ValueNet: A Natural Language-to-SQL System that Learns from Database Information

Ursin Brunner
ZHAW Zurich University of Applied Sciences
Winterthur, Switzerland
Ursin.Brunner@zhaw.ch

Kurt Stockinger
ZHAW Zurich University of Applied Sciences
Winterthur, Switzerland
Kurt.Stockinger@zhaw.ch

***Abstract*—Building natural language (NL) interfaces for databases has been a long-standing challenge for several decades. The major advantage of these so-called NL-to-SQL systems is that end-users can query complex databases without the need to know SQL or the underlying database schema. Due to significant advancements in machine learning, the recent focus of research has been on neural networks to tackle this challenge on complex datasets like Spider. Several recent NL-to-SQL systems achieve promising results on this dataset. However, none of the published systems, that provide either the source code or executable binaries, extract and incorporate values from the user questions for generating SQL statements. Thus, the practical use of these systems in a real-world scenario has not been sufficiently demonstrated yet.**

In this paper we propose *ValueNet light* and *ValueNet* – two end-to-end NL-to-SQL systems that incorporate values using the challenging Spider dataset. The main idea of our approach is to use not only metadata information from the underlying database but also *information on the base data* as input for our neural network architecture. In particular, we propose a novel architecture sketch to extract values from a user question and come up with possible value candidates which are not explicitly mentioned in the question. We then use a neural model based on an encoder-decoder architecture to synthesize the SQL query. Finally, we evaluate our model on the Spider challenge using the *Execution Accuracy* metric, a more difficult metric than used by most participants of the challenge. Our experimental evaluation demonstrates that *ValueNet light* and *ValueNet* reach state-of-the-art results of 67% and 62% accuracy, respectively, for translating from NL to SQL whilst incorporating values.

***Index Terms*—NL-to-SQL, natural language interface, neural networks, transformers**

I. Introduction

Building NL-to-SQL systems is a long-standing challenge in both the database and the natural language processing community. Being able to query databases and other structured data in natural language gives users without knowledge in a query language access to a large amount of information. Therefore, a natural language interface has often been regarded as the most powerful database interface [27].

At the same time proprietary systems such as Google's *Assistant* or Amazon's *Alexa* are improving the way users can access large knowledge bases with natural language. It is therefore an integral focus of any open data effort to offer users a similarly convenient interface to query data by natural language.

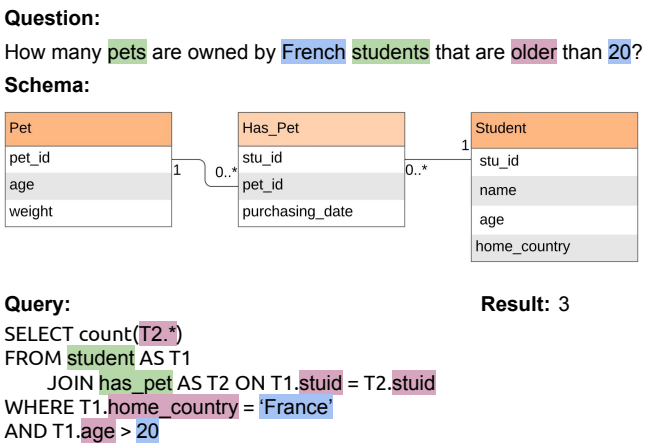

Fig. 1. A typical NL-to-SQL system synthesizes and executes a SQL query given a natural language question and a database schema. Besides synthesizing a valid SQL sketch (non-highlighted words), the system has to select the correct tables *(green)*, columns *(red)* and values *(blue)*

In Figure 1 we see the typical NL-to-SQL flow: Given a natural language question and a database schema, the system has to synthesize a valid SQL query. Once executed, this query should return the answer the user was looking for. Let us consider the query *How many pets are owned by French students that are older than 20?*. This example shows some of the challenges that such a system is confronted with. The fact that "older" is most probably referring to the column *age* (in the *Student* table rather than in the *Pet* table!) cannot be directly extracted from the question. The same principle applies to the fact that "French students" most probably refers to an entry of the column *home_country* containing the *value* "France". In other words, the token "French" of the natural language question does not directly map to a value in the *base data* of the underlying database. The latter challenge is the focus of our paper, i.e. how to build an end-to-end neural architecture that *takes values into account*.

The goal is to generate complex, nested SPJ[1] SQL statements including WHERE-clauses. The WHERE-clauses are constructed by predicting the *correct table, column and base*

[1] Select-Project-Join

data value (e.g. student.home_country = 'France') that correspond to the respective value in the natural language question (e.g. "French students"). We will see below that taking values into account has so far received little attention from the current state-of-the-art of neural network-based NL-to-SQL systems.

The release of *Spider* dataset motivated several research groups to provide contributions in 2019/2020 and has become the de-facto standard for evaluating NL-to-SQL systems. While classical neural architectures [13], [39] achieved rather low accuracy scores on the *Spider* leaderboard[2] (4.8%, 12.4%), more advanced approaches like *IRNet* [16] and *RAT-SQL* [36] score significantly higher with accuracies over 60%.

The *Spider* challenge was released with two evaluation metrics and leaderboards:

- **Exact Matching Accuracy**: The metric measures if a predicted query is equivalent to the gold query. Thanks to its smart *component matching*, this metric is tolerant towards ordering issues (e.g. `SELECT A, B` vs `SELECT B, A`). Unfortunately though the metric does not evaluate *values* such as "French" or "20" in Figure 1 – which is essential in real-world scenarios. In other words, this metric only considers the schema of the database and not the base data. The reason is that queries do not need to be executed against a database for this type of evaluation.
- **Execution Accuracy**: Measures if the results of both predicted and gold query are the same by *executing* them against a real database. To pass this evaluation, a NL-to-SQL system not only has to predict the correct SQL sketch and select the right tables/columns, but has also to identify the correct *values* and place them in the right order.

As of today, most *Spider* contributions focus on the *Exact Matching Accuracy* evaluation (see "Leaderboard - Exact Set Match without Values"). While a good score on this leaderboard is definitely an achievement (it requires a system to predict a complex SQL-sketch and select the right tables and columns), it is still a simplification of a fully fledged NL-to-SQL system. It does not address the challenging task of generating and selecting *values* and it still abstracts from several other difficulties addressed throughout this paper. Although there are a few contributions on the Spider leaderboard that evaluate their systems based on the more complex *Execution Accuracy* (see "Leaderboard - Execution with Values"[2]), there is currently *no published paper or published source code available*. Hence, there is currently no description and systematic evaluation of a neural network-based system for the Spider dataset that translates natural language to SQL and takes values into account.

In this paper we make the following **contributions**:

- To the best of our knowledge, we provide the first *detailed discussion as well as the source code of an NL-to-SQL system* to synthesize a full SQL query including values on the challenging Spider dataset using *Execution Accuracy*. Our approach is thus a step forward in building an end-to-end-system to generate complex, nested SPJ-queries that are typical of real-world scenarios. By providing the source code, our *approach is also reproducible* - which is often difficult to achieve when systems do neither provide the source code nor executable binaries. We achieve a state-of-the-art accuracy of up to 67% on the Spider dataset using the more challenging *Execution Accuracy* metric.
- We provide *two novel NL-to-SQL architectures*: (1) *ValueNet light* - A system which selects the correct values from a given list of possible ground truth values and then synthesizes a full query including the chosen values. (2) *ValueNet* - An end-to-end architecture which extracts and generates value candidates given only natural language questions and the database content. *ValueNet* then uses these value candidates to synthesize a full query including values equivalent to *ValueNet light*.
- We show that the difference in performance between *ValueNet* and *ValueNet light* is relatively small given a *strong generative approach for the candidate generation*. This indicates that if we come up with the right value candidates, the neural model is capable of selecting them correctly.

The paper is organized as follows. In Section 2 we define the problem of NL-to-SQL in more detail. In Section 3 we in describe the high-level architecture of our NL-to-SQL system. In Section 4 we discuss the detailed architecture with focus on ValueNet light and ValueNet. Our experimental results are given in Section 5. In Section 6 we review the related work. Finally, we draw conclusions in Section 7.

II. Problem Definition

A. Intermediate Representation via Abstract Syntax Tree

Before we introduce our end-to-end architecture for translating from natural language to SQL, we describe the concept of *Abstract Syntax Trees* (AST). The idea is to use ASTs as an intermediate representation in the overall translation process - rather than translating directly to SQL. The advantage of using an AST as an immediate representation is to overcome the so-called *mismatch* problem [16], where end-users rarely pose a question as detailed as necessary in order to directly synthesize a SQL query. Therefore, abstracting the details of SQL enables the system to understand the question/SQL query pairs more reliably as shown previously [7], [17]. Another advantage is that an intermediate representation enables the system to be independent of a specific query language.

For our approach we do not create a grammar for such a representation from scratch, but extend the context-free grammar *SemQL* of *IRnet* [16]. We call our extended version *SemQL 2.0*.

Figure 2 shows the complete grammar for SemQL 2.0 that can handle the major relational operators such as *select, project, join, union, intersect* as well as *aggregations and complex, nested queries*. We extend the grammar introduced in

[2]https://yale-lily.github.io/spider (last accessed on Oct 8, 2020)

IRnet [16] mainly by the value representation *V* - highlighted in the figure in green.

```
          Z ::= intersect R R | union R R |
                except R R | R
          R ::= Select | Select Filter |
                Select Order | Select Superlative |
                Select Order Filter |
                Select Superlative Filter
     Select ::= distinct N | N
          N ::= A | A A | A A A | A A A A | A A A A A
      Order ::= asc A | desc A
Superlative ::= most V A | least V A
     Filter ::= and Filter Filter | or Filter Filter |
                = A V | = A R | ≠ A V | ≠ A R |
                < A V | < A R | > A V | > A R |
                ≤ A V | ≤ A R | ≥ A V | ≥ A R |
                between A V V | between A R
                in A R | not in A R
          A ::= max C T | min C T | count C T | sum C T |
                avg C T | C T
          C ::= column
          T ::= table
          V ::= value
```

Fig. 2. SemQL 2.0 Grammar. Our contributions to the previously published SemQL 1.0 grammar are highlighted in green. The identifiers to the left of the *::=* represent all possible actions in the grammar. Italic tokens represent SQL operators or references to tables/columns/values. The pipe separates different implementations of an action.

B. Problem Statements: NL to AST to SQL

Given the grammar of SemQL 2.0, we will now describe the problem statements for translating a natural language question (NLQ) to SQL using a supervised machine learning approach such as neural networks. In particular, we can distinguish between two types of sub problems: (a) NLQs that require synthesizing a SQL-statement by predicting only information that is contained in the *database schema*, i.e. tables and columns. (b) NLQs that require synthesizing a SQL-statement by predicting also values that are contained in the *base data*. In summary, the set of options for prediction in problem (b) is much larger than in problem (a).

Let us define the term *set of options* for predicting matching tables, columns and values using our running example query *How many pets are owned by French students that are older than 20?*. The set of options for predicting *tables* corresponds to the total number of tables contained in the database schema. For instance, in our example shown in Figure 1, we have three tables. The set of options for predicting *columns* is the total number columns of the database schema – which is 10 in our example. However, the set of options for predicting *values* cannot just be looked up in the database schema. Here, we potentially require an inverted index over each column of the entire database. Since the number of values is far greater than the number of tables and columns, predicting the right value is a computationally more complex problem than predicting the right table or column.

1) **Predicting Columns, Tables and SQL-Sketches***:* For predicting columns and tables, the set of options is given by the database schema. In principle, one can simply provide all table names and column names as input to a neural network such that it can learn the mapping between the question tokens and the database schema. As an example, have a look again at Figure 1. During the training phase, the neural network learns that the question tokens *"older than"* should be mapped to column *age* rather than e.g. to *stu_id*.

After predicting tables and columns, the next step is to predict a SQL-sketch which works slightly differently than predicting tables and columns. Although the neural network chooses from a finite set of options, this set changes dynamically. As an example, assume that the last chosen action in the SQL-sketch is an *Order* action. By definition of the SemQL 2.0 grammar (see Figure 2), the only possible options now are action *asc A* and *desc A*. The neural network can therefore only choose from options given by the SemQL 2.0 grammar. These options dynamically change depending on the preceding node in the SemQL 2.0 tree.

1. "How many pets are owned by French students that are older than 20?"
2. "Find all female students who study 'biology' as their major."
3. "Report the total number of students for each fourth-grade classroom."

Fig. 3. Three examples of values (blue) which are typically not directly derivable from the text. The yellow values, on the contrary, are directly derivable from the text.

2) **Predicting Values***:* In contrast to predicting columns, tables, and SQL-sketches, no fixed set of options exists for values. While one could assume that all possible values are contained in the question, this is not always the case. In Figure 3 we see three examples of values which are not typically directly derivable from the text. The terms *French students* are most probably referring to a student with the home country *France*. The term *female* might refer to students whose gender is equal to *'F'* and the fourth-grade classroom might refer to a table *classroom* with grade *4*.

1. "Find all routes that have destination John F Kennedy International Airport with a duration of more than 6 hours."
2. "Which start station had the most trips starting from the 9th of August 2010? Give me the name and id of the station."

Fig. 4. Examples of values which result in a large number of possible value candidates.

Even in cases where all information is available in the question, there is often a large number of possible candidates for a given value. In Figure 4 we see two such examples where a natural language question can have a large number of possible *value candidates* contained in the base data. While *John F Kennedy International Airport* definitely refers to a column containing airports, we have no idea how this value is stored in the database. It could be anything from *JFK* over

John F Kennedy to the full term *John F Kennedy International Airport*. Similarly, we do not know how the date in the second example, i.e. *9th of August 2010*, needs to be formatted to match the given value in the database, which could be stored as "2010-08-09".

The challenge in all these examples can be reduced to one single problem: unlike columns, tables, and SQL sketches, we do not know the set of options for a given value.

III. ValueNet: High Level Architecture

In this section we describe the high level architecture of ValueNet to translate a natural language question to SQL.

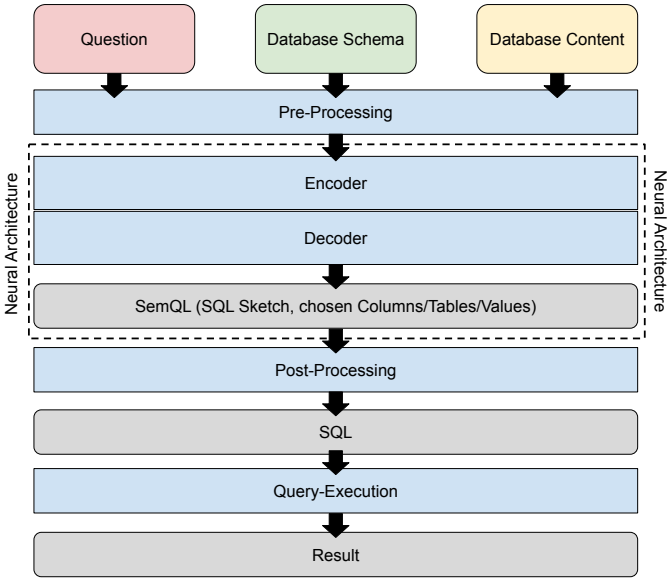

Fig. 5. ValueNet Architecture Overview.

Figure 5 shows the end-to-end process of our NL-to-SQL system. As input our system expects a question in natural language, the schema of the database, and access to the content of the database, e.g. via an inverted index as used in [4]. The output is a SQL statement, that once executed, returns the data the user asked for.

We will elaborate the steps of our architecture based on the initial example in Figure 1 with the question *How many pets are owned by French students that are older than 20?*

Note that previously published systems that were evaluated against the Spider dataset did not incorporate *values* such as "French" or "20". However, these systems would simply fill in a placeholder value (e.g. *'1'*) for each value, as the *Exact Matching Accuracy* of Spider does not validate values.

A. Pre-Processing

During pre-processing we try to achieve two tasks: The first task is to come up with good *value candidates*, which we describe in detail in Section IV. The second task is to come up with useful *hints* for predicting tables, columns and values. These hints are made available to the neural network, i.e. the subsequent components of our architecture, as an additional source of information besides the natural language question and the database schema. The intuition of this process is to give the neural network model "hints" which are easy to establish (e.g. by looking at the database content) and support it to take the correct decision when predicting SQL columns, tables, SQL-sketches and values.

1) **Question Hints***:* For each token in the question we try to figure out if it refers to a *table*, a *column*, a *value*, an *aggregation* or a *superlative*. See Figure 6 for an example on how we classify the tokens of a question. The intuition behind this is that those tokens are probably the most important ones when synthesizing the query.

For now we do not use any advanced NLP methods to find matches between question tokens and schema information - we simply apply stemming to all words and look for exact matches. This process can be improved as part of future work by using word embeddings and other advanced techniques. Keep in mind though, that this pre-processing is just a simple way to provide *prior knowledge* to the neural model. More complex relationships, like e.g. the fact that the token *older* in Figure 6 refers most probably to the column *age* in table *Student*, are established by the encoder part of our neural network.

Fig. 6. Classify question tokens by finding matches in the schema (for columns/tables) or database content (for values).

2) **Schema Hints***:* Schema hints are basically the inverse version of the *Question Hints*. We want to know if a column or table has been mentioned in the natural language question. Again, the intuition is to give hints to the neural network about the importance of a column/table. See Figure 7 for an example. If a table or a column matches exactly with a token in the question (e.g. in Figure 1 for *"pet"* and *"student"*) we classify it as an *exact match*. If it is only a partial match (e.g. for the token *"pet"* and table *Has_Pet*), we classify it as such. A special case is the class *value candidate match*: We apply this class if a value candidate has been found in the database in a specific column. As an example take the value token *"20"* of our running example. As we found it in column *Student.age*, we classify this column as a *value candidate match*.

B. Neural Architecture

The next two process steps, *Encoder* and *Decoder*, are what we call the *Neural Architecture*. This component is the core of our NL-to-SQL system. As input it receives the *question*, *schema* and *hints* from the pre-processing step and synthesizes a query represented in *SemQL 2.0*.

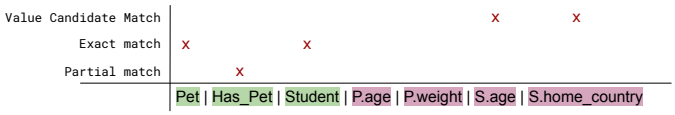

Fig. 7. Schema Hints, based on matches between tables/columns and the tokens in the question. This example refers to a subset of the schema in Figure 1.

In particular, the encoder tries to *encode information about the question and the schema* into a low dimensional space. Based on the encoded information, the decoder then tries to *synthesize a valid query* step by step. Applied to our example, we want the Neural Architecture to learn that e.g. *"How many"* should be translated to `SELECT count()` and that the word *"older"* should refer to the column *Student.age*.

1) **Encoder***:* Our encoder is based on a *pre-trained transformer architecture* [33] which is used in most recent NL-to-SQL systems [16], [20], [36]. Transformer architectures have been used for different tasks such as natural language translation [33], natural language generation [28] and recently also for entity matching as part of data integration [6]. The intuition is that such attention-only architectures of transformers generate better representations of natural language sequences than classical recurrent neural networks (RNNs). Hence, transformer architectures often yield better results on many natural language processing tasks than conventional neural networks.

Our encoder is an extension of IRNet's encoder. The main difference is that our encoder receives not only information about the database schema but also *value candidates*, extracted from the database content. Hence, our encoder can also learn correlations between tokens of an NL question and the actual values in the database.

2) **Decoder***:* The decoder receives the question/table/column/value-encodings from the encoder as input and outputs a query represented as an AST. The decoder consists of an LSTM architecture [18] in combination with multiple Pointer Networks [34] to choose tables, columns and values.

C. Post-Processing

The post-processing step is mainly a deterministic SemQL2.0-to-SQL transformation. We also have to incorporate the selected values, which we will explain in more detail in Section IV.

1) **Translating SemQL2.0 to SQL***:* Transforming SemQL2.0 to SQL is done by traversing the SemQL2.0 syntax tree from its root to leaf nodes. Most actions of the SemQL2.0 grammar can be transformed directly to their SQL equivalent.

2) **Handling Relationships***:* Note that the original SemQL does not know about relationships but only about tables used in *Filter, Select, Order* and *Superlative* actions. The reason for this design choice is the fact that users most likely do not mention all the required tables in their questions. As an example take the database schema in Figure 1. When a user poses the question *"Give me all student names with pets older than 14 years"*, she mentions the tables *Pet* and *Student* as she needs them for *Filtering* and *Selection*. She will though most likely not mention the bridge table *Has_Pet*.

However, a proper SQL query needs to join all mentioned tables by using the bridge tables. A common approach is to transform the database schema into an undirected graph, where the vertexes are tables and edges are primary-key/foreign-key relationships. One can then build up all `JOIN`s deterministically by finding the shortest path between two known vertices (tables) by e.g. using the Dijkstra algorithm. As soon as we deal with more than two tables, an approximation algorithm for the *Steiner tree* [44] problem will solve the problem of connecting all N vertices by the shortest path even more elegantly.

Unfortunately, we found this approach to be too simplistic when we switched to the Spider *Execution Accuracy* metric. Note that the *Exact Matching Accuracy* does not validate which columns are used to join two tables. It is enough to correctly predict the tables of the join without specifying the `ON` clause (e.g. `A INNER JOIN B` instead of `A INNER JOIN B ON A.A = B.A`), which is the approach used by *IRNet*.

Obviously this does not hold true anymore when executing queries using the *Execution Accuracy* metric as we do. Here, leaving out the join restriction results in a cross join yielding a Cartesian product of all rows. We therefore extended the schema graph by incorporating the primary/foreign key columns for each relationship edge.

IV. ValueNet - Detailed Architecture

In this section we introduce *ValueNet light* and *ValueNet* – two novel NL-to-SQL systems which learn from database information by considering both the database schema and the base data. While *ValueNet light* assumes that a set of value options is provided, *ValueNet* builds up a set of options by itself. The main contributions of *ValueNet* and *ValueNet light* are a *novel pre-processing approach* and an *enhanced encoding* step (see architecture in Figure 5).

A. ValueNet light

Let us assume the true values in the database in the first sample sentence of Figure 4 are *'JFK'* and *'6'*. If we had an oracle that provided us a set of options with these values, our neural model would only have to pick the right value at the right time when synthesizing a query. This would work because the encoder most probably establishes more attention between *John F Kennedy International Airport* (the tokens in the natural language question) and *JFK* (the value option from the oracle) rather than between *John F Kennedy International Airport* and *'6'*.

This is what we do with *ValueNet light*. We assume that all values of a query have been established upfront and are now available as a set of options. How to establish the values is

not part of ValueNet light. One possible way to accomplish this, as the authors of Spider suggest [43], would be to interact with the user in a question-answer conversation flow in order to establish all needed information for a query.

For all experiments with *ValueNet light* we compile the set of value options from the ground truth for each given example. This approach complies with the *Execution Accuracy* metric of Spider. *ValueNet light* then encodes all these values as part of the input as visible in Figure 8 (blue value encodings). For instance, the (blue) values "JFK", "Flight", and "Destination" have established a strong connection (attention) with the question tokens "flights", "destinations" and "kennedy".

In the next step, the neural model selects the right value encodings while synthesizing a query.

In the deterministic post-processing step we format the value given the predicted data type of the column. If the column is, for example, of the type text, we add quotes to it. If it is of the type integer, we make sure a floating point is not provided. In the case that the SQL sketch predicts a *Filter* action of type *like*, we further extend the value with the SQL wildcard character %.

B. ValueNet

In contrast to *ValueNet light* we propose with *ValueNet* an end-to-end NL-to-SQL system which solves the harder problem where no value candidates are given upfront. To come up with value candidates we propose an architecture sketch consisting of the following steps:

- *Value Extraction*: Extracting values from the question by using named entity recognition (NER) and heuristics.
- *Value Candidate Generation:* Generating value candidates by searching similar values in the database and by manipulating the extracted values (e.g. n-grams).
- *Value Candidate Validation:* Reducing the set of value candidates by looking them up in the database.
- *Value Candidate Encoding:* Encoding the remaining candidates together with information about the tables and columns they have been found in. This is then the input for the neural architecture.
- *Neural Architecture:* Continuing similar to *ValueNet light* based on the architecture described in Section III.

We will now explain these steps in more detail.

1) **Value Extraction***:* Given a natural language question, we use two different named entity recognition (NER) models to extract potential values. As a first approach, we implement a custom NER model based on a transformer [33] architecture leveraging the popular Transformers [38] library. As a second approach we use a commercial NER API[3].

While a custom NER model has the advantage of being able to fine tune on specific domains or datasets, this approach poses the danger of overfitting on certain types of values. Using a commercial NER API reduces this risk, though obviously at the danger of worse results, as it is not specifically trained to the task at hand. Alternatives to this NER API are other popular off-the-shelf libraries for NER as e.g. the SpaCy [19] *EntityRecognizer*.

In addition to a stochastic NER model we suggest *deterministic heuristics* to extract some types of values. We use the following three simple heuristics to identify candidate values: (1) Content in quotes: e.g. *Whose head's name has the substring 'Ha'?*. (2) Capitalized terms: e.g. *Show all flight numbers with aircraft Airbus A340-300.* (3) Single letters: e.g. *When is the hire date for those employees whose first name does not contain the letter M?*

While a custom NER model can easily be trained to detect these types of values, heuristics allow for augmenting the results of an off-the-shelf solution.

2) **Value Candidate Generation***:* After extracting values from the question, we need to generate value candidates. While for numeric values the extracted value itself is most likely the only necessary candidate, the process of value candidate generation is essential for all *text-based value types*. For ValueNet we implemented three simple methods of value candidate generation – one based on *string similarity*, one based on *handcrafted heuristics* and one based on *n-grams*.

Generating value candidates through *similarity* to existing values in the database is trivial in theory but challenging to implement efficiently. To measure similarity between a text value extracted from the question and values in the database, one can use either classical text distances [35] or distances based on word embeddings [25]. One then only has to scan the database for values with a similarity above a certain threshold.

The need for an efficient implementation stems from the fact that this pre-processing step has to be executed at inference time of each question and its complexity is bound to the size of the database. As the users are, at that point, actively waiting for an answer, the generation of candidates should ideally take less than a second. By using smart indexes and computationally cheap methods for blocking/indexing [8], this effort can be optimized. We further use the Damerau–Levenshtein [9] distance to measure similarity between tokens because of its good trade-off between accuracy and run time.

A second way to generate value candidates is through *handcrafted heuristics*. This is necessary due to the fact that databases have a specific (but reoccurring) approach to implement certain data types. We currently use the following heuristics: (1) Classic *gender* values, for example, are often implemented as a VARCHAR(1) column with content *'F'* or *'M'*. (2) *Boolean* data types are often implemented by a numeric column with value 0 and 1. (3) *Ordinals* as e.g. in *"...fourth-grade students..."* are usually implemented as an integer column. (4) Months (e.g. *August*) are often part of a full date column. By using a wildcard (e.g. *8/%*) once can find them.

While such simple handcrafted heuristics do not generalize to every database, they are a good starting point to bootstrap a generative model which learns such patterns in a more dynamic way.

A third approach for value candidate generation is to use *n-grams*. We use this technique for all extracted values with

[3]https://cloud.google.com/natural-language/docs/analyzing-entities

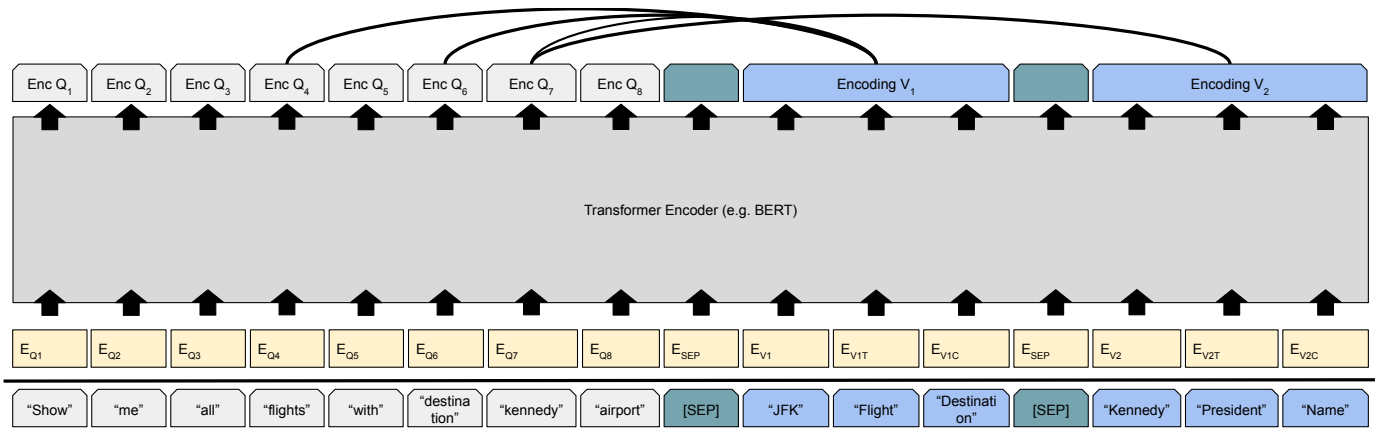

Fig. 8. Encoding of value candidates and question. The black connectors on top visualize the attention built up during encoding between question tokens and the actual values in the database. The large blue value encodings at the output of the encoder visualize the summarizing of a value together with its location (in a certain table column) by an LSTM. The encodings of columns and tables have been omitted for better readability.

more than one token. For example, a value like *'Kennedy International Airport'* generates one trigram, two bigrams, and three single words as value candidates.

3) **Value Candidate Validation***:* Depending on the (1) similarity thresholds, the (2) number of values extracted from the natural language question, and the (3) total number of values in the database, candidate generation might result in a large set of potential candidates. As we see in the experiments for *ValueNet light* and *ValueNet* in Section V, the number of candidates has a direct impact on the accuracy of the model - too many of them makes it harder for the model to choose the correct one. We therefore use the content of the database again in order to reduce candidates. In contrast to *Value Candidate Generation* we do not use similarity metrics, but rather require exact matches.

It is important to understand that we cannot validate all candidates in that way. Consider the following two examples: *'List the top 3 albums of Elton John in the Billboard charts'* and *'Find all albums of Elton John starting with "goodbye"'*. In these cases, we would not find *'3'* or *"goodbye"* in the database. In the first example, the value *3* is not part of the database but is used in the SQL query to limit the results. In the second example the token *"goodbye"* requires a wildcard match. Unfortunately, a wildcard match is not sufficient to validate a candidate, as it will provide too many false positives due to its flexibility. We therefore exclude numeric values and values extracted from quotes from the validation against the database.

During *Value Candidate Validation* we also register in which table and column a value candidate is found.

4) **Value Candidate Encoding***:* All steps so far serve the purpose of establishing a solid set of value candidates as input to our neural architecture. It is after all the neural network that decides which value to choose. The pre-processing only fulfills the purpose of extracting and generating reasonable candidates.

The candidate encoding works similarly to the encoding of tables and columns. However, here we encode the location (i.e. table and column) where we found a candidate together with the value itself.

As an example consider the question *"Show me all flights with destination Kennedy airport"* in Figure 8. The value we are looking for is *"JFK"*, which is contained in table *Flight*, column *Destination*. At the same time the term *"Kennedy"* also appears in other tables, e.g. in table *President*, column *Name*. Thanks to the additional table/column information, the encoder has the opportunity to build up attention (visualized by the black connectors) not only to the value itself, but also to the location where the value has been found. As this question contains the tokens *"flights"* and *"destination"*, the attention established with table *Flight* is higher than to the other value candidate with table *President* and column *Name*.

Each value candidate together with its location, is separated from the other values by using the designated *Separator* token of the encoder. Each value token is further tokenized in word pieces using the WordPiece [29] segmentation algorithm. The input for the encoder is then a list of pre-trained embeddings, one for each word piece.

5) Neural Architecture: After encoding the value candidates we continue similar to *ValueNet light* in Section IV-A. We use the neural architecture explained in Section III-B.

V. Experiments

In this section, we show the results of the experiments conducted with *ValueNet light* and *ValueNet* for translating natural language questions to SQL. In particular, we will address the following research questions with respect to NL-to-SQL systems: (1) How well do our approaches *ValueNet* and *ValueNet light* perform on the NL-to-SQL task incorporating values? (2) What is the difference in performance between *ValueNet light* which starts with a list of values and *ValueNet*, which must come up with a list of value candidates on its own?

We will also show that both *ValueNet light* and *ValueNet* perform similarly or even better than state-of-the-art systems that are evaluated on the *Execution Accuracy*.

A. Dataset

For our experiments we use the Spider [42] dataset which contains 10,181 natural language questions and their SQL equivalents. The queries have four levels of difficulty and contain most SQL operators (ORDER BY/GROUP BY/HAVING and nested queries). The queries are spread over 200 publicly available databases from 138 domains. Each database has multiple tables, with an average of 5.1 tables per database. The dataset is split into training set, validation (development) set and test set. The training set contains 8,659 queries, the validation set 1,034 queries. Note that we do not have access to the test set. The training set covers 146 databases while the validation set covers 20 different, i.e. unseen, databases. Hence, this dataset allows us to evaluate how well our two systems perform *transfer learning* between queries against databases in the training set and queries against *unseen databases* in the validation set.

We further analyzed the value distribution in the Spider dataset. We focused on the *train* split, which contains exactly 7,000 NL questions[4] for the 10,181 samples. 3,531 of the 7,000 sample questions contain values. Theses 3,531 sample questions contain a total of 4,690 values. See Figure 9 for a distribution of the values.

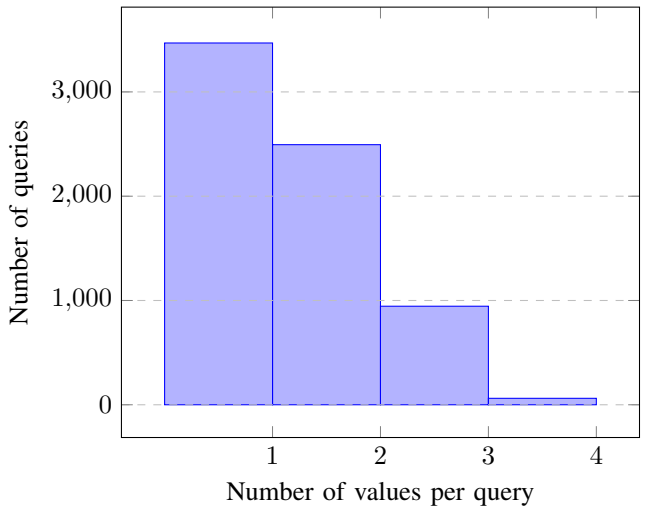

Fig. 9. Value distribution in the Spider data set. 3,469 samples contain no values. 2,494 samples contain one value, 945 two values, 62 three values and 30 samples contain 4 values.

1) Value Difficulty: The creators of the Spider dataset determined the difficulty level of a query without considering values. There is, however, a wide range of difficulty when it comes to extracting the correct values out of a question. We classify this difficulty into four levels:

- *Easy*: The value is clearly extractable by an NER system and is contained in the database as extracted. Example: *"How many pets are owned by students that are older than 20?"* where the value is *20*.
- *Medium*: The value is extractable by an NER system but might appear in a slightly different form in the database. Example: *"What are the rooms for members of the faculty who are professors"* where the value in the database is *Professor*.
- *Hard*: The value is extractable by an NER system but domain knowledge is needed to find the correct value. Example: *"Show all flight numbers from Los Angeles."* where the value in the database is *LAX*.
- *Extra hard*: The value is not explicitly recognizable as value and therefore hard to extract. Example: *"What are the names of nations where both English and French are official languages?"* where the values *English* and *French* can be directly extracted, but a third value in the database is `Language.IsOfficial = True`.

2) Value Types: We find the Spider dataset to contain a wide range of different values. The dataset includes but is not limited to: numeric values, strings and single characters, different ways of representing dates, times and duration, locations (e.g. addresses, countries, airports), specific codes (e.g. *Airbus-A740* or *CIS-220*), status (e.g. *successful* or *completed*) and Boolean, names and salutations as well as e-mail addresses. Despite some missing value types (e.g. phone numbers), we consider the variability of value types to be comparable to a real-world environment.

B. Evaluation Metrics

The Spider challenge[5] comes with two different evaluation metrics: *Exact Matching Accuracy* and *Execution Accuracy*. As we briefly explained in Section I, *Exact Matching Accuracy* compares the synthesized query to the gold query, while *Execution Accuracy* requires executing the synthesized query against a database and compares if the result is the same as when executing the gold query.

To the best of our knowledge, we introduce the first NL-to-SQL system which is evaluated via *Execution Accuracy* and whose source code is publicly available. Note that during the time of writing the paper, there are three systems that participated in the "Leaderboard - Execution with Values" with results from May 2020 (AuxNet + BART, BRIDGE + BERT and GAZP + BERT). However,as of now, these systems have not been published and the source code is not available, therefore, we are currently unable to reproduce their results. For this reason, we only compare our approaches against their reported accuracy in May 2020.

C. Experimental Setup

Hardware: All experiments were executed on a Tesla V100 GPU (32GB memory) with an Intel(R) Xeon(R) CPU E5-2650 v4 (4 cores) and 16GB memory.

[4]Although the training set of Spider already contains 7,000 novel questions, the authors of Spider added further 1,659 questions from existing datasets like e.g. IMDB adding up to a total of 8,659 question/query pairs to train on.

[5]https://yale-lily.github.io/spider

Frameworks: The experiments are implemented using PyTorch. We use the code of IRNet [16] as the base for our implementation. For the encoder model (Section III-B) we use the popular Transformer [38] library. For validation we use the official Spider validation script[6].

Implementation: In our implementation we provide a transformer encoder which can be configured to use any modern pre-trained transformer model like RoBERTa [24] or XLNet [40]. To produce comparable results with state-of-the-art systems, we use the default *BERT-Base* model for all experiments.

We further use bi-directional LSTM networks to summarize multi-token columns/tables/values (described in Section III-B1) with a dimensionality of 300. We use the same dimensionality for the decoder-network described in Section III-B2.

Moreover, we use an Adam [22] optimizer with three different learning rates: $2e{-}5$ for the encoder, $1e{-}3$ for the decoder and $1e{-}4$ for the connection parameters in between. We further use a dropout of 0.3 and a batch size of 20. The learning rates for the encoder are the default parameters for BERT fine-tuning, all other hyperparameters have been set based on an empirical hyperparameter sweep.

To reproduce our experiments we release all code including hyperparameters on Github[7].

D. Results

In this section we report the results of our two approaches *ValueNet light* and *ValueNet* on the Spider dataset using the *Execution Accuracy* metric. This score includes (in contrast to the *Exact Matching Accuracy* metric) the proper prediction of values.

As mentioned previously, there are currently no direct competitors using the *Execution Accuracy* metric since neither the code nor the binaries are available to re-produce their experiments. However, as the Spider leaderboard *Execution with Values* though contains three candidates without publications (*GAZP + BERT, BRIDGE + BERT, AuxNet + BART*), we add these experiments as single data points.

As we see in Figure 10 both *ValueNet* and *ValueNet light* outperform *GAZP + BERT* and *BRIDGE + BERT*. The more advanced model *AuxNet + BART* levels on score with our ValueNet implementation. However, *ValueNet light* also outperforms AuxNet + BART.

Note that we use the smallest version of BERT [12] as encoder, whereas AuxNet + BART uses a much more advanced pretrained language model as encoder, namely BART [23]. We therefore expect our results to be even higher when augmenting the encodings with BART instead of BERT.

[6]https://github.com/taoyds/spider

[7]https://github.com/brunnurs/valuenet

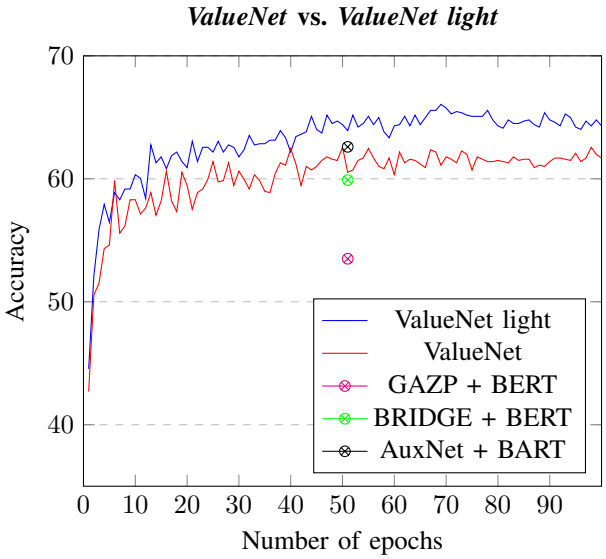

Fig. 10. The performance of *ValueNet light* and *ValueNet* on the Spider data set using the *Execution Accuracy* evaluation metric. We represent our competitors on the Spider leaderboard with three single values due to their unpublished papers/code. We only visualize accuracy scores in the range of 35 to 70% to emphasize the difference (see y-axis). The reported numbers are an average of five runs.

E. ValueNet vs. ValueNet light

As expected there is a performance gap between *ValueNet* and *ValueNet light* of 3%-4%. There are two possible reasons for this performance gap:

(1) *Non-extractable values*: While in *ValueNet light* there is a list of all used values provided, *ValueNet* needs to extract these values first from the question as described in Section IV. Let us understand how many values we lose during that process and keep in mind that each value, that we cannot extract, will result in a failed sample. For the train split of the Spider dataset, which includes 3,531 samples containing values (a total of 4,690 values), we manage to extract all values for 3,200 samples. That means *ValueNet* is capable of extracting around 90% of all values. This share of extractable values stays constant for the validation dataset.

Referring to the value difficulty of Section V-A1 we found that almost all of the remaining 10% not found values belong to the difficulty classes *Hard* and *Extra Hard*. For instance, in the question *"What are the full names of all left handed players?"* we failed to extract the value *'L'* which would match the table/column *players.hand*.

(2) *(Too) Many value candidates*: *ValueNet light* is provided with a list of exact values for a sample query and then has to select each of them at the right time when synthesizing the query. If we revisit Figure 9, we see the distribution of values for all queries in the dataset. We can observe that the maximum number of values a sample contains is 4. We also see that the majority of samples has only 1 or 2 values.

F. Results by Difficulty

The Spider evaluation metric defines the difficulty based on the number of SQL components, selections,

and conditions, so that queries that contain more SQL keywords (**GROUP BY**, **ORDER BY**, **INTERSECT**, nested subqueries, column selections and aggregators, etc) are considered to be harder. Spider defines 4 levels of difficulty: *Easy, Medium, Hard* and *Extra hard*. Many of the *Hard/Extra hard* queries contain multiple SQL keywords in combination with nested subqueries. We now want to analyze the accuracy of translating from NL to SQL with respect to the difficulty of the query.

TABLE I
ACCURACY FOR DIFFERENT TYPES OF QUERIES GROUPED BY DIFFICULTY.

Difficulty	Accuracy
Easy	0.77
Medium	0.62
Hard	0.57
Extra-Hard	0.43

Table I shows the translation accuracy of *ValueNet* grouped by query difficulty. We can see that for easy queries, *ValueNet* achieves an average accuracy of 77%. For extra-hard queries, the average accuracy drops to 43%.

G. Error Analysis

We analyzed the 352 failed examples of ValueNet on the development set. For about 50% of all errors (176 samples) we did a thorough manual analysis and found the following main causes of errors. Be aware that multiple error causes can appear per example.

Column and Table Prediction: In 50% of all analyzed errors ValueNet fails to predict the correct column. In around 25% it chooses a column from another table, therefore the table prediction is also incorrect. The main reason for those errors is that columns across different tables have similar or even identical names and are thus hard to distinguish. Examples for such column names are *name*, *id* or *description*, which appear usually in multiple tables. ValueNet, similar to IRNET [16], struggles with such cases. Incorporating a more sophisticated schema linking approach, as for example proposed by RAT-Sql [36], might help to prevent such errors.

Errors in the SQL Sketch: In 39% of all analyzed cases we find errors in the SQL sketch, i.e. the logical form of the query. It is important to note though that the majority (76%) of these errors appear in queries classified as *Hard* or *Extra Hard* by the Spider authors. It is further interesting to see that we did not find a completely incorrect SQL sketch in any of the analyzed examples. We frequently found that the SQL Sketch was 80% correct but included a minor mistake.

A clear pattern is hard to establish. Some of the *Hard* and *Extra Hard* examples require advanced common knowledge which is hard to incorporate into a model. However, some of the failed examples on lower difficulties could easily be solved with domain-specific training (e.g. "oldest player" is incorrectly interpreted as **ORDER BY** `birthdate` **ASC** rather than **DESC**).

Value Selection: In 9% of all analyzed errors ValueNet selects the wrong value. Note that a third of these cases leads back to one single value – a company name called *"JetBlue Airways"*. We assume that domain-specific fine tuning of the encoder on the database content could avoid such errors.

False Negatives: 9% of all reported errors are false negatives. These examples range from missing or wrong data in the provided databases to mismatches between the question and the ground truth query. One common mistake is e.g. a missing table in the query, even though it is clearly stated in the question.

In many cases, it is debatable if a question really gives a hint for a specific SQL clause or not. One example is the keyword **DISTINCT**, which is often hard to derive from a question if not specifically hinted. In this case, a more advanced error metric might be able to classify the degree of error.

H. End-to-End Performance: Translation Time

An important question to answer is how long it takes *ValueNet* to synthesize a query given an question. At inference time this process happens interactively while the user waits for a result. The process therefore needs to be performant.

TABLE II
TRANSLATION TIME

Step	Average Time [ms]	Standard Deviation [ms]
Pre-Processing	80	5
Value lookup	234	43
Encoder/Decoder	76	14
Post-Processing	13	2
Query-Execution	15	3

In Table II we see the total translation time, split up by the process steps defined in Figure 5. The duration was measured over the execution of all 1,034 samples of the validation set. The overall translation time per query is on average 418 ms and therefore sufficiently fast for interactive usage. For use in larger databases, the process can further be improved by using an advanced inverted index when looking up values, as this step consumes by far the largest amount of time.

VI. RELATED WORK

A. NL-to-SQL

Since the end of the 1970s, building natural language interfaces for databases (NLIDBs) has been a significant challenge. Many of the early work [2], [26], [37] focused on rule-based approaches with handcrafted features. Later systems enabled users to query the databases with *simple keywords* [3], [4], [31]. The next step in the development was to enable processing more complex natural language questions by applying a *pattern-based approach* [10], [45]. Moreover, *grammar-based* approaches were introduced to improve the precision of natural language interfaces by restricting users to formulate queries according to certain pre-defined rules [14], [32]. Finally, a comprehensive overview of NLIDB systems is given in [1].

While most of these approaches work well when customized for a specific database (with a restricted set of keywords or natural language patterns), they are often not competitive in

a cross-domain setting with complex questions. One of the most advanced systems is currently Athena++ [30]. However, neither the source code nor the binaries of that system are publicly available for reproducing the results.

More recent approaches use advanced neural network architectures to synthesize SQL queries given a user question. The work of [13] uses a classical encoder-decoder architecture based on Long Short Term Memory (LSTM) networks [18]. *Seq2SQL* [46] adds a reinforcement learning approach to learn query-generation policies. That system creates a reward signal by executing queries against the database in-the-loop. *SQLova* [20] is the first work to use a transformer-based encoder [33] to solve the WikiSQL [46] challenge.

The Spider [42] dataset, which covers 200 databases and more than 10,0000 training data samples, is currently considered the de-facto standard for evaluating NLIDBs (or NL-to-SQL-systems) based on machine learning approaches. A recent approach [11] introduces a novel framework for generating training data by inverting the data annotation process. The advantage of this approach is to generate training data more quickly and to cover a wider range of queries.

Let us now focus on recent systems that use the Spider dataset for evaluating the accuracy of generating SQL given a natural language question. IRNet [16], for instance, uses a transformer encoder and a decoder based on an LSTM network. It further introduces an intermediate representation based on an abstract syntax tree as an alternative to directly synthesizing SQL. The main goal of this intermediate representation is the abstraction of SQL-specific implementation details. In our work we use and extend this approach to handle natural language queries that incorporate values, i.e. that require analyzing the base data of the database.

RAT-SQL [36] is another NL-to-SQL system that achieved state-of-the-art results on the Spider challenge. It focuses on the problem of *schema encoding* and *schema linking*. The work proposes a new relation-aware self-attention mechanism based on transformers with promising results on non-trivial database schemas.

A good overview about the performance of the above-mentioned systems is given in [21]. The paper conducts a detailed experimental evaluation of both traditional and neural network-based systems whose source code or binaries are available for reproducibility studies. The paper concludes that there is still significant potential for improving current state-of-the-art systems to work in real-world environments.

B. Finding Matching Database Values

The importance of values and the idea of finding correct values through database lookups was already known in works based on the WikiSQL challenge as the meta paper [41] shows. With the Spider [42] challenge providing values in around 50% of its training data samples, it is an ideal dataset to continue working on the challenge of building a real world NL-to-SQL system incorporating values. Unfortunately most works [5], [16], [36] on the Spider challenge chose an evaluation metric that does not consider values. Hence, with our two approaches presented in this paper, we deliver an end-to-end NL-to-SQL system incorporating values and hope to motivate further work to solve this challenge. Moreover, we provide the source code of our system such that the results can be reproduced by other researchers.

VII. Conclusions & Future Work

In this work we propose *ValueNet light* and *ValueNet* – two end-to-end NL-to-SQL systems incorporating values. We evaluate them on the Spider dataset and demonstrate that incorporating values does not affect the accuracy of translating from natural language to SQL negatively. We achieve state-of-the-art results and provide, to our knowledge, the first detailed discussion and source code for an NL-to-SQL system that *synthesizes a full SQL query including values* on the challenging Spider dataset

In particular, with *ValueNet* we propose an architecture sketch to come up with good value candidates. These value candidates are then incorporated into the end-to-end query translation process through the neural model. Generating good value candidates is difficult as the values extracted from a question often differ from the actual values in the database. We thus use the database content, in combination with a generative approach, to produce promising value candidates. Finally, the neural network decides which of these value candidates is the best match for the intention of the natural language question from the user.

Moreover, *ValueNet* is a system that synthesizes complete queries which can be executed against a database. In contrast to many recent works, we provide an approach which can be used in a real-world scenario. We hope to motivate further papers on solving this challenge to use the Spider *Execution Accuracy* metric.

As part of future work, we will further improve the architecture sketch of how to come up with good value candidates. One possible avenue of research is to apply a generative neural network approach (e.g. based on text GANs [15]) in combination with the available data from the database.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 863410. We thank Kate Kosten for linguistic improvements of the paper as well as Ana Sima and Georgia Koutrika for fruitful discussions.

References

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

[2] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *CoRR*, cmp-lg/9503016, 1995.

[3] H. Bast and E. Haussmann. More accurate question answering on freebase. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1431–1440. ACM, 2015.

[4] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *Proceedings of the VLDB Endowment*, 5(10):932–943, 2012.

[5] B. Bogin, M. Gardner, and J. Berant. Representing schema structure with graph neural networks for text-to-sql parsing. *CoRR*, abs/1905.06241, 2019.

[6] U. Brunner and K. Stockinger. Entity matching with transformer architectures-a step forward in data integration. In *International Conference on Extending Database Technology, Copenhagen, 30 March-2 April 2020*, 2020.

[7] J. Cheng, S. Reddy, V. Saraswat, and M. Lapata. Learning an executable neural semantic parser. *Computational Linguistics*, 45(1):59–94, 2019.

[8] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012.

[9] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar. 1964.

[10] D. Damljanovic, M. Agatonovic, and H. Cunningham. Natural language interfaces to ontologies: Combining syntactic analysis and ontology-based lookup through the user interaction. In *Extended Semantic Web Conference*, pages 106–120. Springer, 2010.

[11] J. Deriu, K. Mlynchyk, P. Schläpfer, A. Rodrigo, D. von Grünigen, N. Kaiser, K. Stockinger, E. Agirre, and M. Cieliebak. A methodology for creating question answering corpora using inverse data annotation. In *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.

[12] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[13] L. Dong and M. Lapata. Language to logical form with neural attention. *CoRR*, abs/1601.01280, 2016.

[14] S. Ferré. Sparklis: an expressive query builder for sparql endpoints with guidance in natural language. *Semantic Web*, 8(3):405–418, 2017.

[15] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[16] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J. Lou, T. Liu, and D. Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. *CoRR*, abs/1905.08205, 2019.

[17] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy, July 2019. Association for Computational Linguistics.

[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] M. Honnibal and I. Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[20] W. Hwang, J. Yim, S. Park, and M. Seo. A comprehensive exploration on wikisql with table-aware word contextualization. *CoRR*, abs/1902.01069, 2019.

[21] H. Kim, B.-H. So, W.-S. Han, and H. Lee. Natural language to sql: Where are we today? *Proceedings of the VLDB Endowment*, 13(10):1737–1750, 2020.

[22] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[23] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[24] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[26] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING '04, page 141–es, USA, 2004. Association for Computational Linguistics.

[27] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, IUI '03, page 149–157, New York, NY, USA, 2003. Association for Computing Machinery.

[28] S. Radford, Narasimhan and Sutskever. Improving language understanding by generative pre-training. *Technical report, OpenAI*, 2018.

[29] M. Schuster and K. Nakajima. Japanese and korean voice search. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pages 5149–5152, 03 2012.

[30] J. Sen, C. Lei, A. Quamar, F. Özcan, V. Efthymiou, A. Dalmia, G. Stager, A. Mittal, D. Saha, and K. Sankaranarayanan. Athena++ natural language querying for complex nested sql queries. *Proceedings of the VLDB Endowment*, 13(12):2747–2759, 2020.

[31] A. Simitsis, G. Koutrika, and Y. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *The VLDB Journal—The International Journal on Very Large Data Bases*, 17(1):117–149, 2008.

[32] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, et al. Tr discover: A natural language interface for querying and analyzing interlinked datasets. In *International Semantic Web Conference*, pages 21–37. Springer, 2015.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[34] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015.

[35] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[36] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*, 2019.

[37] D. H. D. Warren and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *Comput. Linguist.*, 8(3–4):110–122, July 1982.

[38] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[39] X. Xu, C. Liu, and D. Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

[40] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

[41] S. Yavuz, I. Gur, Y. Su, and X. Yan. What it takes to achieve 100% condition accuracy on WikiSQL. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1702–1711, Brussels, Belgium, Oct.-Nov. 2018. Association for Computational Linguistics.

[42] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *CoRR*, abs/1809.08887, 2018.

[43] T. Yu, R. Zhang, M. Yasunaga, Y. C. Tan, X. V. Lin, S. Li, H. Er, I. Li, B. Pang, T. Chen, E. Ji, S. Dixit, D. Proctor, S. Shim, J. Kraft, V. Zhang, C. Xiong, R. Socher, and D. R. Radev. Sparc: Cross-domain semantic parsing in context. *CoRR*, abs/1906.02285, 2019.

[44] A. Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9:463–470, 05 1993.

[45] W. Zheng, H. Cheng, L. Zou, J. X. Yu, and K. Zhao. Natural language question/answering: Let users talk with the knowledge graph. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 217–226. ACM, 2017.

[46] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.