DBTagger: Multi-Task Learning for Keyword Mapping in NLIDBs Using Bi-Directional Recurrent Neural Networks

Arif Usta
Bilkent University
Ankara, Turkey
arif.usta@bilkent.edu.tr

Akifhan Karakayali
Bilkent University
Ankara, Turkey
akifhan@bilkent.edu.tr

Özgür Ulusoy
Bilkent University
Ankara, Turkey
oulusoy@cs.bilkent.edu.tr

ABSTRACT

Translating Natural Language Queries (NLQs) to Structured Query Language (SQL) in interfaces deployed in relational databases is a challenging task, which has been widely studied in database community recently. Conventional rule based systems utilize series of solutions as a pipeline to deal with each step of this task, namely stop word filtering, tokenization, stemming/lemmatization, parsing, tagging, and translation. Recent works have mostly focused on the translation step overlooking the earlier steps by using adhoc solutions. In the pipeline, one of the most critical and challenging problems is keyword mapping; constructing a mapping between tokens in the query and relational database elements (tables, attributes, values, etc.). We define the keyword mapping problem as a sequence tagging problem, and propose a novel deep learning based supervised approach that utilizes POS tags of NLQs. Our proposed approach, called *DBTagger* (DataBase Tagger), is an end-to-end and schema independent solution, which makes it practical for various relational databases. We evaluate our approach on eight different datasets, and report new state-of-the-art accuracy results, 92.4% on the average. Our results also indicate that DBTagger is faster than its counterparts up to 10000 times and scalable for bigger databases.

PVLDB Reference Format:
Arif Usta, Akifhan Karakayali, and Özgür Ulusoy. DBTagger: Multi-Task Learning for Keyword Mapping in NLIDBs Using Bi-Directional Recurrent Neural Networks. PVLDB, 14(5): 813 - 821, 2021.
doi:10.14778/3446095.3446103

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/arifusta/DBTagger.

1 INTRODUCTION

Amount of processed data has been growing rapidly pertaining to technology, leading database systems to have a great deal of importance in today's world. Amongst the systems, relational databases are still one of the most popular infrastructures to effectively store data in a structured fashion. To extract data out of a relational database, *structured query language (SQL)* is used as a standard tool. Although SQL is a powerfully expressive language, even technically skilled users have difficulties using SQL. Along with the syntax of SQL, one has to know the schema underlying the database upon which the query is issued, which further causes hurdles to use SQL. Consequently, casual users find it even more difficult to express their information need, which makes SQL less desirable. To remove this barrier, an ideal solution is to provide a search engine like interface, such as Google or Bing in databases. The goal of *Natural Language Interfaces to Databases (NLIDB)* is to break through these barriers to make it possible for casual users to employ their natural language to extract information.

To this end, many works have been published recently attacking the research problem of translation of natural language queries into SQL; such as conventional pipeline based approaches [4, 24, 32, 35] or end-to-end solutions using encoder-decoder based deep learning approaches [20, 33, 34, 41, 44]. Neural network based solutions seem promising in terms of robustness, covering semantic variations of queries. However, they struggle for queries requiring translation of complex SQL queries, such as aggregation and nested queries, especially if they include multiple tables. They also have a huge drawback in that they need many SQL-NL pairs for training to perform well, which makes conventional pipeline based solutions still an attractive alternative. [30].

In the translation pipeline, one of the most important sub-problems is *keyword mapping*, as noted in [25] as an open challenge to be addressed in NLIDBs. *Keyword mapping* task requires to associate each token or a series of consecutive tokens (e.g., keywords) in the natural language query to a corresponding database schema element such as table, attribute or value. It is the very first step of resolving ambiguity for translation. Xu et. al [34] also note that during the translation of the query, *where* clause is the most difficult part to generate which further signifies the task of *keyword mapping.*

Consider the below natural language query examples run on the sample IMDB movie database shown in Figure 1 to better understand the challenges in *keyword mapping* problem.

Example NL Query 1. *"What is the writer of The Truman Show?"*

Challenge 1. The very first challenge in *keyword mapping* is to differentiate and categorize tokens in the query either as database relevant or not. For instance, some of the words in Example 1 (e.g., "is", "the", "of") are just stop words that are needed not to be considered as potential mapping target. An ad-hoc solution is to filter certain words using a pre-defined vocabulary, however such a solution removes "The" in Example 1 preceding the actual database value that needs to be mapped, which will cause the wrong translation.

Challenge 2. Another important challenge is to detect multi-word entities (mostly database values), "The Truman Show" in Example

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446103

1. The most common approach is to build look-up tables or indexes on n-grams of database values and calculate semantic and/or lexical similarity over the candidates. Yet, this is a costly process for on-the-fly calculations regarding possible n-grams of the given NL query.

Example NL Query 2. *"Find all movies written by Matt Demon."*

Example NL Query 3. *"How many movies are there that are directed by Steven Spielberg and featuring Matt Demon?"*

Challenge 3. Consider the queries given in Examples 2 and 3. In the queries, tokens ("written" and "featuring") referring to database tables are syntactic and semantic variations of the actual table ("written_by" and "cast" respectively) that they mapped to in the database (Figure 1). To handle such a challenge, lexical and semantic similarities of tokens over database elements (table and attributes) can be calculated using a third party database such as WordNet [29]. However, in addition to being a costly process to calculate such similarities online, such a solution cannot cover all possible variations of every map target in the database schema. Also, similarity calculation approach requires a manually crafted threshold, τ, to determine how much similarity is sufficient to map to a particular schema element, which makes it undesirable.

Challenge 4. One of the usages of *keyword mapping* step is to resolve ambiguities before getting into translation step. In the above examples, "Matt Demon" refers to a database value residing in multiple tables (e.g., actor, writer). Actual mapping of the keyword is determined by the mappings of neighbouring words surrounding, which implies that query-wise labelling considering *coherence* rather than independent labelling can be beneficiary.

Challenge 5. In addition to an effective solution, an ideal keyword mapping approach must be efficient to be deployed on interfaces where users run queries online. Mapper should output the result in reasonable time.

Most of the pipeline-based state-of-the-art works do not provide a novel solution to the problem of *keyword mapping*, rather they utilize unsupervised approaches such as simple look-up tables looking for exact matches or pre-defined synonyms [4, 31]; or they make use of an existing lexical database [24] such as WordNet [29]; or they exploit domain information to extract an ontology to be used for the task [32]; or they employ distributed representations of words [35] such as word2vec [28] to calculate semantic similarity of tokens over database elements. Although these approaches are effective to some extent, they fail to solve various challenges mentioned by the task of *keyword mapping* single-handedly.

In order to address all of the challenges mentioned above we propose DBTagger, a novel deep sequence tagger architecture used for *keyword mapping* in NLIDBs. Our approach is applicable to different database domains requiring only handful of training query annotations and practical to be deployed in online scenarios finding tags in just milliseconds. In particular, we make the following contributions by proposing DBTagger:

- We tackle the keyword mapping problem as a sequence tagging problem and borrow state-of-the-art deep learning approaches tailored for well-known NLP tasks.

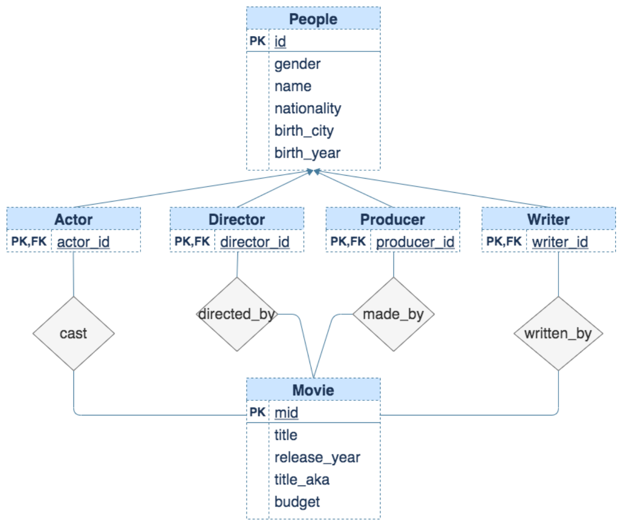


Figure 1: ER diagram of a subset of IMDB movie database

- We extend the neural structure for sequence tagging, by utilizing *multi-task learning* and *cross-skip connections* to exploit the observation we made in natural language query logs of databases, that is, schema tags of keywords are highly correlated with POS tags.
- We manually annotate query logs from three publicly available relational databases, and five different schemas belonging to Spider [42] dataset.
- We evaluate DBTagger, with above-mentioned query logs in two different setups. First, we compare DBTagger with unsupervised baselines preferred in state-of-the-art NLIDBs. In the latter, we evaluate DBTagger architecture by comparing with different supervised neural architectures. We report new state-of-the-art accuracy results for keyword mapping in all datasets.
- We provide comprehensive run time and memory usage analysis over the existing keyword mapping approaches. Our results show that, DBTagger is the most efficient and scalable approach for both metrics.

The remainder of this paper is organized as follows. In the next section, we explain the problem formulation and methodology we follow. We present the neural network structure we designed to solve the keyword mapping problem, and discuss how annotation of queries is handled. In Section 3, we provide experimental results comparing DBTagger with unsupervised baselines and present the performance of different neural models to justify DBTagger architecture. We also provide an efficiency analysis on all baselines. In Section 4, we summarize related work. Section 5 concludes the paper.

2 METHODOLOGY

2.1 Deep Sequence Tagger Architecture

POS tagging and NER refer to sequence tagging problem in NLP for a particular sentence to identify parts-of-speech such as noun, verb, adjective and to locate any entity names such as person, organization, respectively. Recurrent Neural Networks (RNN) are at the core of architectures to handle such problems, since they are a

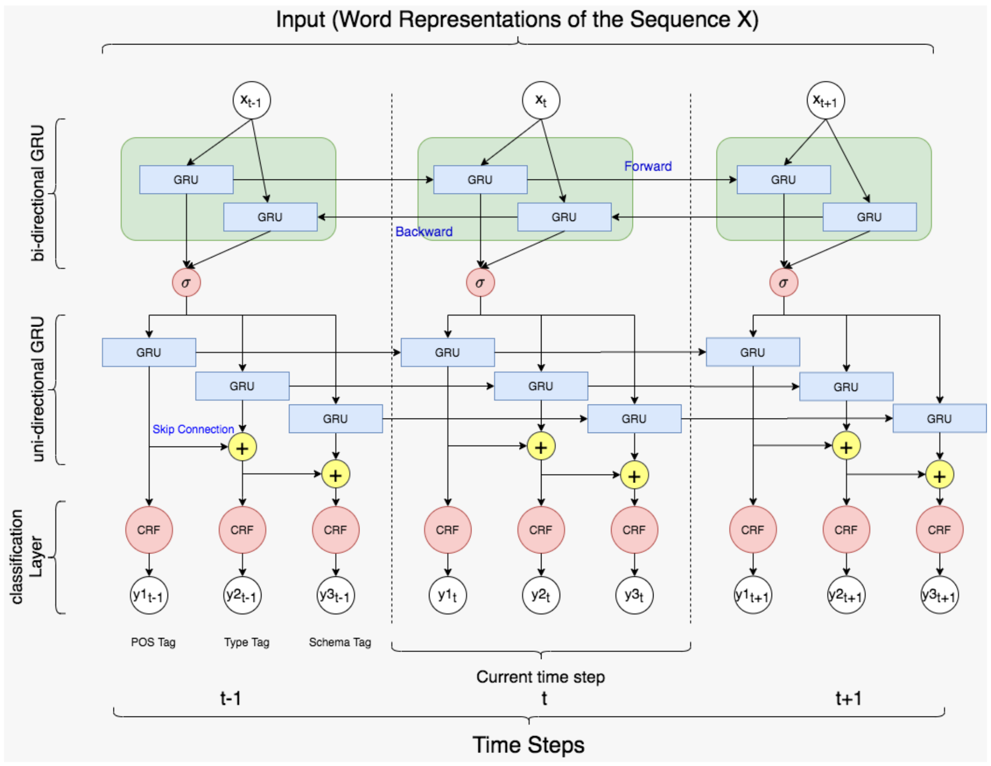


Figure 2: DBTagger Network

family of networks that perform well on sequential data input such as a sentence.

In RNN networks, the basic goal is to carry past information (previous words) to future time steps (future words) to determine values of inner states and consequently the final output, which makes them preferable architecture for sequential data. Given x_t as input at time step t, calculation of hidden state h_t at time step t is as follows:

$$h_t = f(Ux_t + Wh_{t-1}) \tag{1}$$

In practice, however, RNN networks suffer from *vanishing gradient problem*, therefore the limitation was overcome by modifying the gated units of RNNs; such as LSTM [16] and GRU[8]. Compared to vanilla RNN, LSTM has *forget gates* and GRU comprises of *reset* and *update* gates additionally. We experimented with both structures and we chose GRU due to its better performance in our experiments. In GRU, Update Gates decide what information to throw away and what new information to add, whereas Reset Gate is utilized to decide how much past information to forget.

In sequence tagging problem, in addition to past information we also have future information as well at a given specific time, t. For a particular word w_i, we know the preceding words (past information) and succeeding words (future information), which can be further exploited in the particular network architecture called, *bi-directional RNN* introduced in [11]. Bi-directional RNN has two sets of networks with different parameters called forward and backward. The concatenation of the two networks is then fed into the last layer, where the output is determined. This process is demonstrated in the upper part of the Figure 2, named bi-directional GRU.

Sequence tagging is a supervised classification problem where the model tries to predict the most probable label from the output space. For that purpose, although conventional *softmax* classification can be used, *conditional random field (CRF)* [22] is preferred. Unlike independent classification by softmax, CRF tries to predict labels sentence-wise by taking labels of the neighboring words into consideration as well. This feature of CRF is what makes it an attractive choice especially in a problem like *keyword mapping*. CRFs for each class of tags are appended to uni-directional GRU, depicted in lower part of the Figure 2.

2.2 DBTagger Architecture

Formally, for a given NL query, input X becomes a series of vectors $[x_1, x_2, ...x_n]$ where x_i represents the i^{th} word in the query. Similarly, output vector Y becomes $[y_1, y_2, ...y_n]$ where y_i represents the label (actual tag) of the y^{th} word in the query. Input must be in numerical format, which implies that a numerical representation of words is needed. For that purpose, the word embedding approach is state-of-the-art in various sequence tagging tasks in NLP [9] before feeding into the network. So, embedding matrix is extracted for the given query, $W \in R^{nxd}$, where n is the number of words in the query and d is the dimension of the embedding vector for each word. For the pre-calculated embeddings, we used fastText[6] due to it being one of the representation techniques considering subword (character n-grams) as well to deal with the out of vocabulary token problem better.

We consider G to be 2-dimensional scores of output by the uni-directional GRU with size $n \times k$ where k represents the total number of tags. $G_{i,j}$ refers to score of the j^{th} tag for the i^{th} word. For a sequence Y and given input X, we define tag scores as;

$$s(X, Y) = \sum_{i=1}^{n} A_{y_i, y_{i+1}} + \sum_{i=1}^{n} G_{i, y_i} \tag{2}$$

where A is a transition matrix in which $A_{i,j}$ represents the score of a transition from the i^{th} tag to the j^{th} tag. After finding scores, we define probability of the sequence Y:

$$p(Y|X) = \frac{e^{s(X,Y)}}{\sum_{\bar{Y} \in Y_x} e^{s(X,\bar{Y})}} \tag{3}$$

where $\bar{Y}$ refers to any possible tag sequence. During training we maximize the log-probability of the correct tag sequence and for the inference we simply select the tag sequence with the maximum score.

In our architecture, we utilize *Multi-task learning* by introducing two other related tasks; POS and type levels (shown in Figure 2). The reason we apply multi-task learning is to try to exploit the observation that actual database tags of the tokens in the query are related to POS tags. Besides, multi-task learning helps to increase model accuracy and efficiency by making more generalized models with the help of shared representations between tasks [7]. POS and Type tasks are trained with schema task to improve accuracy of schema (final) tags. For each task, we define the same loss function, described above. During backpropagation, we simply combine the losses as follows;

$$L_{total} = \sum_{i=1}^{3} w_i \times L_i \text{ subject to } \sum_{i=1}^{3} w_i = 1 \tag{4}$$

where w_i represents the weight of i^{th} task and L_i represents the loss calculated for the i^{th} task similarly.

Another technique we integrate into the neural architecture is *skip-connection*. Skip connection is used to introduce extra node connections between different layers by skipping one or more layers in the architecture.With skip connections, the model provides an alternative for gradient to back propagation, which eventually helps in convergence. The technique has become compulsory component in many neural architectures deployed in computer vision community, such as the famous architectures ResNet [13] and DenseNet [17]. In the architecture of DBTagger, for each task except the first one (POS), we additionally feed the output of uni-directional GRU layer of previous task into CRF layer of the next task ($i + 1^{th}$ task). With these connections, we further carry the information of previous tasks to later tasks and eventually to the final task, schema tagging.

2.3 Annotation Scheme

In our problem formulation, every token (words in the natural language query) associates three different tags; namely part-of-speech (POS) tag, type tag and schema tag. In the following subsections, we explain how we extract or annotate each of them in detail.

2.3.1 POS Tags. To obtain the POS tags of our natural language queries we used the toolkit of Stanford Natural Language Processing Group named Stanford CoreNLP[27]. We use them as they are output from the toolkit, without doing any further processing since the reported accuracy for POS Tagger (97%) is sufficient enough.

2.3.2 Type Tags. In each natural language query, there are keywords (words or consecutive words) which can be mapped to database schema elements such as table, attribute or value. We divide this mapping into two levels; type tagging and schema tagging. Type tags represent the type of the mapped schema element to be used in the SQL query. In total we have seven different type tags;

- **TABLE**: NLQs contain nouns which may inhibit direct references to the tables in the schema, and we tag such nouns with *TABLE* tag. In the example NL query given in Table 1, noun *movie* has a type tag as TABLE, which also supports the intuition that schema labels and pos tags are related.
- **TABLEREF**: Although the primary sources for table references are nouns, some verbs contain references to the tables most of which are relation tables. TABLEREF tag is used to identify such verbs. Revisiting the example given Table 1, the verb *acted* refers to the table *cast*, and therefore it is tagged with TABLEREF to differentiate better the roles of POS tags in the query.
- **ATTR**: In SQL queries, attributes are mostly used in SELECT, WHERE and GROUP BY clauses. Natural language queries may contain nouns that can be mapped to those attributes. We use ATTR tag for tagging such nouns in the natural language queries.
- **ATTRREF**: Like TABLEREF tag, ATTRREF tag is used to tag the verbs in the natural language query that can be mapped to the attributes in the SQL query.
- **VALUE**: In NLQs, there are many entity like keywords that need to be mapped to their corresponding database values. These words are mostly tagged as *Proper noun-NNP* such as the keyword *John Nash* in the example query. In addition to these tags, it is also likely for a word to have a *noun-NN* POS tag with a *Value* tag corresponding to schema level. In order to handle these cases having different POS tags, we have *Value* type tags (e.g., *Mind* keyword in the example query is part of a keyword that needs to be mapped as *value* to *movie.title*). Keywords with *Value* tags can later be used in the translation to determine "where" clauses in SQL.
- **COND**: After determining which keywords in the query are to be mapped as values, it is also important to identify the words that imply which type of conditions to be met for the SQL query. For that purpose, we have the *COND* type tag.
- **O (OTHER)**: This type of tag represents words in the query that are not needed to be mapped to any schema instrument related to the translation step. Most stop words in the query (e.g., the) fall into this category.

2.3.3 Schema Tag. Schema tags of keywords represent the database mapping that the keyword is referring to; name of a table, or attribute. Tagging a keyword with a type tag is important yet incomplete. To find the exact mapping the keyword refers to, we defined a second level tagging where the output is the name of the tables or attributes. For each entity table (e.g. *movie* table in Figure 1) and for each non-PK or non-FK attribute (attributes which have

Table 1: An example NL query with its tags corresponding to each word in three different levels

NL query	who	acted	John	Nash	in	the	movie	A	Beautiful	Mind
POS tags	WP	VBD	NNP	NNP	IN	DT	NN	DT	JJ	NN
Type tags	O	TABLEREF	VALUE	VALUE	COND	O	TABLE	VALUE	VALUE	VALUE
Schema tags	O	cast	cast.role	cast.role	cond	O	movie	movie.title	movie.title	movie.title

Table 2: Statistics of the databases used

	Database			**Spider**				
Properties (#)	imdb	mas	yelp	academic	college	hr	imdb	yelp
entity tables	6	7	2	7	5	6	6	2
relation tables	11	5	5	8	2	1	11	5
total tables	17	12	7	15	7	7	17	7
total attributes	55	28	38	42	43	35	55	38
nonPK-FK attributes	14	7	16	18	29	21	14	16
total tags	31	19	20	26	36	30	31	20
queries	131	599	128	181	164	124	109	110
tokens in queries	1250	4483	1234	2127	2130	2099	1012	1035

semantics) we define a schema tag (e.g *movie, people, movie.title*, etc., referring to Figure 1). We complete possible schema tags by carrying *OTHER* and *COND* from type tags. We use the same schema tag for attributes and values (e.g *movie.title*), but differentiate them at the inference step by combining tags from both type tags and schema tags. If a word is mapped into *Value* type tag as a result of the model, its schema tag refers to the attribute in which the value resides.

In order to annotate queries, we annotate each word in the query for three different levels mentioned above. While POS tags are extracted automatically, we manually annotate the other two levels. Annotations were done by three graduate and three undergraduate computer science students who are familiar with database subject. Although annotation time varies depending on the person, on the average it took a week to annotate tokens by a single person for two levels (type and schema) for a query log with 150 NL questions, which we believe is practical to apply in many domains.

3 EXPERIMENTAL EVALUATION

3.1 Datasets

In our experiments we used *yelp, imdb [35]*, and *mas [24]* datasets which are heavily used in many NLIDB related works by the database community [2, 24, 32, 35]. In addition to these datasets, we also used different schemas from the *Spider* dataset [42]; which are *academic, college, hr, imdb*, and *yelp*. Spider is comprised of approximately 200 schemas from different domains; however, there are only handful (around 10) of schemas with more than 100 NL questions. Number of questions is important for our deep learning based solution, since it requires certain number of training data to effectively train. Each schema we picked from Spider dataset is among the schemas with most number of NL questions, having over 100 queries to work with. Due to the lack of sufficient database values (many schemas do not have database rows or have few number of rows), we used the Spider dataset only on supervised setup.

The statistics about each dataset for which annotation is done is shown in Table 2. In Table 2 (referring to Figure 1), entity tables refer to main tables (i.e. Movie), relation tables refer to hub tables that store connections between entity tables (i.e. cast, written_by), nonPK-FK attributes refer to attributes in any table that is neither PK nor FK (i.e., gender in People table), and finally total tags refer to unique number of taggings extracted from that particular schema depending on the above mentioned values. Final schema tags of a particular database are determined by composing table names and name of the nonPK-FK attributes in addition to COND and OTHER. In the last two rows of the Table 2, we show annotated number of NL questions, referred to as queries, and the number of total words inside these queries, referred to as tokens.

3.2 Settings

We first split the datasets into train-validation sets with 5 − 1 ratio, respectively to be used for tuning task weights. For models trained on multiple tasks, we used 0.1 − 0.2 − 0.7 as tuned weights for POS, Type and Schema tasks, respectively.

We train our deep neural models using the backpropagation algorithm with two different optimizers; namely Adadelta [43] and Nadam [10]. We start the training with Adadelta and continue it with Nadam. We found that using two different optimizers resulted better in our problem. For both shared and unshared bi-directional GRUs, we use 100 units and apply dropout [15] with the value of 0.5 including recurrent inner states as well. For training, the batch-size is set to 32 for all datasets. Parameter values chosen are similar to that reported in the study [23] (the state-of-the-art NER solution utilizing deep neural networks), such as the dropout and batch size values. We measure the performance of each neural model by applying cross validation with 6-folds. All the results reported are

Table 3: Accuracy scores of unsupervised baselines for relation and non-relation matching

	Database		
Baseline	imdb	mas	yelp
tf-idf	0.594-0.051	0.734-0.084	0.659-0.557
NALIR	0.574-0.103	0.742-0.476	0.661-0.188
word2vec	0.625-0.093	0.275-0.379	0.677-0.269
TaBERT	NA-0.251	NA-0.094	NA-0.114
DBTagger	0.908-0.861	0.964-0.950	0.947-0.923

the average test scores of 6-folds. During inference, we discard POS and Type task results and only use Schema (final) tasks to measure scores.

3.3 Results

3.3.1 Comparison with Unsupervised Baselines. We implemented the unsupervised approaches utilized in the state-of-the art NLIDB works for the keyword mapping task as baselines to compare with DBTagger.

- **tf-idf:** Similar to ATHENA [32], for each unique value present in the database, we first create an exact matching index, and then perform tf-idf for tokens in the NLQ. In case of matches to multiple columns, the column with the biggest tf value is chosen as matching. In order to handle multi word keywords, we use n-grams of tokens up to $n = 3$. For relation matching, we used lexical similarity based on the Edit Distance algorithm.
- **NALIR:** NALIR [24] uses WordNet for relation matching. For non-relation matching,it utilizes regex or full text search queries over each database column whose type is text. In case of matches to multiple columns, the column which returns more rows as a result is chosen as matching. For fast retrieval, we limit the number of rows returned from the query to 2000, as in the implementation of NALIR.
- **word2vec:** For each unique value present in the database, cosine similarity over tokens in the NLQ is applied to find mappings using pre-defined wor2vec embeddings. The matching with the highest similarity over a certain threshold is chosen.
- **TaBERT:** TaBert [38] is a transformer based encoder which generates dynamic word representations (unlike word2vec) using database content. The approach also generates column encoding for a given table, which makes it an applicable keyword mapper for non-relation matching.For a particular token, matching with maximum similarity over a certain threshold is chosen.

We categorize the keyword mapping task as *relation matching* and *non-relation matching*. The former mapping refers to matching for table or column names and the latter refers to matching for database values. For fair comparison, we do not apply any pre or post processing over the NL queries or use external source of knowledge, such as a keyword parser or metadata extractor. Results are shown in Table 3. Each pair of scores represents token wise accuracy for relation and non-relation matching.

DBTagger outperforms unsupervised baselines in each dataset significantly, by up to 31% and 65% compared to best counterpart for relation and non-relation matching, respectively. For relation matching, results of all approaches are similar to each other except the word2vec method for the mas dataset. The main reason for such poor performance is that the mas dataset has column names such as *venueName* for which word2vec cannot produce word representations, which radically reduces chances of semantic matching.

tf-idf gives promising results on the yelp dataset, whereas it fails on the imdb and mas datasets for non-relation matching. This behavior is due to presence of ambiguous values (the same database value in multiple columns) and not being able to find a match for values having more than three words. For the *imdb* dataset, none of the baselines performs well for non-relation matching. The *imdb* dataset has entity like values that are comprised of multiple words such as movie names, which makes it impossible for semantic matching approaches to generate meaningful representations to perform similarity. NALIR's approach of querying over database has difficulties for the imdb and yelp datasets since the approach does not solve ambiguities without user interaction. TaBERT performs poorly for all datasets. TaBERT has its own tokenizer, which tries to deal with tokens that are out of vocabulary (OOV) by breaking the token into sub-words that have representations. OOV keywords appearing in the natural language query are therefore divided by the tokenizer into pieces, which eventually leads to unrelated word representations and therefore non-predictive similarity calculation.

Table 4: Translation Accuracy

	Database		
NLIDB System	imdb	mas	yelp
NALIR	0.383	0.330	0.472
TEMPLAR (on NALIR)	0.500	0.402	0.528
DBTagger Pipeline	0.564	0.551	0.461

3.3.2 Translation Accuracy. In order to show the effectiveness of tags output by DBTagger, we implemented a simple translation pipeline, similar to methodology in [2]. The pipeline generates join paths for SQL translation using shortest length path over schema graph to cover all the mappings output by DBTagger. We count inaccurate, if the algorithm can not output a joining path. We compare our pipeline with a state-of-the-art system, NALIR[24], and TEMPLAR[2], which is an enhancer over an existing NLIDB system. The results are presented in Table 4. The pipeline over DBTagger tags outperforms both systems in imdb and mas datesets, up to 66% and 37% compared to NALIR and TEMPLAR respectively. For queries which do not include nested or group by constraints such as simple select-join queries, our pipeline produces 67%, 77% and 53% translation accuracy for imdb, mas and yelp datasets respectively. Considering the simplicity of the translation algorithm, results demonstrate the efficacy of predicted outputs of DBTagger.

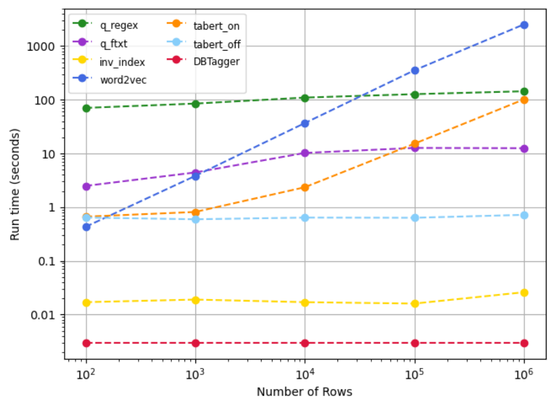
3.4 Impact of DBTagger Architecture

In this experimental setup, we perform keyword mapping in a supervised fashion with different neural network architectures along with a non-Deep Learning (DL) baseline to evaluate architectural decisions.

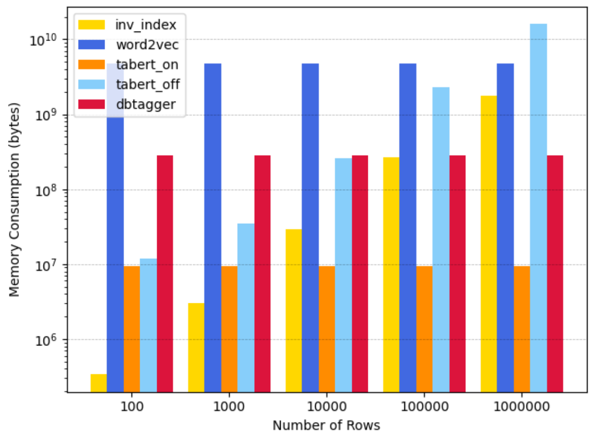
- **CRF:** As a non-DL baseline, we use vanilla CRF. Semantic word representations of the NLQ are fed as input to the model.

Table 5: Performance of Neural Models with Different Architectures in accuracy-F1 metrics

	Database			Spider				
Model	yelp	imdb	mas	academic	hr	college	imdb	yelp
CRF	0.934-0.890	0.907-0.850	0.955-0.932	**0.974-0.956**	**0.881-0.748**	0.878-0.721	0.866-0.821	0.880-0.827
ST_Uni	0.939-0.883	0.905-0.805	0.961-0.938	0.962-0.945	0.844-0.642	0.854-0.692	0.848-0.751	0.865-0.803
ST_Bi	0.947-0.908	0.917-0.832	0.964-0.941	0.966-0.952	0.877-0.689	0.872-0.720	0.882-0.811	0.891-0.841
MT_Seq	0.938-0.886	0.921-0.853	0.964-**0.943**	0.964-0.952	0.835-0.685	0.886-0.714	0.896-0.837	0.895-0.838
DBTagger	**0.968-0.938**	**0.935-0.878**	**0.965**-0.941	0.965-0.954	0.861-0.735	**0.904-0.761**	**0.898-0.855**	**0.897-0.854**



(a) Run Time



(b) Memory Usage

Figure 3: Run Time and Memory Usage of state-of-the-art keyword mapping approaches

- **ST_Uni:** We create a two layers stack of uni-directional GRUs, followed by CRF as the classification layer. This model is trained on only a single task, schema tags.
- **ST_Bi:** Different than the previous architecture, we use bi-directional GRUs instead of uni-directional GRUs. Classification is done on the CRF layer.
- **MT_Seq:** In this model, training is performed on all three tasks. However, each task is trained separately. The predicted tag of the previous task is fed into the next task. To do that, 1-hot vector representations of predicted tags are concatenated with semantic word representations. We stack a bi-directional GRU with a uni-directional GRU to encode the sentence and feed the output vector to the CRF layer.
- **DBTagger:** This model represents the DBTagger architecture where all tasks are used during training concurrently. DBTagger also has cross-skip connections between tasks as depicted in Figure 2.

For all the models, the same hyper parameters are used for fair comparison during training, as explained in Section3.2. The results are shown in Table 5. Each pair of scores represents the accuracy and F1 measures, respectively. DBTagger performs better than the other supervised architectures for six different datasets in accuracy and in terms of F1. Especially for the yelp and college datasets the performance improvement is remarkable, which is up to around 4.5% and 5%, respectively. Vanilla CRF performs well among all (best in two datasets), which signifies its role in the architecture for the sequence tagging problem. ST_Bi performs better than ST_Uni in all datasets, which shows the positive impact of bi-directional GRUs. Compared to single task models, multi task models perform better for all datasets. Except the *mas* dataset for the F1 metric, DBTagger produces better tags compared to the other multi task model, MT_Seq, in which tasks are trained separately.

3.5 Efficiency Analysis

Efficiency is one of the most important properties of a good keyword mapper to be deployable in online interfaces. Therefore, run time performance of keyword mapping approaches mentioned in Section 3.3 is also evaluated.

- **NALIR**: We analyze both querying over database column approaches used in NALIR[24], named as *q_regex* and *q_ftext*, which use *like* and *match against* operators respectively.
- **tf-idf**: Similar to ATHENA [32], we created an exact matching index, using inverted index named as *inv_index*, beforehand to avoid querying over database, .
- **word2vec**: Many works such as Sqlizer [35] make use of pre-trained word embeddings to find mappings, which requires keeping the model in the memory to perform similarities.
- **tabert_on**: TaBert requires database content (content snapshot) to generate encodings for both NL tokens and columns. We call

this setup tabert online, where the model generates the content snapshot to perform mapping when the query comes.

- **tabert_off**: We also use TaBert in offline setup. For each table, database content is generated beforehand to perform encodings. In this setup, we keep the content in the memory to serve the query faster.

We measured the time elapsed for a single query to extract tags and the memory consumption needed to perform mapping for each approach. We also run each experiment with different number of row values to capture the impact of the database size. Figure 3 presents run time and memory usage analysis of keyword mappers. DBTagger ouputs the tags faster than any other baseline and it is scalable to much bigger databases. However, q_regex, q_ftext, tabert_on and word2vec do not seem applicable for bigger tables having more than 10000 rows. The tf-idf technique has nice balance between run-time and memory usage, but it is limited in terms of effectiveness (Table 3). tabert-off performs the tagging in a reasonable time, yet it requires huge memory consumption especially for bigger tables.

4 RELATED WORK

4.1 NLIDBs and Keyword Mapping Approaches

Although the very first effort [14] of providing natural language interface in databases dates back to multiple decades ago, the popularity of the problem has increased due to some recent pipeline based systems proposed by the database community, such as SODA [4], NALIR [24], ATHENA[32] and SQLizer[35].

Recently, end-to-end approaches utilizing encoder-decoder based architectures [3, 5, 12, 18, 33, 34, 37, 40, 41, 44] in deep learning have become more popular to deal with the translation problem. Seq2SQL[44] uses a Bi-LSTM to encode a sequence that contains columns of the related table, SQL keywords and question. The study [44] also provided a dataset called *WikiSql* to the research community working on NLIDB problem for evaluation. SQLNet[34] defines a seq-to-set approach to eliminate reinforcement learning process of Seq2SQL. In another study which used WikiSql dataset, Yavuz et al.[37] employs a process called candidate generation to create keyword mappings to be used in *where* clasue in SQL translation specifically. All of the proposed deep learning based methods use pre-trained word embedding vectors for input to the model. Therefore, keyword mapping is implicitly handled by the model. However, TypeSql [40] tries to enrich input data augmenting entity tags by performing similarity check over the database or knowledge base. Similarly, [18] tries to find possible constant values in the query by performing similarity matching.

Due to the limited nature of WikiSql dataset, having a single table for each database, another important dataset called Spider [42] is provided to the community. Consequently, many studies proposed recently [5, 12, 33, 38, 41] have evaluated their solutions on the Spider dataset. Different from the others, TaBERT [38] as a transformer based encoder, makes use of database content to generate dynamic representations along with contextual encodings to represent database columns. For a comprehensive survey covering existing solutions in NLIDB, the reader can refer to [1, 21].

Similar to our work, Baik et al. [2] propose TEMPLAR, to be augmented on top of existing NLIDB systems to improve keyword mapping and therefore translation using query logs. Though, TEMPLAR is not a standalone mapper, since it requires from a NLIDB system multiple preliminaries to function properly, including parsed keywords and associated metadata with each keyword, which are the main challenges yielded by the keyword mapping problem. Therefore, the mapper cannot be plugged into NLIDB pipelines that does not perform detailed keyword recognition and parsing.

Different from the previous works, DBTagger is an end-to-end keyword mapper, which does not require any processing or external source of knowledge. Also, to the best of our knowledge, our work is the first study utilizing deep neural networks in a supervised learning setup for keyword mapping.

4.2 Deep Learning Solutions for Sequence Tagging in NLP

In NLP community, neural network architectures have been utilized in many research problems. As a pioneer in the field, Collobert et al. [9] proposed Convolutional Neural Networks (CNN) based architecture with CRF layer on top to deal with the sequence tagging problem. Yao et al. [36] applied LSTM in sequence tagging without having CRF as the classification layer. Bi-directional RNN structure was employed first in a speech recognition problem in [11].

Later, instead of simple RNN networks, bi-directional LSTM was adopted and employed by Huang et al. [19] in NER problem. Following that study, Lample et al. [23] proposed a similar architecture with the inclusion of word and character embeddings. They used pre-trained word embeddings along with character level embeddings to extract input matrix to feed into the network. Their study stand as the state-of-the-art in sequence tagging problems in NLP. Similar to [23], Ma and Hovy [26] proposed a neural architecture where character embeddings is done through CNN instead of LSTM. For a comprehensive survey discussing the deep learning solutions for research problems in NLP community, [39] is a great read.

5 CONCLUSION AND FUTURE WORK

In this paper, we present DBTagger, a keyword mapper to be used in translation pipelines in NLIDB systems. DBTagger is a standalone system which does not require any processing or external knowledge such as parser or metadata preliminaries. Inspired by sequence tagging architectures used for well known problems such as POS in the NLP community, DBTagger utilizes a deep neural architecture based on bi-directional GRUs. DBTagger provides the best accuracy results on three publicly available databases and five schemas in Spider dataset, producing keyword tags with 92.4% accuracy on the average over all the datasets within 3 milliseconds, which is 10000 times faster than unsupervised approaches. Our results also show that DBTagger is scalable to large databases containing millions of rows. We believe that DBTagger can be applied in existing NLIDB systems as the first step to improve translation, especially in pipeline-based systems. For the deep learning based approaches, DBTagger can be utilized to be augmented on neural network to enrich input query before feeding into network.

ACKNOWLEDGMENTS

This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under the grant no 118E724.

REFERENCES

[1] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. 2019. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal* 28, 5 (2019), 793–819.

[2] Christopher Baik, H. V. Jagadish, and Yunyao Li. 2019. Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE '19)*.

[3] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. DBPal: A Learned NL-Interface for Databases. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1765–1768.

[4] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. 2012. SODA: Generating SQL for Business Users. *Proc. VLDB Endow.* 5, 10 (2012).

[5] Ben Bogin, Jonathan Berant, and Matt Gardner. 2019. Representing Schema Structure with Graph Neural Networks for Text-to-SQL Parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL '19)*. 4560–4565.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[7] Rich Caruana. 1997. Multitask learning. *Machine learning* 28, 1 (1997), 41–75.

[8] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated Feedback Recurrent Neural Networks. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML'15)*. 2067–2075.

[9] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *Journal of machine learning research* 12, ARTICLE (2011), 2493–2537.

[10] Timothy Dozat. 2016. Incorporating Nesterov Momentum into Adam. In *International Conference on Learning Representations Workshop*.

[11] A. Graves, A. Mohamed, and G. Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 6645–6649.

[12] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL '19)*. 4524–4535.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*. 770–778.

[14] Gary G. Hendrix, Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. 1978. Developing a Natural Language Interface to Complex Data. *ACM Trans. Database Syst.* 3, 2 (1978), 105–147.

[15] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv* abs/1207.0580 (2012).

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17)*. 4700–4708.

[18] Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen-tau Yih, and Xiaodong He. 2018. Natural Language to Structured Query Generation via Meta-Learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers) (NAACL '18)*. 732–738.

[19] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. *ArXiv* abs/1508.01991 (2015).

[20] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL '17)*. 963–973.

[21] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment* 13, 10 (2020), 1737–1750.

[22] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML '01)*. 282–289.

[23] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL '16)*. 260–270.

[24] Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proc. VLDB Endow.* 8, 1 (2014), 73–84.

[25] Yunyao Li and Davood Rafiei. 2017. Natural Language Data Management and Interfaces: Recent Development and Open Challenges. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1765–1770.

[26] Xuezhe Ma and Eduard Hovy. 2016. End-to-end Sequence Labeling via Bidirectional LSTM-CNNs-CRF. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL '16)*. 1064–1074.

[27] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*. 55–60.

[28] Tomas Mikolov, G.s Corrado, Kai Chen, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the International Conference on Learning Representations (ICLR'13)*. 1–12.

[29] George A. Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[30] Fatma Özcan, Abdul Quamar, Jaydeep Sen, Chuan Lei, and Vasilis Efthymiou. 2020. State of the Art and Open Challenges in Natural Language Interfaces to Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 2629–2636.

[31] 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 (IUI '03)*. 149–157.

[32] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *Proc. VLDB Endow.* 9, 12 (2016).

[33] Nathaniel Weir, Prasetya Utama, Alex Galakatos, Andrew Crotty, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Nadja Geisler, Benjamin Hättasch, Steffen Eger, Ugur Cetintemel, and Carsten Binnig. 2020. DBPal: A Fully Pluggable NL2SQL Training Pipeline. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 2347–2361.

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

[35] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[36] Kaisheng Yao, Baolin Peng, Yu Zhang, Dong Yu, Geoffrey Zweig, and Yangyang Shi. 2014. Spoken language understanding using long short-term memory neural networks. In *2014 IEEE Spoken Language Technology Workshop (SLT)*. 189–194.

[37] Semih Yavuz, 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 (EMNLP '18)*. 1702–1711.

[38] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL '20)*. 8413–8426.

[39] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. 2018. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence Magazine* 13, 3 (2018), 55–75.

[40] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. 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, Volume 2 (Short Papers) (NAACL '18)*. 588–594.

[41] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. 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 (EMNLP '18)*. 1653–1663.

[42] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP '18)*. 3911–3921.

[43] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *ArXiv* abs/1212.5701 (2012).

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