increases the performance for **Easy** and **Medium** hardness levels.

[2] For IRNet + BERT, we downloaded the source code and trained the model from authors' homepage (`https://github.com/microsoft/IRNet`), but we were not able to reproduce the authors' suggested dev set exact matching accuracy.

Table 9
Ablation study results of the proposed models for each hardness level on *dev* set. **SPC**, **SCN**, and **JTF** represent the use of Statement Position Code, supplemented column names, and JOIN table filtering, respectively.

Approaches	SPC	SCN	JTF	Easy	Med.	Hard	Extra	ALL
RYANSQL	O	O	O	69.2%	43.0%	28.2%	22.4%	43.4%
	O	O	X	68.0%	40.2%	27.6%	19.4%	41.4%
	O	X	O	62.0%	38.2%	24.1%	14.7%	37.7%
	O	X	X	65.2%	37.7%	29.9%	18.8%	39.9%
	X	O	O	63.2%	39.5%	19.0%	16.5%	38.0%
	X	O	X	68.4%	41.6%	18.4%	14.7%	39.7%
	X	X	O	63.2%	39.3%	14.9%	16.5%	37.2%
	X	X	X	60.0%	38.2%	16.7%	11.8%	35.5%
RYANSQL + BERT	O	O	O	86.0%	70.5%	54.6%	40.6%	66.6%
	O	O	X	86.8%	66.1%	46.6%	42.4%	63.9%
	O	X	O	76.4%	58.2%	46.6%	30.6%	56.1%
	O	X	X	78.0%	63.4%	46.0%	28.8%	58.3%
	X	O	O	85.6%	66.6%	27.0%	22.4%	57.3%
	X	O	X	83.6%	68.4%	25.9%	26.5%	58.0%
	X	X	O	78.4%	60.2%	21.3%	24.7%	52.2%
	X	X	X	77.2%	60.5%	23.6%	25.9%	52.6%

This is because the SPC helps the model to distinguish between each nested SELECT statement, thus removing noise on aligning question tokens and columns.

The use of SCN moderately improves the accuracies for all hardness levels. This is expected, since SCN helps a database column to better align with question tokens by supplementing the column name with its table information.

The JOIN table filtering (JTF) increases performance only when the other two features SPC and SCN are used together. Analysis shows that for some cases, the link tables removed by JTF actually have their corresponding question tokens. One example is the SQL query "SELECT **T3.amenity_name** FROM **dorm** AS **T1** JOIN **has_amenity** AS **T2** ON **T1.dormid** = **T2.dormid** JOIN **dorm_amenity** AS **T3** ON **T2.amenid** = **T3.amenid** WHERE **T1.dorm_name** = 'Smith Hall' " for question "Find the name of amenities Smith Hall dorm have." Table **has_amenity** is considered as a link table, but there exist corresponding clues in the question. Removing the table from $TBL list according to the JTF feature would introduce alignment noise during training. But the evaluation result also shows that, by better aligning question and database schema using the other two features SPC and SCN, the model can recover from the alignment noise introduced by JTF, improving the overall system performance.

Proposed models are also evaluated without all three features SPC, SCN, and JTF to separately see the contribution of our newly proposed sketch. Without the three features, RYANSQL shows 35.5% accuracy on dev set, which is a 6.7% improvement compared to another sketch-based slot-filling model RCSQL; RYANSQL + BERT shows 52.6% dev set accuracy.

Effect of the pretrained language model. There exists a huge performance gap of 23.2% on dev set between RYANSQL and RYANSQL + BERT. SQL component matching F1 scores for the two models are shown in Table 10 to figure out the reason. For the

Table 10
SQL Component matching F1 scores of **RYANSQL** and **RYANSQL + BERT** on *dev* set.

Approaches	SELECT	WHERE	GROUP	ORDER	**keywords**	**ALL**
RYANSQL	69.4%	47.4%	67.5%	73.9%	82.3%	**43.4%**
RYANSQL + BERT	88.2%	74.4%	78.8%	83.3%	88.5%	**66.6%**

Table 11
Evaluations with different pretrained language models on *dev* set.

System	**Dev**
RYANSQL	43.4%
RYANSQL + BERT-base	51.4%
RYANSQL + BERT-large	66.6%
RYANSQL + RoBERTa	65.7%
RYANSQL + ELECTRA	63.6%

item **keyword** (which measures the existence of SQL predefined keywords such as `SELECT` or `GROUP BY`), the performance gap between the two models is 6.2%, which is relatively small compared to the overall performance gap of 23.2%. Meanwhile, the performance gaps on clause components such as `WHERE` are similar to or larger than the overall performance gap. These evaluation results suggest that the use of a pretrained language model mainly improves the column classification performance, rather than base structures classification accuracy of a SQL query.

Next, a series of experiments is conducted to see if additional performance improvements could be gained by applying different pretrained language models. Table 11 shows the evaluation results with four different pretrained language models, namely, BERT-base, BERT-large, RoBERTa (Liu et al. 2019), and ELECTRA (Clark et al. 2020). Although RoBERTa and ELECTRA are generally known to perform better than BERT, the evaluation results showed no performance improvement.

Generality of SCN and JTF. The two proposed input manipulation methods SCN and JTF are applied on IRNet (Guo et al. 2019) to see their generalities. We downloaded the source code from the author's homepage,[3] and trained to obtain the dev set accuracy. Evaluation results are shown in Table 12. The performance improvements due to the two input manipulation methods were almost ignorable; because IRNet has the separated schema linking preprocessing module, whose purpose is to link columns with question tokens, the role of SCN and JTF are greatly reduced.

6.3 Evaluation on Different Data Sets

We conducted experiments on WikiSQL (Zhong, Xiong, and Socher 2017) and CSpider (Min, Shi, and Zhang 2019) data sets to test the generalization capability of the proposed model to new data sets. Table 13 shows the comparison between the proposed RYAN-

[3] `https://github.com/microsoft/IRNet`.

Table 12
Evaluations of IRNet with two input manipulation methods on *dev* set.

System	SCN	JTF	Dev
IRNET	O	O	52.3%
	O	X	52.9%
	X	O	52.3%
	X	X	52.2%[4]

Table 13
Evaluation results on WikiSQL benchmark with other state-of-the-art systems.

System	Dev LF	Dev X	Test LF	Test X
SQLova (Hwang et al. 2019)	81.6 %	87.2 %	80.7 %	86.2 %
X-SQL (He et al. 2018)	83.8 %	89.5 %	83.3 %	88.7 %
Guo and Gao (2019)	84.3 %	90.3 %	83.7 %	89.2 %
HydraNet (Lyu et al. 2020)	83.6 %	89.1 %	83.8 %	89.2 %
RYANSQL + BERT	81.6 %	87.7 %	81.3 %	87.0 %

SQL + BERT and other WikiSQL state-of-the-art systems. Only the systems without execution-guided decoding (EGD) (Wang et al. 2018) are compared, since EGD makes use of the database content. As can be observed from the table, the proposed RYANSQL + BERT showed comparable results to other WikiSQL state-of-the-art systems.

Next, we evaluated the proposed models on the CSpider data set. CSpider (Min, Shi, and Zhang 2019) is a Chinese-translated version of the Spider benchmark. Only the question of the Spider data set is translated; database table names and column names remain in English. Evaluation on the CSpider data set will show whether the proposed model could be applied on the different languages, even when the question language and database schema language are different. To handle the case, we used multilingual BERT, which has the same network architecture with BERT-base but is trained using a multilingual corpus. Table 14 shows the comparisons between the proposed system and other state-of-the-art systems on the leaderboard. Compared to the exact matching accuracy 51.4% of RYANSQL + BERT-base on Spider data set, the multilingual version shows 10% lower accuracy on dev set, but still shows comparable results to other state-of-the-art systems that are designed for CSpider data set. Our proposed system showed 34.7% test accuracy on the test set, and ranked 2nd place on the leaderboard.

[4] We re-trained the IRNet model using the authors' source code with the Spider data set. We were not able to obtain the authors' presented 53.3% accuracy on the dev set, and it turns out that the preprocessed Spider data sets on the authors' homepage and generated from the source code script are different. Since we need to preprocess the data using the source code to apply the input manipulation methods, we presented the dev set accuracy of our re-trained IRNet model, not the one presented in the authors' paper.

Table 14
Evaluation results on CSpider data set[5] with other state-of-the-art systems.

System	Dev	Test
SyntaxSQLNet (Yu et al. 2018b)	16.4%	13.3%
CN-SQL (Anonymous)	22.9%	18.8%
DG-SQL (Anonymous)	35.5%	26.8%
XL-SQL (Anonymous)	54.9%	47.8%
RYANSQL + Multilingual BERT (Ours)	41.3%	34.7%

Table 15
Exact matching accuracy of the proposed system on the Spider dev set, with the different ranges of overlap scores.

Overlap Score	Accuracy	Examples
$0.0 \leq O(Q,S) \leq 0.2$	32.5%	40
$0.2 < O(Q,S) \leq 0.4$	47.3%	74
$0.4 < O(Q,S) \leq 0.6$	46.7%	182
$0.6 < O(Q,S) \leq 0.8$	67.0%	282
$0.8 < O(Q,S) \leq 1.0$	80.5%	456
Total	66.6%	1,034

6.4 Error Analysis

We analyzed 345 failed examples of the RYANSQL + BERT on the development set. We were able to categorize 195 of those examples according to failure types.

The most common cause of failure is column selection failure; 68 out of 195 cases (34.9%) suffered from the error. In many of these cases, the correct column name is not mentioned in a question; for example, for the question ″What is the airport name for airport ‘AKO’?”, the decoder chooses column **AirportName** instead of **AirportCode** as its `WHERE` clause condition column. As mentioned in Yavuz et al. (2018), cell value examples for each column will be helpful to solve this problem.

The second frequent error is table number classification error; 49 out of 195 cases (25.2%) belong to the category. The decoder occasionally chooses too many tables for the `FROM` clause, resulting in unnecessary table `JOIN`s. Similarly, 22 out of 195 cases (11.3%) were due to condition number classification error. Those errors could be handled by observing and updating the extracted slot values as a whole; for example, for a user question “List the maximum weight and type for each type of pet.” the system generates SQL query “`SELECT` **PetType**, **max**(**weight**), **weight** `FROM` **Pets** `GROUP BY` **PetType**.” If the system could observe the extracted slot values as a whole, it would figure out that extracting **weight** and **max(weight)** together for `SELECT` clause is unlikely. Our future work will mainly focus on solving this issue.

[5] `https://taolusi.github.io/CSpider-explorer/`, as of July 2020.

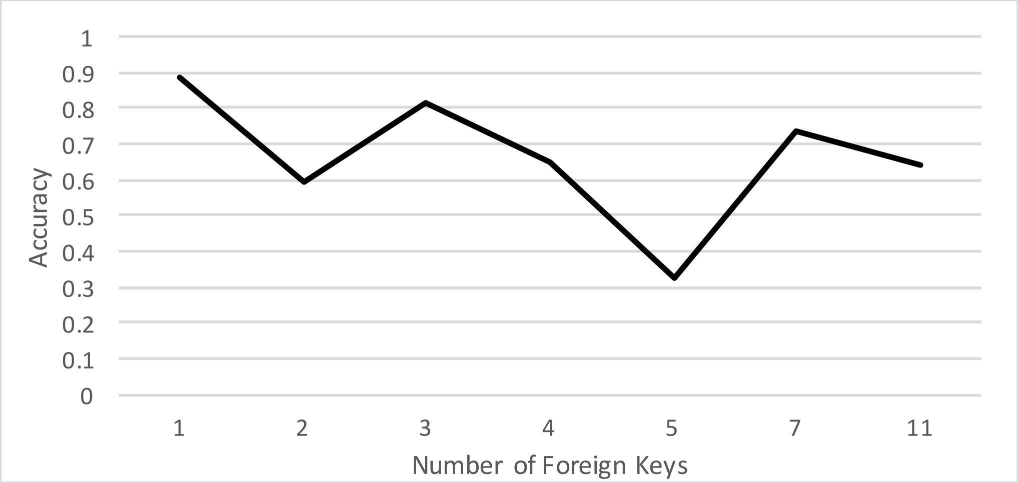


Figure 3
Relation between the number of foreign key relations in the database, and exact matching accuracy of the queries on dev set.

The remaining 150 errors were hard to be classified into one category, and some of them were due to different representations of the same meaning, for example: "`SELECT` **max(age)** `FROM` **Dogs**" vs. "`SELECT` **age** `FROM` **Dogs** `ORDER BY` **age** `DESC` `LIMIT` 1."

Next, we tried to see if the proposed model could handle user questions in which words are different from database column and table names. We define an overlap score $O(Q,S) = \frac{size(w(Q) \cap w(S))}{size(w(S))}$ between a user question Q and its SQL translation S. In the equation, $w(Q)$ is the set of stemmed words in Q, and $w(S)$ is the set of stemmed words from column names and table names used in S. Intuitively, the score measures how much overlap exists between the column/table names of SQL query S and user question Q.

Overlap scores are calculated for question-SQL query pairs in the Spider dev set. The data set is divided into five categories based on the calculated overlap scores; Table 15 shows exact matching accuracies of the proposed RYANSQL for those categories. As can be seen from the table, the proposed system shows relatively low performance on the examples with low overlap scores. This suggests one limitation of the proposed system: Even with the aid of pre-trained language models, the system frequently fails to link between question tokens and database schema when their words are different. Better alignment methods between question tokens and database schema should be studied as a future work to further improve the system performance.

Another limitation of the proposed model is that the model does not use foreign keys during encoding; foreign keys are used only for `JOIN` table filtering. We analyzed the correlation between the number of foreign keys and exact matching accuracies in Figure 3, to figure out the effect of such limitation. The number of foreign keys and exact matching accuracy shows weak negative correlation, with Pearson correlation coefficient $\rho = -0.22$. Based on the analysis result, in future work we will try to integrate the foreign keys into the encoding process, for example, by using the relation aware transformer proposed in Wang et al. (2020), to improve the proposed model further.

7. Conclusion

In this article, we proposed a sketch-based slot-filling algorithm for complex, cross-domain Text-to-SQL problems. A detailed sketch for complex SELECT statement prediction is proposed, along with the Statement Position Code to handle nested queries. Two simple but effective input manipulation methods are additionally proposed to enhance the overall system performance further. The system achieved 3rd place among all systems and 1st place among the systems not using database content, on the challenging Spider benchmark data set.

Based on the error analysis results, as a next step of the research we will focus on globally updating the extracted slots by considering the slot prediction values as a whole. The analysis results also show the need to encode the relation structures of the database schema, for example, foreign keys, to improve the performance. We will also work on a method to effectively use the database content instead of using only the database schema, to further improve the system performance for the cases when database content is available.

References

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Man, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, PeteWarden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from `tensorflow.org`.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *Computing Research Repository*, arXiv:1607.06450.

Beck, Thorsten, Asli Demirguc-Kunt, and Ross Levine. 2000. A new database on the structure and development of the financial sector. *The World Bank Economic Review*, 14(3):597–605. `https://doi.org/10.1093/wber/14.3.597`

Bogin, Ben, Matt Gardner, and Jonathan Berant. 2019. Global reasoning over database structures for text-to-SQL parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3650–3655. Hong Kong.

Clark, Kevin, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *Computing Research Repository*, arXiv:2003.10555.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 4171–4186. Minneapolis, MN.

Dong, Li and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 33–43. Berlin. `https://doi.org/10.18653/v1/P16-1004`

Dong, Li 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. Melbourne. `https://doi.org/10.18653/v1/P18-1068`

Guo, Jiaqi, 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. *Computing Research Repository*, arXiv:1905.082057.

Guo, Tong and Huilin Gao. 2019. Content enhanced BERT-based text-to-SQL generation. *Computing Research Repository*, arXiv:1910.07179.

He, Pengcheng, Yi Mao, Kaushik Chakrabarti, and Weizhu Chen. 2018. X Reinforce schema representation with context. *Computing Research Repository*, arXiv:1808.073837.

Hillestad, Richard, James Bigelow, Anthony Bower, Federico Girosi, Robin Meili, Richard Scoville, and Roger Taylor. 2005. Can electronic medical record systems transform health care? Potential health benefits, savings, and costs. *Health Affairs*. 24(5):1103–1117. https://doi.org/10.1377/hlthaff.24.5.1103, PubMed: 16162551

Hu, Minghao, Yuxing Peng, Zhen Huang, Xipeng Qiu, Furu Wei, and Ming Zhou. 2018. Reinforced mnemonic reader for machine reading comprehension. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4099–4106. Stockholm.

Hwang, Wonseok, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on WikiSQL with table-aware word contextualization. *Computing Research Repository*, arXiv:1902.01069.

Lee, Dongjun. 2019. Clause-wise and recursive decoding for complex and cross-domain text-to-SQL generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6047–6053, Hong Kong. https://doi.org/10.18653/v1/D19-1624

Lei, Wenqiang, Weixin Wang, Zhixin Ma, Tian Gan, Wei Lu, Min-Yen Kan, and Tat-Seng Chua. 2020. Re-examining the role of schema linking in text-to-SQL. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6943–6954. Online.

Lin, Kevin, Ben Bogin, Mark Neumann, Jonathan Berant, and Matt Gardner. 2019. Grammar-based neural text-to-SQL generation. *Computing Research Repository*, arXiv:1905.13326.

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *Computing Research Repository*. arXiv:1907.11692.

Lyu, Qin, Kaushik Chakrabarti, Shobhit Hathi, Souvik Kundu, Jianwen Zhang, and Zheng Chen. 2020. Hybrid ranking network for text-to-SQL, Microsoft Dynamics 365 AI.

Min, Qingkai, Yuefeng Shi, and Yue Zhang. 2019. A pilot study for Chinese SQL semantic parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3652–3658. Hong Kong. https://doi.org/10.18653/v1/D19-1377

Mou, Lili, Rui Men, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. 2016. Natural language inference by tree-based convolution and heuristic matching. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 130–136. Berlin. https://doi.org/10.18653/v1/P16-2022

Ngai, Eric WT, Li Xiu, and Dorothy C. K. Chau. 2009. Application of data mining techniques in customer relationship management: A literature review and classification. *Expert Systems with Applications*, 36(2):2592–2602. https://doi.org/10.1016/j.eswa.2008.02.021

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha. https://doi.org/10.3115/v1/D14-1162

Peters, Matthew, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 2227–2237. Minneapolis, MN. https://doi.org/10.18653/v1/N18-1202

Price, P. J. 1990. Evaluation of spoken language systems: The ATIS domain. In *HLT '90: Proceedings of the Workshop on Speech and Natural Language*, pages 91–95. Hidden Valley, PA. https://doi.org/10.3115/116580.116612

Shi, Tianze, Kedar Tatwawadi, Kaushik Chakrabarti, Yi Mao, Oleksandr Polozov, and Weizhu Chen. 2018. IncSQL: Training incremental text-to-SQL parsers with non-deterministic oracles. *Computing Research Repository*. arXiv:1809.05054.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Srivastava, Rupesh K., Klaus Greff, and Jrgen Schmidhuber. 2015. Training very deep networks. *Advances in Neural Information Processing Systems*, pages 2377–2385.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008.

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

Wang, Bailin, 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*, pages 7567–7578. Online. `https://doi.org/10.18653/v1/2020.acl-main.677`

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

Xu, Xiaojun, Chang Liu, and Dawn Song. 2017. SQLnet: Generating structured queries from natural language without reinforcement learning. *Computing Research Repository*, arXiv:1711.04436.

Yavuz, Semih, Izzeddin Gur, Yu Su, and Xifeng Yan. 2018. What it takes to achieve 100% condition accuracy on WikiSQL. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1702–1711. Brussels. `https://doi.org/10.18653/v1/D18-1197`

Yoon, Deunsol, Dongbok Lee, and Sangkeun Lee. 2018. Dynamic self-attention: Computing attention over words dynamically for sentence embedding. *Computing Research Repository*, arXiv:1808.073837.

Yu, Tao, 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. New Orleans, LA.

Yu, Tao, 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. Brussels. `https://doi.org/10.18653/v1/D18-1193`

Yu, Tao, 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 data set 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. Brussels. `https://doi.org/10.18653/v1/D18-1425`

Zelle, John Marvin and Raymond Joseph Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference On Artificial Intelligence*, volume 2, pages 1050–1055. Portland, OR.

Zhang, Rui, Tao Yu, Heyang Er, Sungrok Shim, Eric Xue, Xi Victoria Lin, Tianze Shi, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019. Editing-based SQL query generation for cross-domain context-dependent questions. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5341–5352. Hong Kong.

Zhong, Victor, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *Computing Research Repository*, arXiv:1709.00103.