Transition-based Graph Generation For Text-to-SQL Task

Anonymous ACL submission

Abstract

In this paper, we present a neural semantic parsing approach – transition-based approach for mapping natural language questions to SQL queries. We represent a SQL query as a semantic graph, in which semantic units from the knowledge base are as nodes, keywords of SQL language and operators are as edges. Then we transform the text-to-SQL generation process to sequence-to-graph process using transition methods and Seq2Seq models, which can simultaneously leverage the advantages of semantic graph representation and the strong prediction ability of Seq2Seq models. Additionally, in order to improve the generation of graph's nodes which refers to matching between questions and the knowledge base, we propose a matching network leveraging multiple levels of granularity and semantic structure of the knowledge base. Experimental results on wikiSQL dataset show that our approach consistently improves performance, achieving competitive results despite the use of relatively simple decoder.

1 Introduction

Semantic parsing maps natural language utterances to executable programs. The programs could be a range of representations such as logic forms (Zelle and Mooney, 1996; Wong and Mooney, 2007; Zettlemoyer and Collins, 2012; Liang et al., 2013; Ling et al., 2016; Iyer et al., 2017). In this work, we regard SQL as the program language, which could be executed on a table or a relational database to obtain an outcome. Datasets are the main driver of progress for statistical approaches in semantic parsing (Liang, 2016). Recently, Zhong et al. (2017) release WikiSQL, the largest hand-annotated semantic parsing dataset with simple SQL queries and dataset schemas. Yu et al. (2018) release Spider, a large scale, complex and cross-domain semantic parsing dataset, which

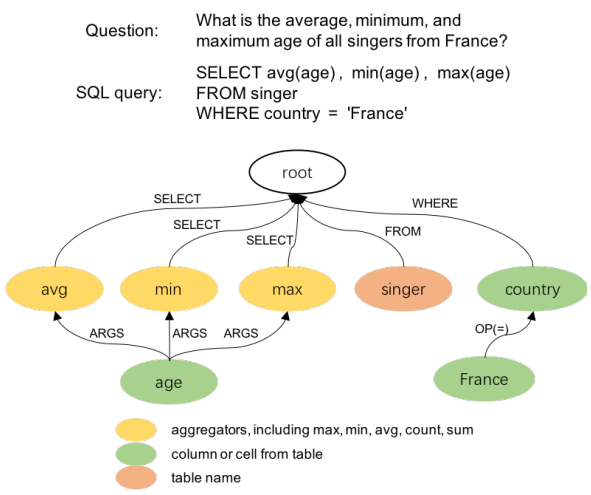

Figure 1: An example of semantic graph for SQL query

is more complex in SQL queries. In recent years, the successful applications of recurrent neural networks to a variety of NLP tasks (Bahdanau et al., 2014; Vinyals et al., 2015) have provided strong impetus to treat semantic parsing as a sequence-to-sequence problem (Jia and Liang, 2016; Dong and Lapata, 2016; Ling et al., 2016). The fact that meaning representations are typically structured objects has prompted efforts to develop neural architectures which explicitly account for their structures. Examples include tree decoders (Dong and Lapata, 2016; Alvarez-Melis and Jaakkola, 2016), decoders constrained by a grammar model (Xiao et al., 2016; Yin and Neubig, 2017; Krishnamurthy et al., 2017), modular decoders which use syntax to dynamically compose various sub-models (Rabinovich et al., 2017), or decoders decomposed into two stages (Dong and Lapata, 2018). These works treat semantic parsing as sequence to structured sequence with tailored neural decoders, in order to satisfy syntax and structure of logical form.

In this work, we propose to represent SQL query as a semantic graph (for example in Figure

1) and transform sequence-to-sequence process to sequence-to-graph process with a simple decoder. We argue that there are at least three advantages to the proposed approach. Firstly, semantic graph has a tight-coupling with knowledge bases (Yih et al., 2015) and share many commonalities with syntactic structures (Reddy et al., 2014), therefore both the structure and semantic constraints from knowledge bases can be exploited during parsing. Secondly, semantic graph can be generated with relatively mature transition methods (Wang et al., 2015; Konstas et al., 2017; Peng et al., 2018b), the syntax of SQL language can be satisfied with the simplest decoder. Thirdly, the model can explicitly share knowledge of local structures for the examples that have the same coarse sketch (i.e., subgraphs, with same edge and types of node pair, pairs of (*root, min*) and (*root, max*) in Figure 1), even though their actual meaning representations are different.

In our framework, the generation of graph is transformed into the generation of action-sequence using transition methods based on Seq2Seq models, which can simultaneously leverage the advantages of semantic graph representation and the strong prediction ability of Seq2Seq models. To achieve this goal, we select the transition system called cache transition system (Gildea et al., 2018) and design an action set which can encode the generation process of semantic graph (including the actions for node generation such as *NodeGen*, the actions for edge generation such as *ArcGen*.). Then, we use the Bi-LSTM to encode input information and design an RNN model to generate the action sequence. Additionally, in order to improve the generation of nodes, we propose a matching network leveraging multiple levels of granularity and the structure of knowledge base. In summary, we mainly make the following contributions:

- We are the first to represent SQL query as a graph, and design rules for transformations between SQL queries and graphs. We utilize Seq2Seq models with transition methods to generate the graph, which can synthesize the advantages of graph representation and the prediction ability of Seq2Seq models.
- We propose a matching network for node generation, based on multiple levels of granularity and the structure of knowledge base, which refers to column-cell relation in the Table database.
- Our architecture is conducted on wikiSQL dataset and achieves competitive performance compared with previous systems, despite employing the simplest sequence decoder.

2 Related Work

The text-to-SQL task attracts much interests from both academia (Yaghmazadeh et al., 2017a) and industry (Zhong et al., 2017) and has been studied for decades (Warren and Pereira, 1982; Popescu et al., 2003; Li et al., 2006; Giordani and Moschitti, 2012; Wang et al., 2017). More recently, neural Seq2Seq models have been applied to semantic parsing with promising results (Zhong et al., 2017; Xu et al., 2017; Dong and Lapata, 2018; Sun et al., 2018), referring semantic parser as the natural-language-to-SQL problem. Yaghmazadeh et al. (2017b) build a semantic parser on the top of SEMPRE (Pasupat and Liang, 2015) to get a SQL sketch, which only has the SQL shape and will be subsequently completed based on the table content. Iyer et al. (2017) maps utterances to SQL queries through sequence-to-sequence learning, user feedbacks are incorporated to reduce the number of queries to be labeled. Zhong et al. (2017) develops an augmented pointer network, which is further improved with reinforcement learning for SQL sequence prediction. Xu et al. (2017) adopts sequence-to-set model to predict columns of WHERE clause, and uses attentional model to predict the slots in WHERE clause. Dong and Lapata (2018) decomposes the decoding into two stages, the first decoder focuses on predicting a rough sketch of the meaning representation and the second decoder fills in missing details by conditioning on the input and the sketch. Sun et al. (2018) develops pointer network with multi-channels, and utilizes column-cell relation to improve the generation of WHERE clause.

3 Problem Formulation

The text-to-SQL task takes a question q and a knowledge base b consisting of k tables every of which consist of n columns and $n \times m$ cells as the input, and outputs a SQL query y.

As shown in Figure 2, we convert SQL queries to action sequences using semantic graphs as an intermediate representation. The tasks and challenges are as follows:

- Design measures for transformations between

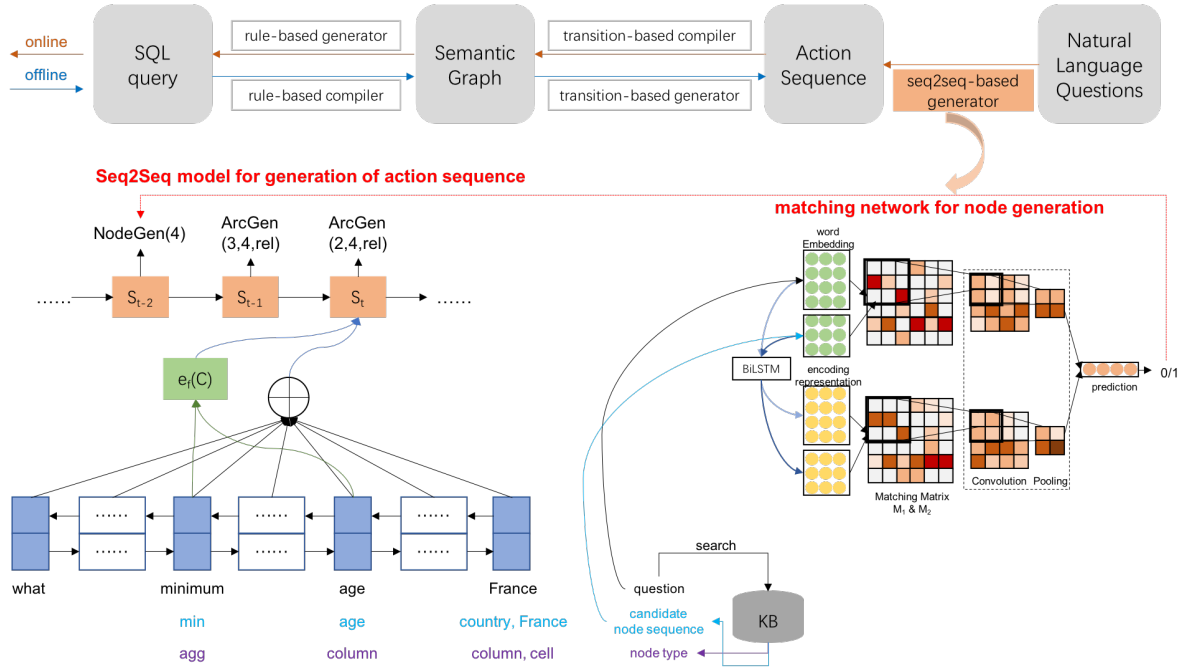

Figure 2: Overview of our method. We convert SQL queries to action sequences using semantic graphs as an intermediate representation. The transformation from the question to the action sequence with Seq2Seq models (Section 5.2), for action of *NodeGen*, a matching network is used (Section 5.3).

SQL queries and graphs, and ensure that the transformations are in the deterministic ways (Section 4.2).

- For the graph generation, we utilize transition methods which have ensured that the action sequence can lead to the gold graph (Nivre, 2008). We design an action set to encode the graph generation (Section 5.1).
- Design the model to generate the action sequence with a natural language question (Section 5.2).

4 Graph Representation for SQL

4.1 Graph Definition

The SQL query consists of semantic units from the knowledge base, aggregators, SQL keywords, and values from the question (see Figure 1). We define the units of the knowledge base, aggregators and value of *limit* as the nodes, all of which are obtained by matching between questions and KBs[1]; SQL keywords are defined as edges, which are obtained during the generation of action sequence. The sets of nodes and edges of graph are shown in Table 1.

4.2 Transformations Between SQL Queries and Graphs

SQL keywords construct SQL queries with some patterns, each keyword leads to a sub-SQL with its own pattern. The transformations between SQL queries and graphs are transformed into mappings between sub-SQLs and sub-graphs. We consider the following three aspects to ensure that transformations are in the deterministic way (see Appendix A.2 for detail):

- **The coverage of sub-SQLs** We collect all patterns of sub-SQLs according to SQL grammars, and ensure that they can be transformed into sub-graphs in the deterministic way, see examples shown in Table 2. Using a depth-first-search algorithm, all SQL queries can be transformed into graphs in the deterministic way.
- **The consistency of transformation from graphs to SQL queries.** We design the parent-child protocol for coordinative constituents and the order for sub-SQLs to ensure that the graph can be transformed into the SQL query in the deterministic way.

[1]Aggregators and value of *limit* are also recognized as the KB, utilizing their full names and synonyms to represent them, such as the full name of *avg* is *average*, *top* is the synonym of value of *limit*, the mentioned KB in this paper includes aggregators and value of *limit* by default

–	group tag	explanation
nodes	*agg*	aggregators, including min, max, count, sum, avg
	col, *cell*, *tab*	from KB, including table name, column name, content of a (row, column)
	limv	**digit value**, following the ORDER BY, limiting the number of results
	root	virtual node, representing the start of clauses, including root, t_root
edges	*OP*	operators, including >, <, =, !=, >=, <=, LIKE, IN, NOT IN, BETWEEN
	KW	basic keywords, including SELECT, WHERE, GROUP BY, HAVING, ORDER BY DESC/ASC
	LIMIT	limiting the number of results, following the ORDER BY, LIMIT
	CON_REL	relations between conditions in WHERE clause, including AND, OR, NOT
	SUB_REL	relations between sub-SQLs, including INTERSECT, UNION, EXCEPT
	TAB_REL	relations between tables and their foreign keys, including FROM, JOIN, ON
	ARGS	a virtual relation, representing that parent node acts on child node, such as *agg*(*col*)
	KW_COM	combinations of *KW*s and operators to insure that one pair of (parent, child) has only one relation, e.g. SELECT with other *KW*s and *OP*, HAVING with *ARGS*

Table 1: The definition of the graph for the SQL query.

- **The constraints of graph.** We design measures to ensure that the node pair (parent, child) have exactly one relation in graph, to ensure that the graph can be generated by transition methods.

Then, all SQL queries can be transformed into graphs with 100% accuracy. Meanwhile, all graphs can be transformed into SQL queries which have fully covered sub-SQLs of the ground truth and exactly matched with them, the negatives (the percentage is 3.4%) that are failed to match the ground truth due to only the orders of sub-SQLs.

sub-SQLs	sub-graphs
$*agg* ($*col*)	($***agg***, $*col*, *ARGS*)
ORDER BY $*col* LIMIT $*limv*	($***col***, $*limv*, LIMIT)
$*col* *OP* $*value*	($***col***, $*value*, *OP*)

Table 2: Examples of transformations between sub-SQLs and sub-graphs, $***node*** is the parent of sub-graph and has relations with other sub-graphs.

5 Transition-based Graph Generation

We transform the generation of a SQL query into the graph generation, which is transformed into action-sequence generation, as shown in Figure 2.

5.1 Action Set for Graph Generation

A **cache transition parser** (Gildea et al., 2018) consists of a stack, a cache, and an input buffer, which has been shown to have good coverage of the graphs. The stack δ is a sequence of (integer, node) pairs, with the last element at the topmost position. The buffer β is a suffix sequence of input, with the first element to be read as a newly introduced node of the graph. Finally, the cache $\eta = [v_1, ..., v_m]$ is a sequence of nodes, with the first element at the leftmost position and the last element at the rightmost position. The configuration of the parser has the form:

$$C = (\delta, \eta, \beta, G_p) \quad (1)$$

Where δ, η and β are as described above, and G_p is the partial graph that has been built so far. The initial configuration of the parser is $([], [\$, ..., \$], [n_1, ..., n_m], \emptyset)$, meaning that the stack and the partial graph are initially empty, and the cache is filled with k occurrences of the special symbol \$. The buffer is initialized with the input sentence. The final configuration is $([], [\$, ..., \$], [], G)$, where the stack and the cache are as in the initial configuration and the buffer is empty. The constructed graph is the target semantic graph.

In the first step, we map the input sentence $w_{1:n} = w_1, ..., w_n$ to the candidate node sequence $n_{1:m} = n_1, ..., n_m$ by searching over KB (see Section 6.1 for detail). The input sentence and the candidate node sequence are as the input. The transitions of the parser are specified as follows.

- *NodeGen(n_i)* reads the next candidate node n_i and its corresponding token w_j or tokens span $w_{j:j+k}$ of the question, utilize a matching network (introduced in Section 5.3) to determine wether it should be left. If the node isn't left after the matching network, then it is mapped to ϵ which is shifted out of the buffer immediately, otherwise it is ready for further processing in the next two steps.

- *ArcGen(i, d, l)* builds an arc with direction d and label l between the leftmost node of buffer β and the i-th node in the cache η. The label l is *NONE* if no arc is made and we use the action *ArcNone* in this case. We consider all

arc decisions between the leftmost node of the buffer and each node of the cache. We can consider this phase as first making a binary decision whether there is an arc, and then predicting the label in case there is one, between each node pair.

- *PushNode(n_i)* shifts the next input node out of the buffer and moves it into the last position of the cache. We also take out the node n_j appearing at position j in the cache and push it onto the stack δ, along with the integer j recording its original position in the cache.

- *Pop* pops a pair (i, v) from the stack, where the integer i records the position in the cache that it originally came from. We place node v in position i in the cache, shifting the remainder of the cache one position to the right, and discarding the last element in the cache.

- *TableSel*[2] selects primary keys and foreign keys of generated tables $[t_i, ..., t_j]$, which is always generated at the start of one clause. Primary key is utilized as default column, such as in the SQL query *select count (*) from singer* for question *how many singers do we have*, * is short for the primary key. Foreign keys are utilized to join tables or group rows.

Our cache transition parser constructs the semantic graph of the example in Figure 1 is shown in Appendix A.1.

We use the equation $i^\star = argmax_{i\in[m]} \min\{j \mid (v_i, \beta_j) \in E_G\}$ to choose the cache node to take out in the action of *PushNode*, where $v_{i^\star}$ is the node from the cache whose closest neighbor in the buffer β is the furthest forward in β.

The correctness of the oracle is shown by Gildea et al. (2018).

5.2 Action Sequence Generation

We utilize Seq2Seq models to generate action-sequence with a question. Our model takes a question $w_{1:n}$ and its corresponding candidate node sequence $n_{1:m}$ as the input, and the action sequence $a_{1:q}$ as the output.

BiLSTM Encoder. Given an input word sequence $w_{1:n}$, we use a bidirectional LSTM to encode it. At each step j, the hidden states of both directions are concatenated as the final hidden state $h_j = [\overrightarrow{h_j}; \overleftarrow{h_j}]$ for word w_j.

[2]An action for the dataset with multiple tables

LSTM Decoder. We use an attention-based LSTM decoder (Bahdanau et al., 2014). For each time-step t, the decoder feeds the concatenation of the embedding of previous action e_{t-1} and the previous context vectors for words μ_{t-1} into the LSTM model to update its hidden state, as $s_t = LSTM(s_{t-1}, [e_{t-1}; \mu_{t-1}])$.

Then the attention probabilities α for time-step t is calculated as:

$$e_{t,i} = v_c^T tanh(W_h h_i + W_s s_t + b_c)$$
$$\alpha_{t,i} = \frac{exp(e_{t,i})}{\sum_{j=1}^{N} exp(e_{t,j})} \quad (2)$$

W_h, W_s, v_c and b_c are model parameters. The new context vector $\mu_t = \sum_{i=1}^{n} \alpha_{t,i} h_i$. The output probability distribution over all actions at the current state is calculated by:

$$P_{\sum_a} = softmax(V_a[s_t; \mu_t] + b_a) \quad (3)$$

where V_a and b_a are learnable parameters, and the number of rows in V_a represents the number of all actions. The symbol $\sum_a$ is the set of all actions.

Transition State Features for Decoder. We extract features from the current transition state configuration C_t:

$$e_f(C_t) = [e_{f_1}(C_t); e_{f_2}(C_t); ...; e_{f_l}(C_t)] \quad (4)$$

where l is the number of features extracted from C_t and $e_{f_k}(C_t)(k = 1, ..., l)$ represents the embedding for the k-th feature, which is learned during training. These feature embeddings are concatenated as $e_f(C_t)$, and fed as additional input to the decoder, as shown:

$$s_t = LSTM(s_t, [e_{t-1}; \mu_{t-1}; e_f(C_t)]) \quad (5)$$

We use the following features in our experiments:

- Node type: indicates the types of processed nodes which consist of current node of cache and the leftmost node of buffer, the set of types consist of (*agg, col*[3]*, cell, tab, limv, root, none*), *none* represents no type for node.

- Column-cell relation: If the types of current node pair are (*col*, *cell*) or (*cell*, *col*), we label 1 if the cell belongs to the column in the table, otherwise 0.

[3]Here, we individually label primary columns and foreign columns for table joined and column omitted

5.3 Matching Network for NodeGen Action

Given the input sentence and its candidate node sequence, a matching network is used to select nodes (the action of *NodeGen*). We construct a classification model for selection of nodes, based on matching between questions and KBs. The column-cell relation can improve the prediction of column (Sun et al., 2018), as that a cell or a part of it typically appears in the question acting as the WHERE value, but a column name might be represented with a totally different utterance or need be predicted by its corresponding cells (such as "players from France" indicating "Country = France"). We propose a multiple levels of granularity matching network using column-cell relation, as show on the lower right of Figure 2.

In this paper, we represent a node as a KB sentence. For columns, the KB sentence is represented as $x_i^o = [w_{i1}^o; w_{i2}^o; ...; w_{im}^o]$, m is the number of words in i-th column. For cells, it is represented as $x_{ij}^c = [x_j^o; w_{ij_1}^c; ...; w_{ij_k}^c]$, k is the number of words in (i, j)-th cell, the representation combines semantic of columns and cells which can be utilized to improve the prediction of columns. Similarly, for table names, the sentence is the concatenation of table names and all its columns; for aggregators and value of limit, the sentence is the concatenation of their full names and synonyms.

Given a question q and a KB sentence r, the model looks up an embedding table and represents q and r as $Q = [e_{q,1}, ..., e_{q,n_q}]$ and $R = [e_{r,1}, ..., e_{r,n_r}]$ respectively, where $e_{q,i}, e_{r,i} \in R^d$ are the embeddings of the i-th word of q and r respectively. $Q \in R^{d*n_q}$ and $R \in R^{d*n_r}$ are then used to construct a word-word similarity matrix $M_1 \in R^{n_q*n_r}$ and a sequence-sequence similarity matrix $M_2 \in R^{n_q*n_r}$ which are two input channels of a convolutional neural network (CNN). The CNN distills important matching information from the matrices and encodes the information into a matching vector v.

Specifically, $\forall i, j$, the (i, j)-th element of M_1 is defined by

$$e_{1,i,j} = e_{q,i}^\top \cdot e_{r,j} \tag{6}$$

M_1 models the matching between q and r on a word level.

To construct M_2, we first employ a BiLSTM to transform Q and R into hidden vectors, get $H_q = [h_{q,1}, ..., h_{q,n_q}]$ as the hidden vectors of Q and $H_r = [h_{r,1}, ..., h_{r,n_r}]$ as the hidden vectors of R. Then, $\forall i, j$, the (i, j)-th element of M_2 is defined by

$$e_{2,i,j} = h_{q,i}^\top A h_{r,j} \tag{7}$$

where $A \in R^{m \times m}$ is a linear transformation. M_2 models the matching between q and r on a segment level.

M_1 and M_2 are then processed by a CNN to form vector v. $\forall f = 1, 2$, CNN regards M_f as an input channel, and alternates convolution and max-pooling operations. The output of the final feature maps are concatenated and mapped to a low dimensional space with a linear transformation, as the matching vector $v \in R^q$.

By learning word embedding and parameters of LSTM from training data, words or segments in a question that are useful for selecting a node always have high similarity with some words or segments in the KB sentences and result in high value areas in the similarity matrices. These areas will be transformed and selected by convolution and pooling operations and carry important information in the utterance to the matching vector. This is how our model identifies important information in question and leverage it in selecting nodes from KB sentences.

5.4 Training and Inference

We train our models using two cross-entropy losses, one is over on the oracle action sequence $a_1, ..., a_q$, the other is over on the selection of nodes:

$$L = -\sum_{t=1}^{q} \log P_a(a_t | a_{<t}, n_t, X) - \sum_{t=1}^{q} \log P_n(n_t | X, R_t) \tag{8}$$

where X represents the input question and R is the corresponding candidate node sequence, n_t represents the leftmost node of input buffer after action of *NodeGen* and R_t is the candidate node of time t.

6 Experiments

We conduct experiments on the WikiSQL dataset[4]. Following other models, we use three evaluation metrics, logical form accuracy (Acc_{lf}) measures the percentage of the examples that have exact string match with the ground truth, query match accuracy (Acc_{qm}) eliminate the false negatives due to only the ordering issue based on Acc_{lf}, execution accuracy (Acc_{ex}) measures the proportion of correct answers.

[4]https://github.com/salesforce/WikiSQL

Mothods	Dev			Test		
	Acc_{lf}	Acc_{qm}	Acc_{ex}	Acc_{lf}	Acc_{qm}	Acc_{ex}
Attentional Seq2Seq	23.3%	–	37.0%	23.4%	–	35.9%
Aug.PntNet (Zhong et al., 2017)	44.1%	–	53.8%	43.3%	–	53.3%
Seq2SQL (Zhong et al., 2017)	49.5%	–	60.8%	48.3%	–	59.4%
SQLNet (Xu et al., 2017)	–	63.2%	69.8%	–	61.3%	68.0%
STAMP (Sun et al., 2018)	61.7%	–	75.1%	61.0%	–	74.6%
COARSE2FINE (Dong and Lapata, 2018)	–	–	–	71.7%	–	78.5%
Seq2Graph	70.6%	74.7%	80.1%	71.1%	74.9%	80.7%

Table 3: Performances of different approaches on the wikiSQL dataset. Three evaluation metrics are logical form accuracy (Acc_{lf}), query match accuracy (Acc_{qm}) and execution accuracy (Acc_{ex}). Our model is (**Seq2Graph**).

Our parameters setting follows Xu et al. (2017). Adam (Kingma and Ba, 2014) with a learning rate of 0.0001 is used as the optimizer, and the model that yields the best performance on the dev set is selected to evaluate on the test set. Dropout with rate 0.3 is used during training. Beam search with a beam size of 10 is used for decoding. Both training and decoding use a K40 GPU. Hidden state sizes for all encoders and decoder are set to 100. The word embeddings are initialized from Glove pre-trained word embeddings (Pennington et al., 2014) on Common Crawl, and are updated during training.

6.1 Preprocess and Postprocess

Before Generating the action sequence, the candidate node sequence should be obtained firstly.

Generation for node sequence[5]. We use every word and its stem[6] to search over the KB, get candidate nodes for every word. Meanwhile, we use some rules to improve the recall of systems, such as the similarity of word embedding (see Appendix B for detail). Then we merge the span of words which are corresponding to the same node, and get the candidate node sequence[7]. In the generation of action sequence, for every candidate node, we use the matching network (5.3) to determine whether it should be left in *NodeGen* action. If the node is redundant, it has the relation *ArcNone* with all other nodes, and will be discarded in the transformation from action sequence to SQL query.

Action Sequence to SQL query. Given action sequence, graph is obtained in the deterministic way. As introduced in 4.2, the transformation from graph to SQL query is in the deterministic way too. Here, we delete edges whose label is *ArcNone* and nodes which are isolated from graph.

[5]We don't strictly restrict the alignment on boundaries, just the order of nodes.

[6]http://www.nltk.org

[7]All nodes which belong to the same span are organized into a sequence by the order of [*agg*, *col*, *cell*, *limv*]

6.2 Model Comparisons

We compare our model (Seq2Graph) against several previously published systems.

- Zhong et al. (2017) develop Seq2SQL, in which two separate classifiers are trained for SELECT aggregator and SELECT column, separately, and utilizing reinforcement learning for further model training. Results of Aug.PntNet and Attentional Seq2Seq are also reported in (Zhong et al., 2017).
- Xu et al. (2017) develop SQLNet, which predicts SELECT clause and WHERE clause separately, sequence-to-set architecture and column attention are adopted to predict the WHERE clause.
- Sun et al. (2018) develop STAMP, which is an end-to-end learning pointer network with multiple channels and utilizes column-cell relation to improve WHERE clause.
- Dong and Lapata (2018) develop COARSE2FINE, in which two separate classifiers are trained for SELECT aggregator and SELECT column, a sketch encoder-decoder model is trained for WHERE clause in which a classifier is trained for the selection of sketch.

Both SQLNet and COARSE2FINE are only applicable to wikiSQL datasets, as they restrict the numbers of column and aggregator in SELECT clause.

Our model represents SQL query as a graph which is applicable to any text-to-SQL dataset, and utilizes transition-based Seq2Seq model to generate the graph as a end-to-end process. From Table 3, we can see that Seq2Graph is competitive with other models which have multiple sub-models or tailored decoders, and performs better than existing systems on Acc_{ex} with 2% points.

Methods	Dev			Test		
	Acc_{sel}	Acc_{agg}	Acc_{where}	Acc_{sel}	Acc_{agg}	Acc_{where}
Seq2SQL (Zhong et al., 2017)	89.6%	90.0%	62.1%	88.9%	90.1%	60.2%
SQLNet (Xu et al., 2017)	91.5%	90.1%	74.1%	90.9%	90.3%	71.9%
STAMP (Sun et al., 2018)	89.6%	89.7%	77.3%	90.0%	89.9%	76.3%
SQLGRAPH	92.0%	89.8%	79.7%	90.0%	91.4%	79.6%

Table 4: Performances of the matching network for node generation on WikiSQL dev and test sets. Accuracy (Acc_{lf}) is evaluated on SELECT column (Acc_{sel}), aggregator (Acc_{agg}), and WHERE clause (Acc_{where}), respectively.

6.3 Result Analysis

Following other models, we also report more fine-grained evaluation metrics over constituents of SQL queries, as shown in Table 4. The improvement of WHERE clause is consistent with our primary intuition on improving the prediction of column and cell (value of WHERE clause) with matching based on multiple levels of granularity and column-cell relation.

We give a case study to illustrate the generation results by Seq2Graph, with a comparison to SQLNet. Results are given in Figure 3. In the first example, Seq2Graph utilizes the history of actions and column-cell relation to improve the predictions of edges, the parent of the cell *de vaux continental* is *make*, as *model* has been the child of *root* with relation *SELECT*. In the second example, Seq2Graph can predict column (*location*) by cell (*leicester*) to improve the predictions of columns.

question	What model has de vaux continental as the make
SQL query	SELECT model WHERE make = de vaux continental
SQLNet	SELECT model WHERE model = de vaux continental
Seq2Graph	SELECT model WHERE make = de vaux continental
question	who is the coach of the team in Leicester?
SQL query	SELECT coach WHERE location = leicester
SQLNet	SELECT coach WHERE team = leicester
Seq2Graph	SELECT coach WHERE location = leicester

Figure 3: Examples of Seq2Graph, Compared with SQLNet

The node sequence is automatically obtained with searching and matching, Table 5 shows the recall of nodes and the accuracy of node sequence, from which we can see that Seq2Graph needs to improve the matching between the question and KB to improve selections of nodes. Meanwhile, we find that the accuracies of edges with operators are lower, which may be improved utilizing contextual information between nodes which is independent with nodes.

	train	dev and test
Node Recall (R_n)	91.7%	92.1%
Node Sequence Accuracy (P_{ns})	83.6%	82.1%

Table 5: Performances of Node sequence. R_n measures the percentage of the questions that all standard nodes are recalled after searching, P_{ns} measures the percentage of the questions that all standard nodes are in the right order after matching (Evaluation on randomly cases).

6.4 Discussion

Compared to the models that restrict SQL queries to fixed forms of "select-aggregator-where", our model is less tailored. We believe that it is easy to apply our model to other datasets with appropriate definition of graph. We take full advantages of graph representation. The generation of graph can be recognized as a bottom-up process, and is decomposed into two stages, at the first stage, focuses on matching between questions and KBs to select units of KBs which are as nodes of the graph, utilizes information of questions that has a tight-coupling with KBs; at the second stage, focuses on commonalities between syntactic structures of questions and syntax of logical forms to generate edges between nodes, utilizes information of questions that is independent with KBs.

7 Conclusion

In this paper, we transform text-to-SQL task into graph generation, and utilize transition-based Seq2Seq model to generate the graph. The proposed model can synthesize the advantages of semantic graph representation and the prediction ability of Seq2Seq models, and are applicable to any text-to-SQL dataset. Experimental results show that SeqGraph model get competitive results compared with other models. In the future, we would like to apply the framework to other logical forms and datasets, and improve transition using monotonic hard attention model with contextual information between nodes.

References

David Alvarez-Melis and Tommi S Jaakkola. 2016. Tree-structured decoding with doubly-recurrent neural networks.

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

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280*.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*.

Daniel Gildea, Giorgio Satta, and Xiaochang Peng. 2018. Cache transition systems for graph parsing. *Computational Linguistics*, 44(1):85–118.

Alessandra Giordani and Alessandro Moschitti. 2012. Translating questions to sql queries with generative parsers discriminatively reranked. *Proceedings of COLING 2012: Posters*, pages 401–410.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760*.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*.

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

Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. 2017. Neural amr: Sequence-to-sequence models for parsing and generation. *arXiv preprint arXiv:1704.08381*.

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.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84.

Yunyao Li, Huahai Yang, and HV Jagadish. 2006. Constructing a generic natural language interface for an xml database. In *International Conference on Extending Database Technology*, pages 737–754. Springer.

Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 59(9):68–76.

Percy Liang, Michael I Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

Xiaochang Peng, Daniel Gildea, and Giorgio Satta. 2018a. Amr parsing with cache transition systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-18)*.

Xiaochang Peng, Linfeng Song, Daniel Gildea, and Giorgio Satta. 2018b. Sequence-to-sequence models for cache transition systems. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1842–1852.

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 (EMNLP)*, pages 1532–1543.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157. ACM.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association of Computational Linguistics*, 2(1):377–392.

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. *arXiv preprint arXiv:1804.08338*.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive sql queries from input-output examples. In *ACM SIGPLAN Notices*, volume 52, pages 452–466. ACM.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015. A transition-based algorithm for amr parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 366–375.

David HD Warren and Fernando CN Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8(3-4):110–122.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1341–1350.

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

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

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017b. Type-and content-driven synthesis of sql queries from natural language. *arXiv preprint arXiv:1702.01168*.

Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question answering with knowledge base.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

John M Zelle and Raymond J Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pages 1050–1055.

Luke S Zettlemoyer and Michael Collins. 2012. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*.

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

A Appendices

A.1 Example of cache transition system

Our cache transition parser constructs the semantic graph of the example in Figure 1 is shown in Figure 4.

A.2 Transformation Between SQL Query and Graph

We would like to ensure that the transformations between SQL queries and graphs are in the deterministic ways. Our transformation considers three aspects, and following three aspects will be introduced in detail:

The coverage of SQL keywords. We collect all patterns of sub-SQLs from two text-to-SQL datasets (wikiSQL and Spider) and SQL grammars, and ensure that they can be transformed into sub-graphs, as shown in Table 6, the first column shows patterns of sub-SQLs, the corresponding sub-graphs are shown in second column.

- For the sub-SQL of $*A_S*, the $*col* is related to $*root* with relation *SELECT* if there is no $*agg*.
- For the sub-SQL of $*C_S*, the $**value** may be a nested query which will be represented as a new sub-SQL $*Sel_S*, we add a new node $*t_root* as the whole parent of the sub-graph.
- For the sub-SQL of $*Sel_S*, we add a new virtual node $*t_root* as the parent of sub-graph, which will have relations with other sub-graphs.

Using a depth-first-search algorithm, all SQL queries can be transformed into graphs in the deterministic ways.

The consistency of transformation from graph to SQL query. We design the parent-child protocol for coordinative constituents and the order for sub-SQLs to ensure that the graph can be transformed into SQL query in the deterministic way.

- We design the order of sub-SQLs in a SQL query, the general pattern of a SQL query is *SELECTFROM WHERE GROUP BY ORDER BY*, the order of other sub-SQLs is consistent with their orders in questions.

stack	cache	buffer	actions	Graph
[]	[$, $, $, ROOT]	[What(), is(), the(), average(avg),]	NodeGen(); NodeGen(); NodeGen()	G = ()
[]	[$, $, $, ROOT]	[average(avg), minimum(min), and()]	NodeGen(1)	G
[]	[$, $, $, ROOT]	[(avg/**agg**), minimum(min), and()]	ArcGen(3,r,SELECT) (2,-,None) (1,-,None) (0,-,None)	G += (0,1,SELECT)
[(0,$)]	[$, $, ROOT, avg]	[minimum(min), and(), maximum(max)]	PushNode	G
[(0,$)]	[$, $, ROOT, avg]	[minimum(min), and(), maximum(max)]	NodeGen(2)	G
[(0,$)]	[[$, $, ROOT, avg]	[(min/**agg**), and(), maximum(max)]	ArcGen(3,-,None) (2,r,SELECT) (1,-,None) (0,-,None)	G += (0,2,SELECT)
[(0,$) (0,$)]	[$, ROOT, avg, min]	[and(), maximum(max), age(age), of()]	PushNode	G
[(0,$) (0,$)]	[$, ROOT, avg, min]	[and(), maximum(max) age(age), of()]	NodeGen()	
[(0,$) (0,$)]	[$, ROOT, avg, min]	[maximum(max), age(age), of(), all()]	NodeGen(3)	G
[(0,$) (0,$)]	[$, ROOT, avg, min]	[(max/**agg**), age(age), of(), all()]	ArcGen(3,-,None) (2,-,None) (1,r,SELECT) (0,-,None)	G += (0,3,SELECT)
[(0,$) (0,$) (0,$)]	[ROOT, avg, min, max]	[age(age), of(), all(), singers(singer)]	PushNode	G
[(0,$) (0,$) (0,$)]	[ROOT, avg, min, max]	[age(age) , of(), all(), singers(singer)]	NodeGen(4)	G
[(0,$) (0,$) (0,$)]	[ROOT, avg, min, max]	[(age/**col**), of(), all(), singers(singer)]	ArcGen(3,r,ARGS) (2,r,ARGS) (1,r,ARGS) (0,-,None)	G+=(3,4,ARGS)(2,4,ARGS) (1,4,ARGS)
[(0,$) ··· (1,avg)]	[ROOT, min, max, age]	[of(), all(), singers(singer), from()]	NodeGen() NodeGen()	G
[(0,$) ··· (1,avg)]	[ROOT, min, max, age]	[singers(singer), from(), France(···)]	NodeGen(5)	G
[(0,$) ··· (1,avg)]	[ROOT, min, max, age]	[(singer/**tab**), from(), France(···)]	ArcGen(3,-,None) (2,-,None) (1,-,None) (0,r,FROM)	G +=(0,5,FROM)
[(0,$) ··· (1,min)]	[ROOT, max, age, singer]	[from(),France(country france)]	NodeGen()	G
[(0,$) ··· (1,min)]	[ROOT, max, age, singer]	[France(coutry france)]	NodeGen(6,7)	G
[(0,$) ··· (1,min)]	[ROOT, max, age, singer]	[(coutry france/**col cell**)]	ArcGen(3,-,None) (2,-,None) (1,-,None) (0,r,WHERE)	G += (0,6,WHERE)
[(0,$) ··· (1,max)]	[ROOT, age, singer, country]	[(france/cell)]	PushNode	
[(0,$) ··· (1,min)]	[ROOT, age, singer, country]	[(france/**cell**)]	ArcGen(3,r,OP(=)) (2,-,None) (1,-,None) (0,-,Node)	G += (6,7,OP(=))
[(0,$) ··· (1,age)]	[ROOT, singer, country, cell]	[]	PushNode	G
[]	[$, $, $, ROOT]	[]	Pop; Pop; Pop; Pop; Pop; Pop; Pop	G

Figure 4: Example run of the cache transition system constructing the graph for the sentence *What is the average, minimum, and maximum age of all singers from France?* with the candidate node sequence *What is the average (**avg**), minimum (**min**), and maximum (**max**) age (**age**) of all singers (**singer**) from France (**france**) ?* and cache size of 4. The left 3 and last columns show the parser configurations after the actions shown in the forth column. Here, the candidates of node in the action of *NodeGen* are the exact one which may include inaccurate ones in actully

sub-SQLs	sub-graphs
$*Sent_S* = *Sel_S* (*SUB_REL Sel_S*)⋆	($*root*, $*t_root*, *SUB_REL*)
$*Sel_S* = ($*A_S*)+ $*F_S* ($*W_S*)? ($*G_S*)⋆ ($*R_S*)⋆	($*root*, parent of $*A_S*, SELECT)+ ($*root*, $*tab*, FROM) ($*root*, parent of $*W_S*, WHERE)? ($*root*, $*col*, GROUP BY)⋆ ($*root*, $*col*, ORDER BY DESC/ASC)⋆
$*A_S* = ($*agg*)? $*col*	($*agg*, $*col*, *ARGS*)?
$*F_S* = FROM $*tab* (JOIN $$tab_2$$ ON $$col_1$$ = $$col_2$$)⋆	($*tab*, $$tab_2$$, JOIN) ($$col_1$$, $$col_2$$, *OP(=)*) ($*tab*, $$col_1$$, ON)
$*G_S* = *GROUP BY* $*col* (HAVING $*A_S OP* $value)?	($*col*, parent of $*A_S*, HAVING)? (parent of $*A_S*, parent of $*value*, *OP*)?
$*R_S* = *ORDER BY DESC/ASC* $*col* (LIMIT $*limv*)?	($*col*, $*limv*, LIMIT)?
$*C_S* = $*col OP* $*value* / $*Sel_S*	($*col*, $*value*, *OP*) or ($*col*, $*t_root*, *SELECT_OP*)
$*W_S* = WHERE $$C\_S_1$$ (*CON_REL* $$C\_S_2$$)⋆	(parent of $$C\_S_1$$, parent of $$C\_S_2$$, *CON_REL*)⋆

Table 6: Transformation between the SQL query and graph, the SQL query is decomposed to sub-SQLs and the graph is represented as pairs of (parent, child, relation), ⋆ represents 0 or more, + represents 1 or more, ? represents 0 or 1. The $value of $*C_S* may be a nested query, which can be represented as another $*Sel_S*

- For the coordinative constituents in SQL queries, such as coordinative conditions *$a AND $b AND $c* in *WHERE* clause, we design the parent-child protocol which sets the front constituent as parent and the next as child with the relation between them, the sub-graph is *($a, $b, AND) ($b, $c, AND)*.

Then the graph can be transformed into SQL query in a deterministic way, although the generate SQL query may be failed to match the ground truth due to the order-matters problem of sub-SQLs (the percentage of queries with order-matters problem is 3.4%).

The constraints of Graph. The node pair (parent, child) must have exactly one relation in graph. We design some measures to ensure that:

- Combine some keywords of SQL language into one when they relate to one node pair and the pair only occurs once in question, including combinations of (*SELECT*, other *KW*), (*SELECT*, *OP*), (*SELECT*, *OP*, *ARGS*) and (*HAVING*, *ARGS*). Figure 5 gives some examples (see cases A and B), the procedures for other combinations are similar.
- Adjust the (parent, child) pair of relations in *SUB_REL* group to the first (parent, child)

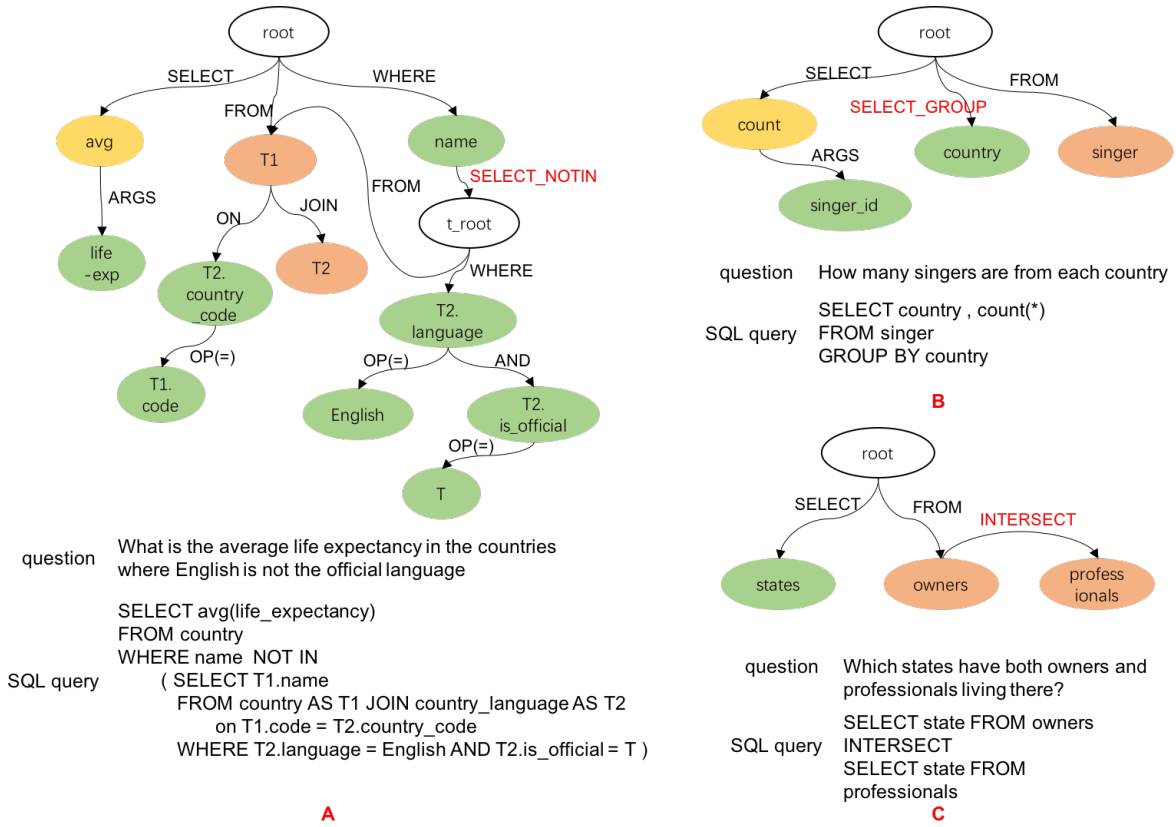

Figure 5: Examples for graph constraints, the first example is for combining *SELECT* and operator *NOT IN*, the second example is for combining *SELECT* and *GROUP BY*, the third example shows the adjustment of *SUB_RELs*.

pair which are the parent of sub-graphs and different in two $*Sel_S*s, Figure 5 gives an example (see case C).

B Preprocess: Generation of candidate nodes

We use some rules to improve the recall of systems.

- For aggregators and value of *limit*, we collect synonyms for them, such as (**count number**) for *count*, (**total amount**) for *sum*, (**minimun**) for *min*, (**maximum**) for *max*, (**average**) for *avg*, (**top** for *limv*). Meanwhile, we recognize the superlative form of adjectives and using it as a candidate synonym for *min*, *max*, and *limv*.

- We use the similarity of words embedding (threshold is **0.6**) to improve the recall of columns, but don't compute with stop words, such as *the*, *a* etc..

- We automatically add the column as node to current word or span if a cell that belongs to it is recognized at current position.

- We automatically add the cells as nodes to current word or span if its corresponding column is recognized at current position, and the values of all cells are binary (such as *True* and *False*).

- We automatically add the primary columns and foreign columns of the table if it is recognized at current position, for omissions of columns and table joined.

- We automatically add the $*t_root* as nodes to current word if it is the pronouns (e.g. which, that, who, etc.), conjunctions (e.g. and, or, etc.) or negative words (e.g. not, don't. etc.) that can indicate a subordinate clause.

- We assign number and date into *NUMBER* both in questions and KB, in order to improve matching between number and date.