The VLDB Journal
https://doi.org/10.1007/s00778-023-00809-w

REGULAR PAPER

xDBTagger: explainable natural language interface to databases using keyword mappings and schema graph

Arif Usta[1] · Akifhan Karakayali[2] · Özgür Ulusoy[3]

Received: 10 October 2022 / Revised: 26 April 2023 / Accepted: 31 July 2023

Abstract
Recently, numerous studies have been proposed to attack the natural language interfaces to data-bases (NLIDB) problem by researchers either as a conventional pipeline-based or an end-to-end deep-learning-based solution. Although each approach has its own advantages and drawbacks, regardless of the approach preferred, both approaches exhibit black-box nature, which makes it difficult for potential users to comprehend the rationale behind the decisions made by the intelligent system to produce the translated SQL. Given that NLIDB targets users with little to no technical background, having interpretable and explainable solutions becomes crucial, which has been overlooked in the recent studies. To this end, we propose xDBTagger, an explainable hybrid translation pipeline that explains the decisions made along the way to the user both textually and visually. We also evaluate xDBTagger quantitatively in three real-world relational databases. The evaluation results indicate that in addition to being lightweight, fast, and fully explainable, xDBTagger is also competitive in terms of translation accuracy compared to both pipeline-based and end-to-end deep learning approaches.

Keywords Natural language interface for databases · NLIDB · Text-to-SQL · Multi-task learning · Explainable artificial intelligence · XAI

1 Introduction

SQL is used as a standard tool to extract data out of a relational database. 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 in using SQL. Consequently, casual users find it even more challenging to express their information needs, which makes SQL less desirable. To remove this barrier, an ideal solution is to provide a search engine-like interface in databases. The goal of *NLIDB* is to break through these barriers to make it possible for casual users to employ their natural language to extract information.

Recently, many works have been developed attacking the NLIDB problem; such as conventional pipeline-based approaches [3, 35, 43, 45, 53] or end-to-end deep-learning-based approaches [19, 28, 36, 44, 50, 52, 57, 60]. Neural network-based solutions seem promising in terms of translation accuracy and robustness, covering semantic variations of queries. However, they struggle with 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 pipeline-based or hybrid solutions still an attractive alternative. [40].

Whether it is a pipeline-based or an end-to-end deep learning approach, existing solutions have black-box nature when it comes to outputting translated SQL. Being a black-box solution makes it difficult for users to understand how the result SQL is produced along the way, which is a vital defectiveness for any modern intelligent system that should aim to gain the trust of the users [20]. This undesirable property of NLIDB solutions becomes much more consequential in an

✉ Arif Usta
arif.usta@uwaterloo.ca

Akifhan Karakayali
akifhan.karakayali@tcmb.gov.tr

Özgür Ulusoy
oulusoy@cs.bilkent.edu.tr

1 University of Waterloo, Waterloo, ON, Canada

2 The Central Bank of the Republic of Türkiye, Ankara, Turkey

3 Bilkent University, Ankara, Turkey

Published online: 23 August 2023

online scenario, especially for casual users (i.e., users with little to no technical expertise in SQL), who are the primary potential audience targeted by the problem of NLIDB.

In a recently published survey covering NLIDB, authors asserted that translation is only one part of an ideal interface, and hence researchers must complement these solutions with other problems such as query explanation in order to effectively serve all users with diverse technical background [30]. Although NLIDB is a well-studied problem in the literature, the transparency and explainability of the proposed solutions have been overlooked. An intelligent system such as an NLIDB solution has to be transparent and self-explanatory to the user so that they can comprehend the decisions made by the system. As highlighted by many previous studies [17, 18, 20, 41], having an explainable intelligent system exhibits many benefits including but not limited to improving users' trust in the system, helping users understand the decisions made by the system, and showing the limitations of the systems for certain use-cases, all of which can be instrumental towards developing more user friendly and preferable NLIDB systems.

Consider the below pair of NL query and SQL translation from a movie database domain to understand better how an explainable NLIDB solution would be handy and, in fact, essential for users to reason with the results:

NL Query 1 *Who is the director of the series House of Cards produced by Netflix?*

Translated SQL 1
```
Select * From
tv_series, copyright, company,
directed_by, director
Where (tv_series.msid = copyright.msid)
and (copyright.cid = company.cid)
and (tv_series.msid = directed_by.msid)
and (directed_by.did = director.did)
and (tv_series.title = "House of Cards")
and (company.name = "Netflix")
```

Although such a SQL query is easy to understand for an expert user at a glance, it is difficult to interpret for casual users, which are the ones targeted by the NLIDB problem. In the above query, there are 2 keywords, *House of Cards* and *Netflix*, found under the attributes *title* and *name* of their respective entity tables *tv_series* and *company*. However, to find the corresponding tuple(s) matching these two values from different tables, SQL requires a join operation; in this case, a 5-way join. The subsequent four lines after the *Where* clause in the above example represent conditions to ensure the right join. More importantly, although one needs access to 3 entity tables; 2 (tv_series and company) for utterances found in the query and 1 (director) for the desired information asked by the user, SQL requires two more intermediate tables, *copyright* and *directed_by*, to complete the join. In an ideal NLIDB, the story behind the translated SQL, such as above, should be provided to the user to some extent. In addition to the explanations needed for understanding SQL structure, the NLIDB system should also ideally give explanations for how it matches schema elements (e.g., three tables and two attributes for the above example) to the corresponding utterances.

To address above-mentioned concerns, we propose an explainable, end-to-end NLIDB solution, **Explainable DBTagger (xDBTagger)**, by extending our previous work in [47][1]. We embrace a holistic approach to propose a complete solution which is efficient, scalable, and explainable to serve all types of users regardless of their background in addition to being competitively effective. xDBTagger is a hybrid solution utilizing both deep learning and rule-based approaches. To the best of our knowledge, xDBTagger is the first study exercising explainable artificial intelligence (XAI) paradigm in the NLIDB problem. In what follows, we list the main contributions of our work:

- We use our previous work [47], which is a deep learning model specifically tailored for sequence tagging in NLIDB, to extract keyword mappings given the NLQ.
- We propose a novel wrapper tailored for sequence tagging problems around a state-of-the-art XAI work, LIME [42], to explain decisions made for keyword mappings output for each token in NLQ. We provide explanations for each keyword mapping corresponding to tokens in NLQ by highlighting both the positive and negative contributions of each surrounding token.
- We propose an effective and efficient SQL translation algorithm suitable for explainability by utilizing keyword mappings and schema graphs extending our previous work in [47] by designing heuristics tailored to address complex queries involving aggregates. We provide textual and visual explanations for the user to comprehend how the translation algorithm works.
- We quantitatively evaluate the entire pipeline in three publicly available datasets against both pipeline-based and end-to-end deep learning approaches.
- We deploy xDBTagger in a user-friendly and interpretable interface in which the user is presented the translated SQL along with the explanations for the decisions made throughout the pipeline.

The remainder of the paper is organized as follows. In the next section, we give an overview of the system architecture of xDBTagger. Section 3 presents the neural network structure we design for the keyword mapping step. We explain how we modify LIME to produce explanations for keyword mappings output by DBTagger in Sect. 4. We thoroughly

[1] Available at https://github.com/arifusta/DBTagger

review the main components of the SQL extraction algorithm in Sect. 5. In Sect. 6, we provide quantitative experimental results for both the keyword mapper and the entire SQL translation pipeline (Sect. 6.2). In addition to quantitative results, we illustrate the user interface and provide examples of textual and visual explanations in the interface (Sect. 6.3). We summarize the related work and conclude the paper in Sects. 7 and 8, respectively.

2 System architecture

Figure 1 depicts an overview of the translation pipeline along with explanation components to make the decision-making throughout the pipeline interpretable. The workflow starts with an input NLQ from the user. The query first goes through pre-processing, which removes special characters and punctuations such as commas and quotes. After removing those characters, the query is tokenized into words using spaces. These tokens are then converted to 300-dimensional vector representations using a pre-trained word embedding model. In our implementation, we used a pre-trained fast-text [4] model.

The output of the embedding model $X = [x_1, x_2, ..., x_n]$ is an array of 300-dimensional vectors where n is the length of the query (i.e., the number of tokens in the NLQ). Each 300-dimensional vector is a representation of each word in the original query. Following that, X is fed as the input to DBTagger model, which outputs corresponding keyword mappings for each token in the query. DBTagger outputs 2 series of outputs; 1 for type tags (i.e., schema element such as table, attribute if the token is relevant for SQL translation or "O" if irrelevant) and 1 for schema tags (i.e., deeper level tags such as name of a table or an attribute) of the tokens in the NLQ.

We use LIME [42] to explain keyword mappings output by DBTagger. LIME requires a black-box model which can output prediction probabilities and raw input text for explanation. To satisfy these conditions, we construct a DBTagger Model Blackbox, which takes the raw text as input and gives predictions with probabilities as output. Vanilla LIME tries to explain a single classified output given a sequential input text. However, it is not the case in our problem (i.e., sequence tagging problem requiring a classification and, therefore, an explanation for each token in the query). To alleviate this issue, we implement a wrapper around LIME which is explained in Sect. 4.1. This wrapper takes DBTagger Model Blackbox, raw text input, and predicted output tags to produce an explanation for each token.

3 Keyword mapper—DBTagger

In this section, we first provide background information about the neural network structure utilized for sequence tagging problems such as Part-of-Speech (POS) tagging and Name Entity Recognition (NER) in the NLP community. Next, we explain the network structure of *DBTagger*, our keyword mapper solution in the pipeline, by pointing out modifications we introduce on top of the state-of-the-art sequence tagging architecture to better capture characteristics of the keyword mapping problem in the scope of NLIDB. In particular, we describe how we utilize *skip connections* and *multi-task learning* to exploit the observation we made that POS tags are correlated to final mappings. Lastly, we discuss how we annotate three different class labels of tokens to employ multi-task training.

3.1 Deep sequence tagger architecture

POS tagging and NER refer to sequence tagging problems in NLP for a particular sentence to identify parts-of-speech such as noun, verb, and adjective and locate any entity names such as person and organization, respectively. Although both problems are similar to each other in terms of being a classification problem for each word in a sentence, they differ in such a way that for NER it is also vital to determine multi-word boundaries for the classification, which is what we reduce the keyword mapping problem to in NLIDB. Therefore, we borrow the deep neural network architectures employed for sequence tagging problems, such as NER, as a backbone in our architecture.

Recurrent neural networks (RNN) are at the core of architectures utilized in the previous studies dealing with sequence tagging problems [27, 34, 37] since they are a family of networks that perform well on sequential data input such as a sentence. In this particular problem, sequence tagging (*keyword mapping*), RNNs are employed to output a sequence of labels for the original sentence (the query), input as a sequence of words.

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:

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

U and W are constants representing weights that are updated during training. 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 [24] and GRU [7]. Compared to vanilla RNN, LSTM has *forget gates* and GRU comprises of *reset* and *update* gates additionally. We experimented with both structures and chose GRU for its better performance in our experiments. In GRU, Update Gates decide what information

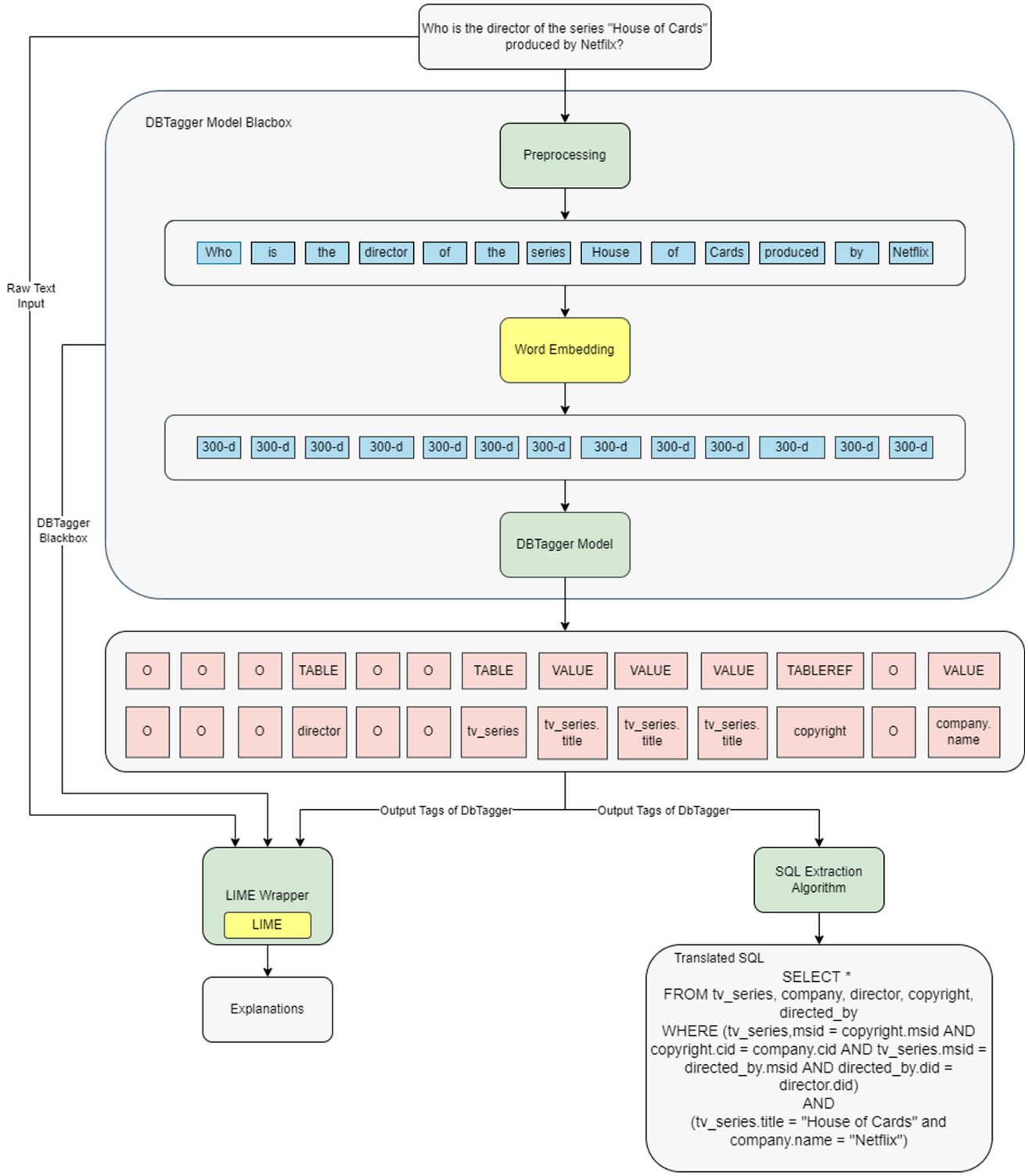


Fig. 1 System architecture of xDBTagger

to throw away and what new information to add, whereas Reset Gate is utilized to decide how much past information to forget. The calculation of GRU is as follows:

$$z = \sigma(U_z.x_t + W_z.h_{t-1}) \tag{2}$$
$$r = \sigma(U_r.x_t + W_r.h_{t-1}) \tag{3}$$
$$z_t = tanh(U_z.x_t + W_s.(h_{t-1} \cdot r)) \tag{4}$$
$$z_t = \sigma(U_z x_t + W_z h_{t-1}) \tag{5}$$

In the sequence tagging problem, in addition to past information, we also have future information 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 [16]. 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 Fig. 2.

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)* [33] is preferred. Unlike independent classification by softmax, CRF tries to predict labels sentence-wise by considering labels of the neighboring words as well. This feature of CRF is what makes it an attractive choice, especially in a problem like *keyword mapping*. This finding was also reported in [34], where authors claim that CRF as the output layer gives 1.79 more accuracy compared to the softmax layer in NER task. The final outlook of the architecture of deep sequence tagger is depicted in Fig. 2.

3.2 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 ith 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 yth 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, the 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 [4] due to it being one of the representation techniques considering sub-word (character n-grams) as well to deal with the out-of-vocabulary token problem better.

We consider G to be two-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, the last CRF layer defines tag scores as;

$$s(X, Y) = \sum_{i=1}^{n} A_{y_i, y_{i+1}} + \sum_{i=1}^{n} G_{i, y_i} \tag{6}$$

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, the probability of the sequence Y is calculated as:

$$p(Y|X) = \frac{e^{s(X,Y)}}{\sum_{\bar{Y} \in Y_x} e^{s(X,\bar{Y})}} \tag{7}$$

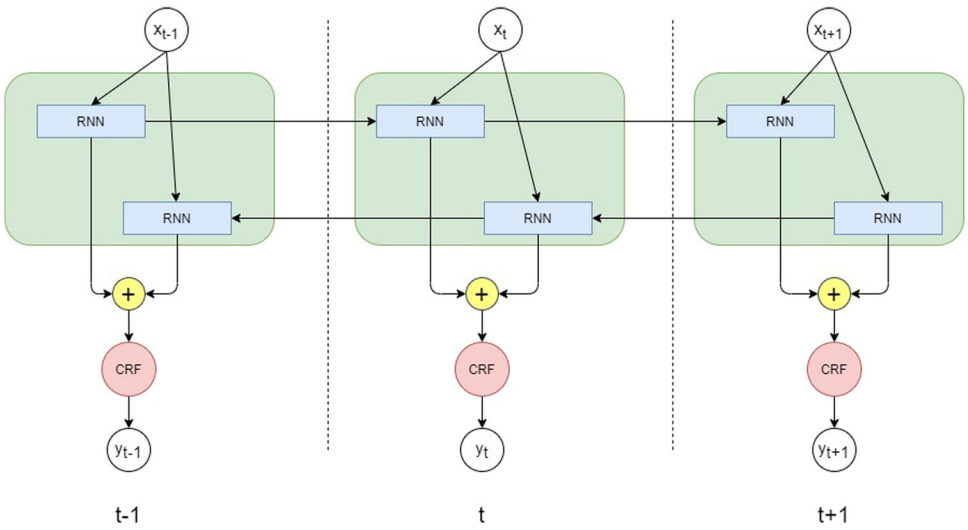


Fig. 2 Deep sequence tagger network

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 Fig. 3). 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 [6]. As a multi-task training paradigm, POS and Type tasks are trained together with schema task to improve the accuracy of schema (final) tags. The multi-task learning architecture is shown in Fig. 3. It has mainly two components; a shared encoder producing a common representation to be used in each task and a separate but same CRF layer to classify tokens according to each task. Therefore, we use the same loss function for each task as described above. To combine the losses, we employed a linear weighted sum technique, which is an intuitive and common way to combine multiple tasks [6, 10, 29].

To further exploit the information carried for individual tasks into other tasks and eventually into our final task (i.e., schema tags), we additionally utilize *skip-connection* paradigm. Skip connection is used to introduce extra node connections between different layers by skipping layers or directly forwarding an output to another unit in the architecture without applying non-linear activation functions to allow gradients to flow without exploding or vanishing. 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 the computer vision community, such as the famous architectures ResNet [21] and DenseNet [25]. In the architecture of DBTagger, for each task except the first one (POS), we additionally feed the output of the uni-directional GRU layer of the previous task into

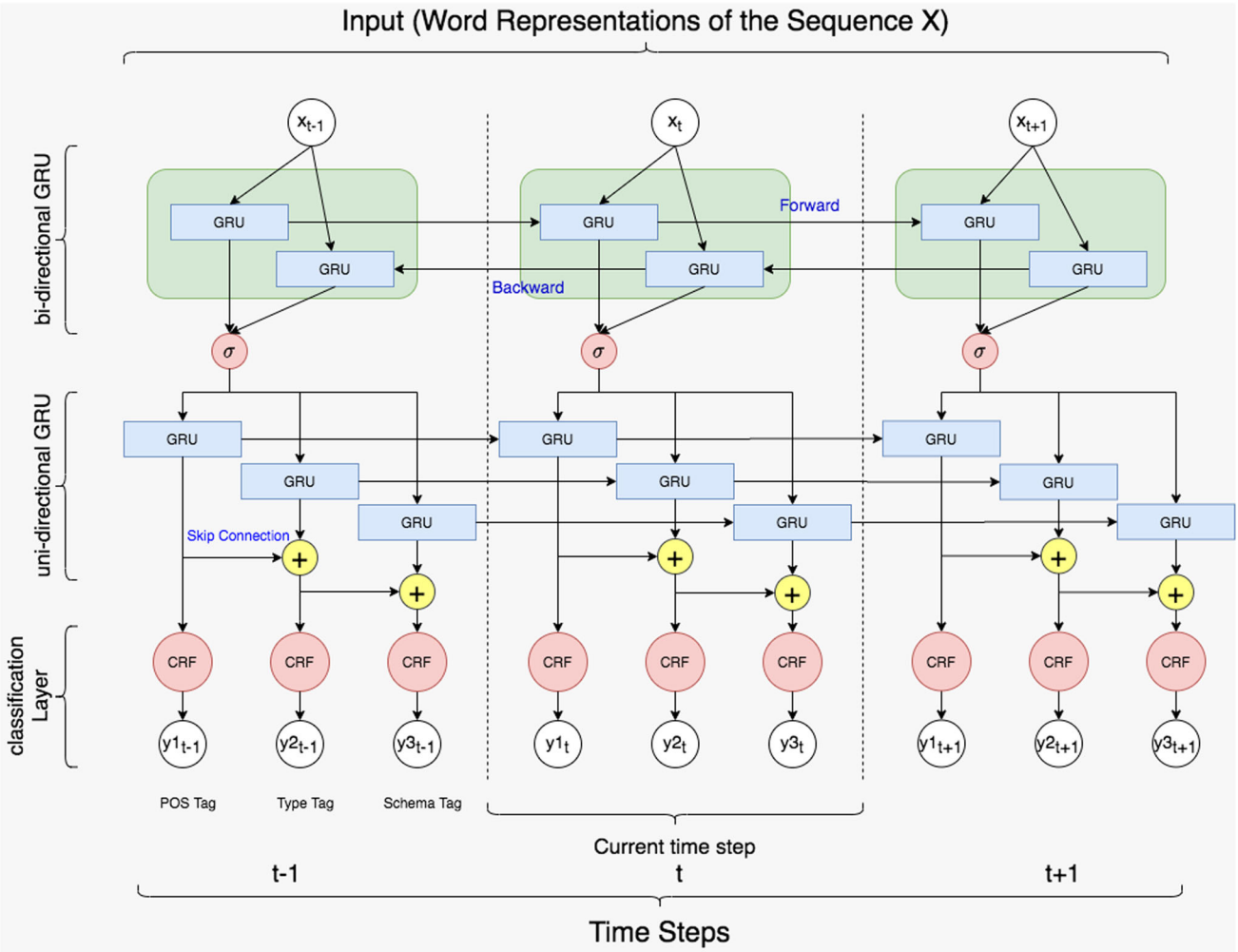


Fig. 3 DBTagger network

the 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.

3.3 Annotation scheme

We tackle keyword mapping as a sequence tagging problem, which is a supervised classification problem. In our problem formulation, every token (i.e., words in the natural language query) associates three different tags: 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.

3.3.1 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 [38]. 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.

3.3.2 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 1 An example NL query with its tags corresponding to each word in two target levels

NLQ	Type Tag	Schema Tag
Who	O	O
is	O	O
the	O	O
director	TABLE	director
of	O	O
the	O	O
series	TABLE	tv_series
House	VALUE	tv_series.title
of	VALUE	tv_series.title
Cards	VALUE	tv_series.title
produced	TABLEREF	copyright
by	O	O
Netflix	VALUE	company.name

- **TABLE**: NLQs contain nouns that 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 *director* 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 in Table 1, the verb *produced* refers to the table *copyright*, 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. For instance, in the query *"In what year was Benedict Cumberbatch born?"*, *year* has a noun and ATTR labels for POS and type tags, respectively. We use the ATTR tag for labeling such nouns in natural language queries.
- **ATTRREF**: Like the TABLEREF tag, the ATTRREF tag is used to tag the verbs in the natural language query that can be mapped to the attributes in the SQL query. For instance, in the query *"List all the Sci-Fi movies released in 2010?"*, *released* has VDB (i.e., verb) as a label for the POS tag, whereas it refers to *release_year* attribute in the *movie* table.
- **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 *House of Cards* 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., *House* keyword in the example query is part of a keyword that needs to be mapped as *value* to *tv_series.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 instance, there is no specific word that has COND as a schema label for the given query in Table 1, whereas for the query *"List all the Sci-Fi movies released in 2010?"*, *in* indicates an equality predicate, and hence COND as schema label. 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. Example words are shown in Table 1.

3.3.3 Schema tags

Schema tags of keywords represent the database mapping that the keyword is referring to, the name of a table, or the attribute. Tagging a keyword with a type tag is important yet incomplete. To find the exact mapping the keyword refers to, we define a second-level tagging where the output is the name of the tables or attributes. For each entity table (e.g., *movie* table in the shortest path component of Fig. 6) and for each non-PK or non-FK attribute (attributes which have semantics) we define a schema tag (e.g., *movie, people, movie.title*, etc., referring to Fig. 6). 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.

4 Explanations for keyword mapper

In this section, we provide the details about the techniques we employ to explain the decisions made by DBTagger, our keyword mapper in the pipeline. First, we give a short overview of the LIME [42] work and highlight its applicability and limitation in the context of the sequence tagging problem. Next, we explain how we tailor LIME to the sequence tagging problem (i.e., classification problem for each item in the sequence) to deploy in our pipeline.

4.1 LIME

LIME [42] is short for "Local Interpretable Model-Agnostic Explanations", where each part in the name exhibits a desirable property a black-box explanation model must have. "Local" implies that LIME is an outcome explanation model, explaining the decision made on a particular instance, which is in line with our goal. "Model-agnostic" refers that LIME works with any type of input data (e.g., image, text) or a black-box model (e.g, a linear classifier such as logistics regression or a neural network based model), which is one of the reasons why we use LIME in xDBTagger that includes a neural network-based keyword mapper that we want to explain to the user.

Interpretation and explanation are important terms often used interchangeably in the context of XAI; however, they have distinct meanings. The former is more involved in providing abstracts in a way humans can make sense of, whereas the latter revolves around highlighting important features that play a role in decision-making for a given instance [15]. Analogously, the explanations that are not interpretable are useless, which is addressed by LIME. LIME argues that interpretable data representations differ from actual feature representations by asserting that interpretable data representations, such as binary vectors stating the existence of a word, are easily understood by humans. In contrast, actual feature representations, such as word embedding vectors, are not that straightforward and comprehensible. This distinctness is crucial since explanations produced by LIME are based on interpretable data representations.

In particular, LIME provides the importance of the features for a given instance as explanations, and to make them interpretable it follows a binary approach that highlights how important certain parts of the input are when they are present or absent. For a given input, LIME perturbs the input by randomly removing parts of the input and tries to understand how the model behavior changes. For instance, LIME creates a series of artificial sentences for a particular text input as a sentence in which random tokens are removed. LIME then tries to assign an importance score to each token for the decision (e.g., a target label by a classifier) by weighing the changes in model behavior. If the score is positive, the token is helpful when deciding the outcome for a particular input, whereas it is disadvantageous to the outcome when the score is negative. The absolute value of the score implies the contribution the token makes to the outcome, either positively or negatively.

4.2 LIME wrapper

Due to its properties, LIME is applicable in classification problems where the input is a sentence, and it is important to explain the importance of each token in the sentence in deciding a particular class. Also, note that the architecture of DBTagger, our keyword mapper, also utilizes signals from neighboring tokens when deciding the type and schema classes of a particular token by using CRF (see Fig. 3) at the last layer. This property of DBTagger aligns perfectly with the applicability of LIME in a text classification problem, as explained above. However, vanilla LIME is not directly applicable where the model generates multiple outcomes for a given sentence. In other words, vanilla LIME produces

explanations for models that classify the whole text sequence into one class (e.g., sequence classification such as sentiment analysis), whereas in our case, there is a classification for every token in a sentence, referred to as sequence tagging problem in NLP. Hence, we make modifications and add a wrapper around LIME to output explanations for each token suitable for DBTagger.

In particular, the wrapper around LIME uses four groups of information; the NLQ (i.e., the text input as a list of tokens), DBTagger Model Black-box (i.e., the probabilities of target classes for each token), predicted type and schema classes of DBTagger Model, and output mask to perturb the sentence suitable for keyword mapping problem. The main purpose of this wrapper is to coordinate the communication between LIME and the output mask. With the help of the output mask, it becomes possible for LIME to produce an explanation for a specific token; however, we still need to select the token that will be explained. To achieve this, we only explain the tokens with a predicted tag other than "*O*" (i.e., words labeled as irrelevant, thus not worth explaining for) for time efficiency. For each token that needs to be explained LIME wrapper produces 10 perturbed sentences, which we feed into the DBTagger model. The probabilities of the schema label given each perturbed sentence is then used to calculate the impact of surrounding words for the token to be explained. Since the average length of NLQs in words is around 10, we did not use a bigger number for the parameter of perturbed sentences required by the LIME wrapper. Having more perturbed sentences is redundant, which would slow down the explanation step of the translation pipeline.

Once the token selection is made, the wrapper adjusts the output mask so that the model gives the output for the selected token. Therefore LIME can analyze the output and produce an explanation for that token. This process is repeated for every token that is selected for the explanation.

5 SQL translation algorithm

We use simple yet effective algorithms to construct the translated query given a set of type and schema tags output by the keyword mapper, DBTagger. In particular, for the translation algorithm, we have three channels of input (see Fig. 1); (i) type tags (i.e., lists of tags indicating whether each word in NLQ is a table, column or value) and (ii) schema tags (i.e., names of the schema elements found such as table or column names for each word having a valid type tag) output by DBTagger, and (iii) the input query.

Utilizing the above-mentioned inputs, the SQL translation algorithm has 4 main components, which are (i) *Schema Graph Extraction*, (ii) *Join-Path Inference*, (iii) *Where Clause Completion* and (iv) *Heuristics for Aggregate Queries*. Each component is explained in detail in the following subsections.

5.1 Schema graph extraction

As the first step of the translation algorithm, we construct a *schema graph* out of the database upon which the NLQ is issued. The schema graph is similar to Entity-Relationship (ER) Diagram, which is a heavily used tool to illustrate conceptual design of relational databases. In the schema graph, we have only two types of nodes; (i) table and (ii) attribute. Table nodes resemble Entity-Sets in ER, and attribute nodes in the schema graph are similar to attributes depicted as part of entity-sets. The main difference between the two is that there is no relationship-set in a schema graph; table nodes are connected directly through their shared attributes (i.e., attributes exhibiting referential integrity, for instance, `movie` node is connected to 5 other table nodes through `msid`) attribute, which is the primary key (PK) of the movie table.

Each connection between a pair of nodes in the schema graph is an undirected edge, connecting table and attribute nodes to represent their *has-A* relations. An edge in the schema graph can only be between two different types of nodes (i.e., between a table and an attribute). Furthermore, we assumed that all foreign keys (FK) have the same name as their PK, and in the graph, there is only one node representing them. This simplification does not create any problem for database schemas that do not have self-references or multiple FKs referencing the same PK. This assumption can be relaxed by adding edges between FKs and PKs if the given database schema contains tables with different name PK-FK pairs, multiple FKs referencing the same PK, or self-references. An example schema graph extracted from the type and schema tags output by DBTagger for the example NLQ given in Sect. 1 is depicted in Fig. 4.

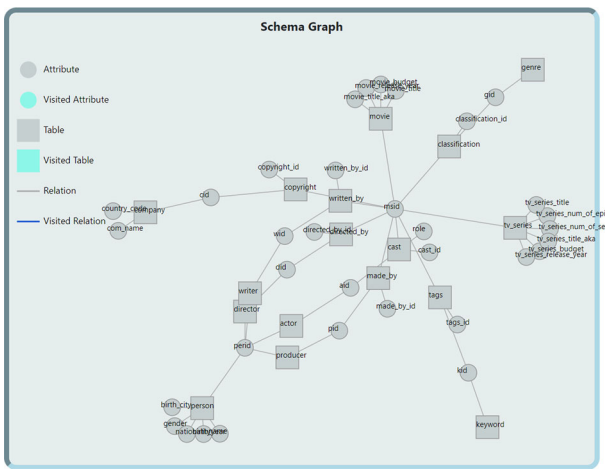


Fig. 4 An example schema graph consisting of tables, columns and their relations

5.2 Join-path inference

The main reason why we extract schema graphs is to produce correct join conditions required for SQL queries touching multiple tables. After constructing the schema graph, we find the shortest path between the table nodes to complete the necessary join conditions. Revisiting the example query given in the Introduction *"Who is the director of the series House of Cards produced by Netflix?"*, the keyword mapper detects three tables in the query which are director, tv_series, and company. As it can be seen in the corresponding schema graph depicted in Fig. 4, such tables are not connected to each other directly because they do not have direct relation through PK-FK attribute pairs. In order to translate the NLQ into SQL, we need to combine these tables, which is done by finding intermediate tables required to connect them together, such as copyright (e.g., linking tv_series and company) and directed_by (e.g., linking tv_series and director) to complete the join. Therefore, it becomes a problem of finding a possible path between the set of nodes in a fully connected graph and, while doing that, avoiding redundant paths for which shortest-path algorithms can be utilized. In our translation pipeline, we employed Dijkstra's algorithm to find the minimal path between table nodes detected in the NLQ. We used intermediate table nodes to construct the correct join conditions in the translated SQL.

The entire algorithm is given in Algorithm 1. Using the type and schema tags output by DBTagger, a set of tables T is created containing all the related tables for the given NLQ. T is composed of found tables (i.e., name of the tables for those words that have *TABLE* or *TABLEREF* as type tag), and tables of found attributes (i.e., name of the columns for those words that have *ATTR* or *ATTRREF* or *VALUE* as type tag). After that, T is given as an input to Algorithm 1 with the schema graph, and the shortest paths that contain and join the tables in T are found. An example join-path found for the given NLQ is depicted in Fig. 4. In the path, each consecutive nodes connected through an attribute node exhibits a join condition (e.g., *director* and *directed_by* tables require a join through attribute *did*).

```
ExtractJoinRelation (G, T)
    Input  : Database Graph G and list of tables T
    Output: Graph paths that contains SQL join information
    joinPath ← ∅;
    candidate ← null;
    foreach table t_i ∈ T do
        foreach table t_j ∈ T do
            if t_i ≠ t_j then
                paths ← findShortestPaths(G, t_i, t_j) foreach
                path p ∈ paths do
                    if T ⊆ p then
                        returnPaths.append(p);
                        return returnPaths;
                    else
                        missingTables ← T \ p;
                        if candidate == null then
                            candidate ← (p, missingTables);
                        else
                            if length(missingTables) <
                            length(candidate_1) then
                                candidate ←
                                (p, missingTables);
                    end
        end
    end
    returnPaths.append(candidate_0);
    foreach table t ∈ candidate_1 do
        paths ← findShortestPaths(G, candidate_00, t);
        foreach path ∈ paths do
            listReduced ← False;
            foreach table t_2 ∈ candidate_1 do
                if t_2 ≠ t ∧ t_2 ∈ path then
                    candidate_1.remove(t_2);
                    listReduced ← True;
            end
            if listReduced then
                returnPaths.append(path);
        end
    end
    return returnPaths;
```

Algorithm 1: Inferring Shortest Join-Path

5.3 Where clause completion

The next step is to construct WHERE conditions of SQL. The process starts with gathering the outputs of type and schema tags from DBTagger. A two-dimensional array M is created where $M_{i,1}$ contains the i^{th} query token, $M_{i,2}$ contains type tag of the i^{th} token, and $M_{i,3}$ contains the schema tag of the i^{th} token. Following that, a pre-processing step is applied on the array M to smooth out consecutive tokens that have the same mapping information by merging them together. In particular, the tokens that have *VALUE* as a type tag inhibit WHERE conditions to focus. For each list of consecutive tokens with *VALUE* mapping, a pair of the query token (e.g., House of Cards) and column name of the respective table (e.g., title column of the tv_series table) is created and added to where conditions of SQL. Algorithm 2 shows how WHERE conditions are extracted from the given list M.

5.4 Heuristics for aggregate queries

Finally, we used a simple and effective technique to detect potential aggregate operations for the constructed SQL. There are some specific keywords—such as total, many, count etc.—that imply certain aggregate operations. Using these keywords, we define keyword sets for each aggregate operation and perform a search in the NLQ for potential aggregate keywords. Algorithm 3 shows the outline of the performed search. After retrieving the mapping output from DBTagger, we select the words that have TABLE,

ExtractWhereConditions (M)
  Input : Two dimensional array M, containing the NLQ and keyword mapping information
  Output: SQL WHERE conditions $whereConditions$
  $whereConditions \leftarrow \emptyset$;
  $M \leftarrow mergeConsecutiveMappings(M)$; // Merges multi-word entities to a single mapping e.g: Brad Pitt
  foreach *token* $k_{1i}, k_{2i}, k_{3i} \in M_1, M_2, M_3$ **do**
    if k_{2i} *is VALUE* **then**
      $whereConditions.add(k_{1i}, k_{3i})$; // k_{1i} and k_{3i} contains the keyword and column information of that keyword respectively
  end
  return $whereConditions$;

Algorithm 2: Extraction of SQL WHERE Conditions

ExtractAggregateClause $(M, prevWindow)$
  Input : Two dimensional list M containing natural language query and keyword mapping information, $prevWindow$, the length of the sliding window, for finding aggregate keywords in NLQ
  Output: SQL AGGREGATE Clause
  $SUMKeywords \leftarrow getSUMKeywords()$;
  $COUNTKeywords \leftarrow getCOUNTKeywords()$;
  $AVGKeywords \leftarrow getAVGKeywords()$; **foreach** *token* $k_{1i}, k_{2i}, k_{3i} \in M_1, M_2, M_3$ **do**
    if $k_{2i} \in [TABLE, TABLEREF, ATTR, ATTRREF]$ **then**
      foreach *token* $k_{1j} \in [M_{1i-prevWindow}, ..., M_{1i}]$ **do**
        if $k_{1j} \in SUMKeywords$ **then**
          return $(SUM, k_{1i}, k_{2i}, k_{3i})$;
        if $k_{1j} \in COUNTKeywords$ **then**
          return $(COUNT, k_{1i}, k_{2i}, k_{3i})$;
        if $k_{1j} \in AVGKeywords$ **then**
          return $(AVG, k_{1i}, k_{2i}, k_{3i})$;
      end
  end
  return $None$

Algorithm 3: Extraction of SQL AGGREGATE Clause

TABLEREF, ATTR, or ATTRREF as the type tag as our candidates for aggregation. For each keyword set, we search the words that appear before our candidates. If we find a matching keyword, we return the candidate keyword, its mapping information, and the matching aggregate operation. If no matching is found, the algorithm returns $None$, implying that no aggregation should be applied. For searching the previous words of the token k_i, we define a window size $prevWindow$ and perform the keyword search for the words that are inside this window, namely $[k_{i-prevWindow}, ..., k_{i-2}, k_{i-1}]$.

Table 2 Statistics of the databases used

Properties (#)	Database imdb	mas	yelp
entity tables	6	7	2
relation tables	11	5	5
total tables	17	12	7
total attributes	55	28	38
nonPK-FK attributes	14	7	16
total tags	31	19	20
queries	131	599	128
tokens in queries	1250	4483	1234

6 Experimental setup

6.1 Datasets

In our experiments we used *yelp, imdb* [53], and *mas* [35] datasets which are heavily used in many NLIDB related works by the database community [2, 35, 43, 45, 53].

The statistics about each dataset for which annotation is done are shown in Table 2. In Table 2 (referring to Fig. 4), entity tables refer to main tables (e.g., Movie), relation tables refer to hub tables that store connections between entity tables (e.g., cast, written_by), nonPK-FK attributes refer to attributes in any table that is neither PK nor FK (e.g., gender in People table), and finally total tags refer to a 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 names of the nonPK-FK attributes in addition to COND and OTHER. In the last two rows of Table 2, we show the number of annotated NL questions, referred to as queries, and the number of total words inside these queries, referred to as tokens.

6.2 Quantitative evaluation of xDBTagger

6.2.1 Keyword mapping evaluation

In order to train DBTagger, the keyword mapper for the pipeline, we first split the datasets into train-validation sets with a $5-1$ 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 [59] and Nadam [14]. We start the training with Adadelta and continue 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 [23] 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 [34] (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 sixfold. All the results reported are the average test scores of sixfold. During inference, we discard POS and Type task results and use only Schema (final) tasks to measure scores.

We implemented three different unsupervised approaches utilized in the state-of-the-art NLIDB works for the keyword mapping task as baselines to compare with DBTagger. We implemented sql querying over database column approaches (regex and full-text search), which is preferred in NALIR [35]. We implemented a well-known tf-idf baseline for exact matching by constructing an inverted index over unique database values present, as in the work ATHENA [43]. We also implemented a semantic similarity matching approach in which pre-defined word embeddings are used. This approach is exercised by Sqlizer [53]. In addition to these conventional unsupervised solutions, we also implemented TaBERT [55], a pre-trained language model utilizing transformer architecture to compare with our proposed solution. For all the baselines, there is a component employing similarity matching which requires a manually crafted threshold, τ, to determine how much similarity is sufficient to map to a particular schema element, which makes it difficult to tune for different databases. For instance, when one chooses a lower similarity threshold, it becomes more likely to identify a true positive (i.e., higher recall) mappings; however, the solution becomes prone to generate false positives (i.e., lower precision) as a result for keywords that are not related to database elements such as stop words, sql specific words (i.e., return, find, minimum, etc.) For better comparison, we experimented with different thresholds for each baseline in each dataset and chose the one that resulted in the higher overall precision. 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.

- **tf-idf:** Similar to ATHENA [43], 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. 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 [35] uses WordNet, a lexical database in which synonyms are stored for relation matching. They calculate similarity for tokens present in the NLQ over WordNet, and determine a matching if the similarity is bigger than a manually defined threshold. For non-relation matching, for each token present in the NLQ, 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 word2vec embeddings. The matching with the highest similarity over a certain threshold is chosen.
- **TaBERT:** TaBert [55] is a transformer-based encoder which generates dynamic word representations (unlike word2vec) using database content. The approach also generates column encodings for a given table, which makes it an applicable keyword mapper for non-relation matching by performing cosine similarity over both encodings. For a particular token, matching with the maximum similarity over a certain threshold is chosen.

Effectiveness comparison

For a fair comparison, we do not apply any pre- or post-processing over the NL queries or use an 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. For TaBERT, we only report for non-relation matching, because the approach is not applicable to 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, the results of all approaches

Table 3 Accuracy scores of keyword mappers for relation and non-relation matching

Baseline	Database		
	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

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 the 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 the 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 the 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 for the non-relation matching task, which we believe is due to two reasons. Firstly, TaBERT has its own tokenizer, which relies on BERT base. The tokenizer tries to deal with out-of-vocabulary tokens by breaking the token into sub-words that have representations. This approach might be useful for a language model; however, it is problematic in the keyword mapping setup since the values present in the databases are domain-specific, which are likely to not occur in the general corpus data used to train such transformers. Also, databases such as imdb, have many entity-like values such as *Eternal Sunshine of the Spotless Mind* which is comprised of several words. Such keywords appearing in the natural language query are therefore divided by the tokenizer into pieces, which eventually leads to unrelated word representations and, thus, non-predictive similarity calculation. The other limitation of TaBERT is its requirement of using cosine similarity. Such an approach requires a manually defined threshold which is not easy to come up with. When a smaller similarity threshold is picked, chances of finding a true positive increases; however, the model becomes prone to generate false positives as well for keywords that are not related to database elements such as stop words and sql specific words (e.g., the, return, find, minimum).

We argue that unsupervised baselines may perform reasonably for relation-matching, whereas they fail to answer the challenges yielded by non-relation matching. This is due to the ambiguity present in the databases, such as having values that occur in multiple tables (e.g., "Matt Damon" may appear in both actor and director tables) and domain-specific values that are not covered in word embeddings (e.g., word2vec and TaBERT) trained on general corpus data.

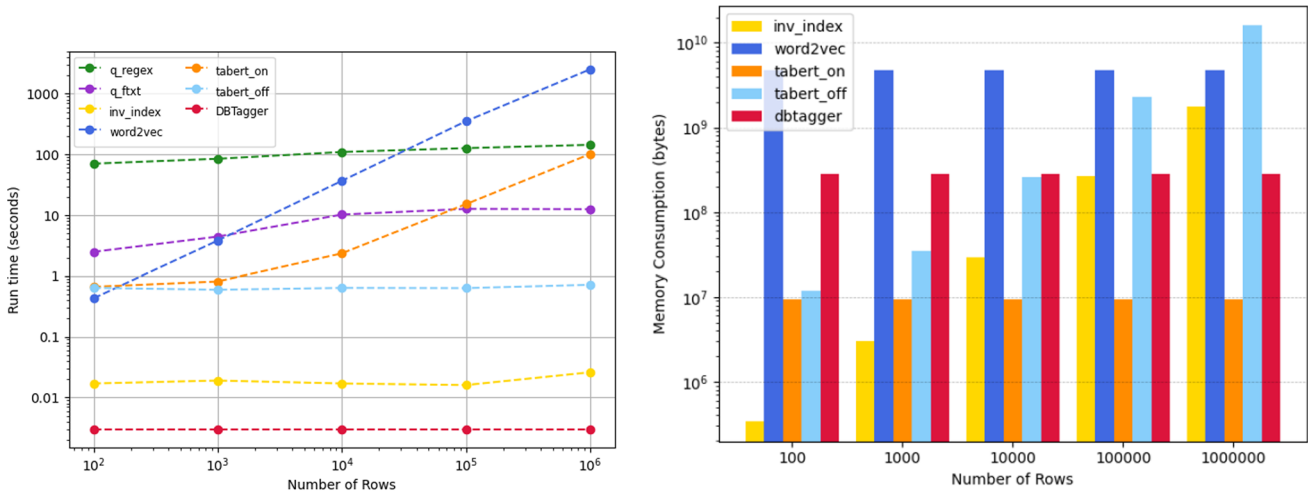
Importantly, the effectiveness of a keyword mapping approach is critical in any NLIDB solution trying to translate NLQ into SQL because the final accuracy is upper bound to the correct keyword mappings. Such mappings play an important role, particularly in where clauses, or from clauses that are required to perform query along with the intermediate tables required for joins, which are more than likely for enterprise-level databases.

Efficiency comparison

Efficiency is one of the most important properties of a good keyword mapper to have to be deployable in online interfaces. Therefore, we also evaluated the run-time performance and memory consumption of keyword mapping baselines mentioned in the previous section.

- **NALIR**: We analyze both querying over database column approaches used in NALIR [35], named as *q_regex* and *q_ftext*, which use *like* and *match against* operators respectively. NALIR [35] uses *q_regex* approach for tables having less than 2000 rows and *q_ftext* for tables having more rows.
- **tf-idf**: Similar to the indexing strategy exercised in ATHENA [43], we implemented an exact matching strategy, using an inverted index named as *inv_index*, beforehand to avoid querying over the database. The inverted index stores each unique value present in the database along with its frequency in each candidate collection (i.e., database columns).
- **word2vec**: Many works such as Sqlizer [53] make use of pre-trained word embeddings to find mappings, which requires keeping the model in the memory to perform the task using cosine similarity.
- **tabert_on**: TaBERT [55] requires database content (referred to as content snapshot in the paper) to generate encodings for both NL tokens and columns. We call this setup tabert online, where the model generates the content snapshot on the fly, hence online, 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 a different number of row values to capture the impact of the database size. Figure 5 presents run time and memory usage analysis of keyword mappers. DBTagger outputs 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 a nice balance between run-time and memory usage, but it is limited



(a) Run Time (b) Memory Usage

Fig. 5 Run Time and Memory Usage of state-of-the-art keyword mapping approaches

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, and its effectiveness as a candidate keyword mapper is not sufficient.

DBTagger is agnostic to database sizes in terms of efficiency since the memory required to perform the inference step is always the same, which only relies on the complexity of the deep learning model (i.e., the number of layers and parameters). The running time of an inference step of a query also relies on the length of the query in terms of words; however, the difference will be negligible since the actual cost is in 0.01 seconds. The other approach agnostic to database size in terms of memory consumption is tabert_on; however, as it can be seen from Fig. 5a, it is not practical to be deployed in real-world applications since it is not scalable to larger databases. For mappings, Tabert first has to generate content encodings (i.e., semantic representations of columns), processing rows of the tables entirely, which is a time-consuming process. If one wants to generate encodings offline, as in the case for tabert_off approach, the process becomes faster; however, this time, the memory space required to perform mappings gets bigger depending on the size of the tables. Note that Tabert generates encodings for each table in the database, indicating that similarity calculation will be performed not once but as many times as the number of tables. Another major bottleneck of these similarity-based approaches is the need for a sliding window technique (i.e., n-gram) to perform mapping, especially for multi-word entities (e.g., movie titles). N-grams of words of length between 3 and 6 are utilized, which consequently increases the number of times similarity calculation is performed, which automatically increases time complexity. Although query over database approaches (e.g., q_regex and g_ftext) preferred in NALIR do not occupy memory space, they fail to perform schema mapping in a reasonable time, especially when the number of rows in the tables gets bigger than 1000. That is why NALIR utilizes only g_ftext for tables with more than 2000 rows, which is still slow in a typical online setup.

6.2.2 Query translation results

We compared xDBTaggeer with two different types of baselines; first we compared our solution against its true competitors, pipeline-based approaches [2, 35] in binary accuracy metric. Next, we compared xDBTagger against one of the recent deep learning based solutions, Lgesql [5] in partial exact match accuracy to evaluate how effective our solution is while requiring less resource.

The numbers of the queries for the three datasets we used in our experiments are provided in Table 2. Also, recall that, we applied sixfold cross-validation (i.e., leaving 1 fold out for test and using the other fivefold for training the model) to train our keyword mapper. In order to evaluate query translation results, we performed the translation pipeline for each test fold left out from the training model for *yelp* and *imdb*. We used only 1 test fold for *mas* dataset to make the final number of test queries to be similar to each other.

Comparison with pipeline-based approaches

In this setup we manually evaluated the translated SQL queries, counting as correct if and only if the translated query is the same as the ground truth in terms of SQL semantics and correct in SQL syntax; and incorrect otherwise. Hence, we report binary accuracy results of xDBTagger.

In order to evaluate the comparative performance of xDBTagger, we used two different pipeline-based solutions; namely NALIR [35] and TEMPLAR [2] (an enhanced ver-

Table 4 Overall SQL query translation results against pipeline-based studies

Accuracy (%)	xDBTagger	NALIR+	NALIR
imdb	61.83	50.00	38.30
scholar	58.96	40.20	33.00
yelp	69.53	52.80	47.20

sion of the NALIR, referred as NALIR+, utilizing query logs to detect keyword mappings of the tokens in NLQ). The reason why we choose these two baselines is threefold. Firstly, both studies reported accuracy results of their translation pipeline for the same set of three datasets we used in our work. Secondly, both are pipeline-based solutions; that is, they are comprised of sub-solutions for each step in the translation pipeline similar to xDBTagger. Lastly, TEMPLAR [2] tries to enhance the translation pipeline of an existing NLIDB solution (e.g., NALIR) by solely focusing on keyword mappings. Similarly, xDBTagger utilizes the keyword mappings output by DBTagger [47] in the translation pipeline.

The overall translation accuracy results for xDBTagger, along with the two baselines explained above, are provided in Table 4. Accuracy results of the baselines are taken from the TEMPLAR study [2]. xDBTagger outperforms both baselines in all three datasets, up to 78% and 46% compared to NALIR and NALIR+, respectively. Considering efficiency (see Fig. 5 for reference) of the keyword mapper utilized in xDBTagger, simplicity of the translation algorithm explained in Sect. 5 and having fully explainable end-to-end translation pipeline, the accuracy of xDBTagger stands out even more compared to their counterparts.

In order to further show the efficacy of xDBTagger for other types of queries, we categorized the queries reflecting their difficulty in terms of translation and report accuracy for each category. The results are presented in Table 5. The numbers in parenthesis represent the number of queries falling under that particular category. The category *Nested* represents the number of queries xDBTagger could not translate due to the translation requiring a nested SQL query. Most of the queries fall under the category *Select-Join with Multiple Tables*, where xDBTagger performs most competitively across all categories. Accuracy results in Table 5 also indicate that heuristics for aggregation queries are effective at translating more than half of the queries under that category on the average.

We further manually evaluated $WHERE$ conditions of the queries having aggregate operations and reported accuracies in Table 6. As it can be seen, xDBTagger is able to extract $WHERE$ conditions fairly well with 90% average accuracy for imdb and scholar, and 76% accuracy for yelp. Although xDBTagger performs the worst for queries having aggregate operations in terms of full translations (Table 5), it still extracts correct $WHERE$ conditions for both utterances found in the NLQ and the join-path, which is reported to be the most challenging part of the translation in [52]. Results show that extracting correct $WHERE$ conditions for the translation is one of the main strengths of xDBTagger.

One of the limitations of the translation pipeline of xDBTagger is that it fails to translate certain types of queries correctly. xDBTagger is not able to translate the NLQs requiring nested SQL and group by queries. The biggest challenge for nested SQL queries is first to identify how many subqueries are required and then combine them together to produce both semantically and syntactically correct SQL queries. We argue that a rule-based approach similar to ours is insufficient to generalize to unseen NLQs requiring nested SQL. Therefore, we opted not to address those types of queries. However, note that such queries correspond to NLQs often exhibiting complex information needs, which constitutes a relatively minor percentage of queries issued by non-technical users who are the primary audience targeted by NLIDB. The results provided in Table 4 include those queries as well; hence less accuracy is observed overall. Another limitation is that queries requiring aggregate operations on top of grouping are difficult to translate (i.e., prone to a mistranslation) for xDBTagger. The strength of our translation pipeline comes from its keyword mapper, which is also evident by the results shown in Table 6, yet most of the queries with aggregation need not only mapping correct schema elements but also identifying possible grouping and/or the correct aggregation operation, which is rather difficult to generalize with a rule-based translation pipeline. Although we implemented heuristics (see Sect. 5.4) to address queries involving aggregates without group by and consequently translate more than half of them correctly (see Table 5), most of the incorrectly translated queries fall under this category.

Comparison with end-to-end neural network based approach

In this experimental setup, we compared xDBTagger against one of the state-of-the-art, end-to-end neural network-based approach, which is lgesql [5]. In order to fairly compare against them, we used partial exact match accuracy metric, since in lgesql database content is not utilized to predict database values (e.g., a specific movie title such as "House of Cards"). Therefore, final translated SQL queries have *value* as a placeholder. However, correctly predicting schema mappings for database values is regarded as the most challenging step (i.e., bottleneck) of the translation pipeline [54], which is also the strength of our solution. Therefore, we performed the comparison in exact match accuracy on 2 particular parts where each solution can be evaluated fairly. These are *where* and *sql keywords*. Note that as we explained above, our trans-

Table 5 Translation accuracy results of xDBTagger according to categorization of the queries

	Non-nested					Nested
	Select-Join (No-Aggregation)			Having aggregation	Overall	
	Single Table	Multiple Table	Overall			
imdb	88.89 (9)	68.60 (86)	70.53 (95)	63.64 (22)	69.23 (117)	14
scholar	100.00 (2)	86.96 (69)	87.32 (71)	43.59 (39)	71.81 (110)	24
yelp	60.00 (5)	83.61 (61)	81.81 (66)	63.64 (55)	73.55 (121)	7

Table 6 Accuracy of WHERE conditions of the queries with aggregate operations

	Aggregate queries with group by	Aggregate queries without group by	Overall
imdb	100.00 (6)	87.50 (16)	91.27 (22)
scholar	88.89 (18)	90.48 (21)	89.53 (39)
yelp	63.64 (11)	79.55 (44)	76.80 (55)

Table 7 Overall SQL query translation results against Lgesql in partial exact match accuracy

Partial exact match (%)	imdb		scholar		yelp	
	Where	Keywords	Where	Keywords	Where	Keywords
$lgesql_{glove}$	45.4	90.0	90.4	87.4	36.4	91.9
$lgesql_{bert}$	61.5	95.4	90.5	91.6	37.3	91.9
$lgesql_{electra}$	78.5	94.5	88.4	87.4	56.5	91.9
$xDBTagger$	69.7	92.4	84.5	72.6	53.8	88.9

lation pipeline does not handle group by queries, therefore we did not include it in the evaluations.

The comparison in partial exact match results is provided in Table 7. Note that *where* represents correct predicates in where clause by only considering left-hand side and the condition operation ignoring the database value. For instance, given the Example query *"Who is the director of the series House of Cards produced by Netflix?"*, the predicate of $company.name = "value"$ in the translated SQL out of lgesql is considered correct. However, our solution predicts the database value as well.

It can be seen from Table 7 that lgesql performs better compared to xDBTagger in both metrics, especially when augmented with Electra [8], a more complex pre-trained language model. However, XDBTagger performs competitively across all datasets, even surpassing the performance of lgesql when it is augmented with less complex contextual embeddings such as glove and bert for imdb and yelp datasets.

We also provide efficiency comparison of xDBTagger and lgesql with regards to different metrics in Table 8. Run times include pre-processing steps required to perform the translation. Not suprisingly, lgesql is 700 times more complex in terms of number of parameters in the model, requiring much more space to be stored (approximately 2000 times) and outputs the translation results slower than xDBTagger. On top of that, lgesql still requires a post-processing step, which we did not include in this experiment, to fill-in the *value* placeholders so that it can produce correct translations, which is another overhead. Typical post-processing step is based on an exact matching scheme [54] of query n-grams over database values, which we used as a baseline for keyword mapping under the name *inv_index* in Sect. 6.2.1 (see Table 3) and we reported that it is not effective for database value matching in imdb and scholar datasets. These results in Tables 7 and 8 indicate that xDBTagger is much more efficient compared to lgesql while not compromising effectiveness.

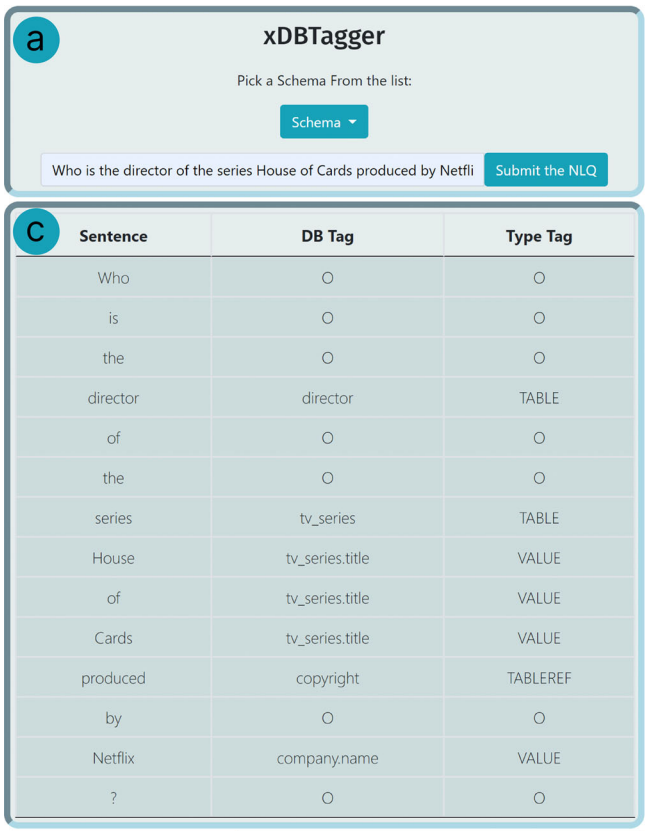
Table 8 Efficiency comparison of xDBTagger with Lgeseql

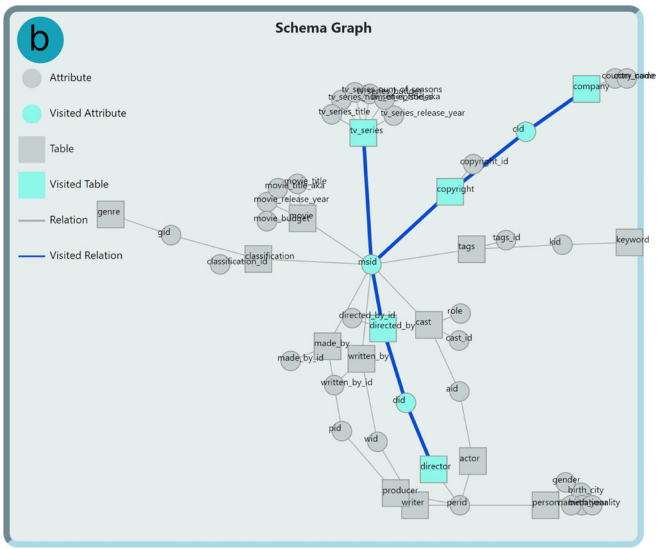
	$xDBTagger$	$lgeseql_{electra}$
Total # parameters	500 K	350 M
Size of model file	2.13 mb	4.0 gb
Run time per query (seconds)	1.93	3.10

6.3 Explainable user interface of xDBTagger

We constructed a simple, single-page web application where users can input a natural language query into our NLIDB pipeline and retrieve the translated SQL. This web application is developed using flask micro-framework and javascript. Figure 6 depicts interface components through the different stages of the translation pipeline. The pipeline has the following three main steps that we explain to the user separately;

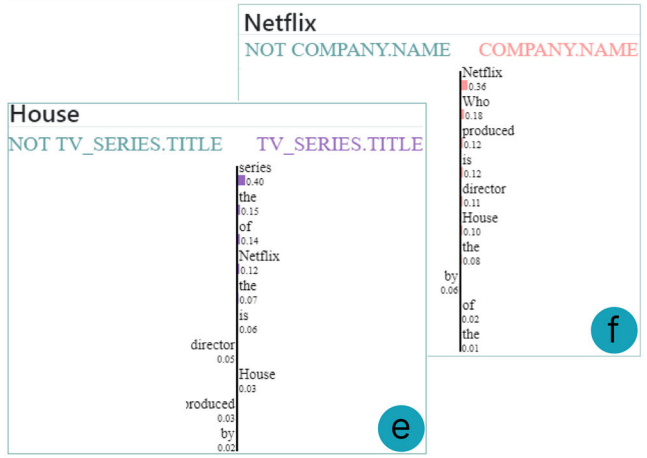
1. Finding type and schema mappings of the tokens in the NLQ using DBTagger. As discussed in Sect. 3, DBTagger is a sequence-tagging deep learning model, which





c

Sentence	DB Tag	Type Tag
Who	O	O
is	O	O
the	O	O
director	director	TABLE
of	O	O
the	O	O
series	tv_series	TABLE
House	tv_series.title	VALUE
of	tv_series.title	VALUE
Cards	tv_series.title	VALUE
produced	copyright	TABLEREF
by	O	O
Netflix	company.name	VALUE
?	O	O



d

SQL	Explanation
SELECT * FROM tv_series,	required in WHERE clause
company,	required in WHERE clause
director,	detected TABLE mapping by DBTagger
copyright,	detected TABLE mapping by DBTagger
directed_by	required for JOIN relation
WHERE(tv_series.msid = copyright.msid	join condition
AND copyright.cid = company.cid	join condition
AND tv_series.msid = directed_by.msid	join condition
AND directed_by.did = director.did)	join condition
AND (tv_series.title = "House of Cards"	detected VALUE mapping by DBTagger
AND company.name = "Netflix")	detected VALUE mapping by DBTagger

Fig. 6 Interface components of xDBTagger; **a** NLQ input panel, **b** schema graph panel on which required join path is drawn, **c** predictions panel for keyword mapper, **d** result panel containing the translated SQL query along with its explanations, **e**, **f** explanation pop-ups for each token in NLQ. Components **b**, **c**, **d**, **e** and f contain visual and/or textual explanations for the algorithms utilized through the translation pipeline

behaves as a black-box mechanism, i.e., it only classifies the tokens in the NLQ to the most probable class among the possible candidates based on probabilities it derives, yet it does not provide why it makes a particular decision. Therefore, we need to explain how DBTagger maps tokens in the NLQ to the schema elements in the database. To this end, we provide two different interface components. First, we list all the type and schema mappings output by DBTagger for each token in a table (6c) to make the decisions made by DBTagger transparent. Second, we visually explain why DBTagger came up with a particular pair of type and schema mapping for a token in a pop-up (6e,f) using LIME wrapper (explained in Sect. 4) to make it easier for the user to comprehend the decisions made by DBTagger.

2. Extracting the join path necessary to access tables that include utterances found in the previous step. We first visually draw the schema graph on the panel (depicted in Fig. 6b) so that the user can better understand the schema underlying the database and the relational connections it inherits. Moreover, we also highlight the nodes and the edges visited along the path on the graph to construct the JOIN clause to explain how certain tables that are not present in predictions panel (6c) (i.e., intermediate tables completing the join) appear in the final SQL translation.
3. Constructing the SQL by forming the WHERE clauses and applying post-processing heuristics to handle certain group of queries. After finalizing the SQL translation, we part-by-part explain how the translated SQL is composed in the result panel, shown in 6d. In particular, we explain why we include each table in the FROM clause and each logical expression in the WHERE clause.

NLQ input panel (Fig. 6a) allows the user to select a database schema from a dropdown list and input a query to the NLIDB pipeline for the selected schema. After selecting the schema over which the query is to be issued, the schema graph panel displays the extracted graph (i.e., the graph without the highlighted nodes and edges, depicted in Fig. 4), as explained in Sect. 5.1. When a query is processed in the pipeline and a SQL is generated for the translation, we highlight the nodes and edges of the graph in blue color, as shown in Fig. 6b, that are used in any part of the generated SQL; (i) tables and their attributes required for the WHERE clauses and (ii) tables required for the correct join operation along with their attributes making the PK-FK connections.

In the predictions panel shown in Fig. 6c, we present the type and schema mapping outputs of our keyword mapper inside a table so that the user can visually see how his/her query is predicted by the keyword mapper. Furthermore, we used pop-ups to display LIME explanations of each word of the query that is not tagged as 'O' to make it easier for the user to comprehend the decision-making behind the keyword mapper model. When the user clicks on a particular row of the table in Fig. 6c, a pop-up (e.g., Fig. 6e) is displayed containing the explanation for the word visually by highlighting the neighboring words that contribute the most, either positively or negatively, when predicting the output.

For example, for the query shown in the input panel Fig. 6a, we provided the explanation pop-ups for tokens "House" and "Netflix" in Fig. 6e and f, respectively. In Fig. 6e, there are two labels named 'NOT TV_SERIES.TITLE' and 'TV_SERIES.TITLE', which correspond to categories of tokens contributing to the label of 'tv_series.title' negatively and positively, respectively. Below both labels, on each side, there are tokens from the NLQ with a contribution score associated with them. A positive contribution means that the token increases the prediction probability of the explained token for the given class (i.e., TV_SERIES.TITLE), and a negative contribution decreases that probability.

For the token in Fig. 6e, we can see that the word 'series' in the NLQ has the highest positive contribution marginally compared to other tokens. This means that the word 'series' is the most influential neighboring word when determining the mapping classification of the token 'House'. Similarly, Fig. 6f gives the explanation for the word 'Netflix' in the given NLQ. The explanation shows that the word itself has the highest positive contribution, which is expected since the entity is self-expressive and should infer the name attribute of a particular entry in the company table.

The result panel, shown in Fig. 6d, presents the generated SQL query and the explanation of how it is composed for the given NLQ. We divide the generated SQL statement into parts and explain why we include each part in the final statement so that users with a less technical background in SQL can better comprehend how the final SQL is composed. In particular, we explain why we include tables and logical expressions in the FROM and WHERE clauses, respectively. For instance, for the example NLQ given in Fig. 6a, there are three different explanations for why a certain table is included in the FROM clause. 'tv_series' table is included because there are tokens whose schema mappings are title attribute of the 'tv_series' table (8-10th rows in the prediction table shown in Fig. 6c). Director and copyright tables appear thanks to the predictions of DBTagger, whereas 'directed_by' is present because it is a required table to connect 'tv_series' and 'director' through join.

For each logical expression we put in the finalized SQL statement, we provide an explanation as well. Broadly, we divide the explanations into two categories. If a logical expression is to provide a connection between tables in the schema graph, we say it is required for the join condition. If we detect a type mapping of *VALUE* (e.g., 8–10th and 13rd rows in the prediction table shown in Fig. 6c), we state that a value is detected by DBTagger as shown in the last two rows in the explanation table in Fig. 6d.

7 Related work

Although the very first effort [22] 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 [3], NALIR [35], ATHENA [43] and SQLizer [53].

However, with the recent advancements in deep neural networks, the problem of NLIDB has also attracted researchers from the NLP community. [60] provided a dataset called *WikiSql* to the research community working on NLIDB problem for evaluation. WikiSql is comprised of 26, 531 tables and

80, 654 pairs which can be used for input for the translation problem. Consequently, many works [26, 46, 52, 54, 56, 60] utilizing encoder-decoder abstraction have been proposed to evaluate their translation solutions on the WikiSql dataset. However, the dataset only includes schemas with a single table, limiting detailed evaluation of the solutions due to simplicity.

To remediate this limitation, *Spider* dataset is provided in the work [58] to the community. Many studies utilizing the pioneer work BERT [13], a pre-trained language model based on the transformer [48] architecture, have evaluated their solutions on Spider [58] dataset. Some works [19, 36] focus on the schema linking process to enrich the input NLQ for better leveraging the schema information. In IRNet [19], the authors first query n-grams of the NLQ over the database elements to find candidates and then feed these found candidates to the additional schema encoder, whereas Lin et al. [36] integrate these found candidates into the input as a serialization technique before encoding. Rat-SQL [49] proposes a modified transformer layer to leverage schema information better by introducing bias towards the schema for the attention mechanism. In addition to these studies, language representation techniques utilizing BERT such as TaBERT [55] and Grappa [11] have been introduced to leverage tabular data specific representations in related downstream tasks such as NLIDB problem. For a comprehensive survey covering existing solutions in NLIDB, the reader can refer to [1, 31].

To the best of our knowledge, there is no hybrid solution utilizing both neural network- and rule-based techniques, proposed similar to xDBTagger. Nonetheless, some of the earlier works [19, 36, 54, 56] embracing end-to-end neural network approaches focused more on enriching the input by trying to map tokens in the NLQ to database values similar to our keyword mapper, DBTagger. However, as shown in the work [47], such solutions proposing ad-hoc querying over database tables to find candidate mappings are not efficient and not scalable to bigger databases unlike xDBTagger.

There have been previous studies [12, 32, 39, 51] proposed in line with interpretable interfaces to databases. However, such solutions rather focus on providing additional information for the SQL query results in the form of summarized texts or snippets exploiting signals of the tuples returned by the result SQL. In this work, our goal is not to explain query results but to explain the decisions that lead to the result SQL for each step in the translation pipeline. To our knowledge, xDBTagger is the first NLIDB system exercising XAI principles to explain how the translation is performed.

8 Conclusion

In this work, we presented xDBTagger, the first end-to-end explainable NLIDB solution to translate NLQs into their counterpart SQLs. xDBTagger is a hybrid solution taking advantage of both deep learning- and rule-based approaches. First, we detect keyword mappings of the tokens in the input NLQ using a novel deep learning model trained in a multi-task learning setup. Next, we explain the decisions for the keyword mappings using a modified version of a state-of-the-art XAI solution LIME [42]. We visually illustrate the importance of each surrounding word for each mapping by highlighting their contributions which can be either positive or negative. In addition, we draw the schema graph to visualize better the database schema over which the query is issued. We also color the nodes representing tables and attributes in the graph to explain how the required join conditions in the result SQL are extracted. Finally, we explain each part of the result SQL to the user by providing the reason why we need that particular part given the input NLQ. Our quantitative experimental results indicate that in addition to being fully explainable, xDBTagger is effective in terms of translation accuracy and more preferable compared to other pipeline-based solutions in terms of efficiency.

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

References

1. Affolter, K., Stockinger, K., Bernstein, A.: A comparative survey of recent natural language interfaces for databases. VLDB J. **28**(5), 793–819 (2019)
2. Baik, C., Jagadish, H.V., Li, Y.: Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 374–385 (2019)
3. Blunschi, L., Jossen, C., Kossmann, D., Mori, M., Stockinger, K.: Soda: generating SQL for business users. Proc. VLDB Endow. **5**(10), 932–943 (2012)
4. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching word vectors with subword information. Trans. Assoc. Comput. Linguist. **5**, 135–146 (2017)
5. Cao, R., Chen, L., Chen, Z., Zhao, Y., Zhu, S., Yu, K.: LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 2541–2555. Association for Computational Linguistics, Online. https://doi.org/10.18653/v1/2021.acl-long.198, https://aclanthology.org/2021.acl-long.198 (2021)
6. Caruana, R.: Multitask learning. Mach. Learn. **28**(1), 41–75 (1997)
7. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Gated feedback recurrent neural networks. In: Proceedings of the 32nd International Conference on International Conference on Machine Learning, JMLR.org, ICML'15, vol. 37, pp. 2067–2075 (2015)
8. Clark, K., Luong, M., Le, Q.V., Manning, C.D.: ELECTRA: pre-training text encoders as discriminators rather than generators. arXiv:2003.10555 (2020)
9. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. J. Mach. Learn. Res. **12**(null), 2493–2537 (2011)

10. Crawshaw, M.: Multi-task learning with deep neural networks: a survey. CoRR abs arXiv:2009.09796 (2020)
11. Deng, X., Awadallah, A.H., Meek, C., Polozov, O., Sun, H., Richardson, M.: Structure-grounded pretraining for text-to-SQL. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1337–1350. Association for Computational Linguistics, Online (2021)
12. Deutch, D., Frost, N., Gilad, A.: Explaining natural language query results. VLDB J. **29**(1), 485–508 (2020)
13. Devlin, J., Chang, M. W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics, Minneapolis (2019)
14. Dozat, T.: Incorporating nesterov momentum into adam. In: ICLR Workshop, JMLR.org (2016)
15. Došilović, F.K., Brcic, M., Hlupic, N.: Explainable artificial intelligence: a survey. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 0210–0215 (2018)
16. Graves, A., Mohamed, A., Hinton, G.: Speech recognition with deep recurrent neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6645–6649 (2013)
17. Gregor, S., Benbasat, I.: Explanations from intelligent systems: theoretical foundations and implications for practice. MIS Q. **23**, 497–530 (1999)
18. Gunning, D., Aha, D.: DARPA's explainable artificial intelligence (XAI) program. AI Mag. **40**(2), 44–58 (2019)
19. Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.G., Liu, T., Zhang, D.: Towards Complex Text-to-SQL in Cross-domain Database with Intermediate Representation, pp. 4524–4535. Association for Computational Linguistics, Florence, Italy (2019)
20. Hayes-Roth, F., Jacobstein, N.: The state of knowledge-based systems. Commun. ACM **37**(3), 26–39 (1994)
21. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
22. Hendrix, G.G., Sacerdoti, E.D., Sagalowicz, D., Slocum, J.: Developing a natural language interface to complex data. ACM Trans. Database Syst. **3**(2), 105–147 (1978)
23. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Improving neural networks by preventing co-adaptation of feature detectors. ArXiv abs arXiv:1207.0580 (2012)
24. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**, 1735–80 (1997)
25. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K. Q.: Densely connected convolutional networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2261–2269 (2017)
26. Huang, P.S., Wang, C., Singh, R., Yih, W., He, X.: Natural Language to Structured Query Generation via Meta-learning, pp. 732–738. Association for Computational Linguistics, New Orleans (2018)
27. Huang, Z., Xu, W., Yu, K.: Bidirectional LSTM-CRF models for sequence tagging. arXiv:1508.01991 (2015)
28. Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., Zettlemoyer, L.: Learning a neural semantic parser from user feedback. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 963–973 . Association for Computational Linguistics, Vancouver (2017)
29. Jou, B., Chang, S.F.: Deep cross residual learning for multitask visual recognition. In: Proceedings of the 24th ACM International Conference on Multimedia, Association for Computing Machinery, New York, MM '16, pp. 998–1007. https://doi.org/10.1145/2964284.2964309 (2016)
30. Katsogiannis-Meimarakis, G., Koutrika, G.: A survey on deep learning approaches for text-to-SQL. VLDB J. **32**(4), 905–936 (2023). https://doi.org/10.1007/s00778-022-00776-8
31. Kim, H., So, B.H., Han, W.S., Lee, H.: Natural language to SQL: Where are we today? Proc. VLDB Endow. **13**(10), 1737–1750 (2020)
32. Koutrika, G., Simitsis, A., Ioannidis, Y.E.: Explaining structured queries in natural language. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), pp. 333–344. https://doi.org/10.1109/ICDE.2010.5447824 (2010)
33. Lafferty, J.D., McCallum, A., Pereira, F.C.N.: Conditional random fields: probabilistic models for segmenting and labeling sequence data. In: Proceedings of the Eighteenth International Conference on Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, ICML '01, pp. 282–289 (2001)
34. Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., Dyer, C.: 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, pp. 260–270. Association for Computational Linguistics, San Diego (2016)
35. Li, F., Jagadish, H.V.: Constructing an interactive natural language interface for relational databases. Proc. VLDB Endow. **8**(1), 73–84 (2014)
36. Lin, X. V., Socher, R., Xiong, C.: Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In: Findings of the Association for Computational Linguistics: EMNLP 2020, pp. 4870–4888. Association for Computational Linguistics, Online (2020)
37. Ma, X., Hovy, E.: End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 1064–1074. Association for Computational Linguistics, Berlin (2016)
38. Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., McClosky, D.: The stanford CoreNLP natural language processing toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations, pp. 55–60 (2014)
39. Müller, T., Grust, T.: Provenance for SQL through abstract interpretation: value-less, but worthwhile. Proc. VLDB Endow. **8**(12), 1872–1875 (2015)
40. Özcan, F., Quamar, A., Sen, J., Lei, C., Efthymiou, V.: 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, Association for Computing Machinery, New York, NY, USA, SIGMOD '20, pp. 2629–2636 (2020)
41. Poulin, B., Eisner, R., Szafron, D., Lu, P., Greiner, R., Wishart, D.S., Fyshe, A., Pearcy, B., MacDonell, C., Anvik, J.: Visual explanation of evidence with additive classifiers. In: Proceedings of the National Conference on Artificial Intelligence, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, vol. 21, p. 1822 (2006)
42. Ribeiro, M. T., Singh, S., Guestrin, C.: "why should I trust you?": explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13–17, 2016, pp. 1135–1144 (2016)
43. Saha, D., Floratou, A., Sankaranarayanan, K., Minhas, U.F., Mittal, A.R., Özcan, F.: ATHENA: an ontology-driven system for natural language querying over relational data stores. Proc. VLDB Endow. **9**(12), 1209–1220 (2016)
44. Scholak, T., Schucher, N., Bahdanau, D.: PICARD: parsing incrementally for constrained auto-regressive decoding from language

models. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pp. 9895–9901. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic (2021)
45. Sen, J., Lei, C., Quamar, A., Özcan, F., Efthymiou, V., Dalmia, A., Stager, G., Mittal, A., Saha, D., Sankaranarayanan, K.: ATHENA++: natural language querying for complex nested SQL queries. Proc. VLDB Endow. **13**(12), 2747–2759 (2020)
46. Sheinin, V., Khorashani, E., Yeo, H., Xu, K., Vo, N.P.A., Popescu, O.: Quest: a natural language interface to relational databases. In: Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018) (2018)
47. Usta, A., Karakayali, A., Ulusoy, O.: DBTagger: multi-task learning for keyword mapping in NLIDBs using bi-directional recurrent neural networks. Proc. VLDB Endow. **14**(5), 813–821 (2021)
48. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 30. Curran Associates Inc, New York (2017)
49. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: RAT-SQL: relation-aware schema encoding and linking for text-to-SQL parsers. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 7567–7578. Association for Computational Linguistics, Online (2020)
50. Weir, N., Utama, P., Galakatos, A., Crotty, A., Ilkhechi, A., Ramaswamy, S., Bhushan, R., Geisler, N., Hättasch, B., Eger, S., Cetintemel, U., Binnig, C.: DBPal: a Fully Pluggable NL2SQL training pipeline. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '20, pp. 2347–2361 (2020)
51. Wen, Y., Zhu, X., Roy, S., Yang, J.: Interactive summarization and exploration of top aggregate query answers. Proc. VLDB Endow. **11**(13), 2196–2208 (2018)
52. Xu, X., Liu, C., Song, D.: SQLNet: generating structured queries from natural language without reinforcement learning. arXiv preprint arXiv:1711.04436 (2017)
53. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: SQLizer: query synthesis from natural language. Proc. ACM Program. Lang. **1**(OOPSLA), 63:1-63:26 (2017)
54. Yavuz, S., Gur, I., Su, Y., Yan, X.: What it takes to achieve 100% condition accuracy on WikiSQL. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pp. 1702–1711. Association for Computational Linguistics, Brussels (2018)
55. Yin, P., Neubig, G., Yih, Wt., Riedel, S.: TaBERT: pretraining for joint understanding of textual and tabular data. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 8413–8426. Association for Computational Linguistics, Online (2020)
56. Yu, T., Li, Z., Zhang, Z., Zhang, R., Radev, D.: TypeSQL: Knowledge-based Type-aware Neural Text-to-SQL Generation, pp. 588–594. Association for Computational Linguistics, New Orleans (2018)
57. Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., Radev, D.: 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, pp. 1653–1663. Association for Computational Linguistics, Brussels (2018)
58. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.: Spider: A Large-scale Human-Labeled Dataset for Complex and Cross-domain Semantic Parsing and Text-to-SQL Task, pp. 3911–3921. Association for Computational Linguistics, Brussels (2018)
59. Zeiler, M. D.: ADADELTA: an adaptive learning rate method. arXiv:1212.5701 (2012)
60. Zhong, V., Xiong, C., Socher, R.: Seq2SQL: generating structured queries from natural language using reinforcement learning. arXiv preprint arXiv:1709.00103 (2017)