ValueNet: A Neural Text-to-SQL Architecture Incorporating Values

Ursin Brunner
Zurich University of Applied Sciences
Switzerland
ursin.brunner@zhaw.ch

Kurt Stockinger
Zurich University of Applied Sciences
Switzerland
kurt.stockinger@zhaw.ch

ABSTRACT

Building natural language interfaces for databases has been a long-standing challenge for several decades. The major advantage of these so-called text-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 text-to-SQL systems achieve promising results on this dataset. However, none of them extracts and incorporates 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* – the first end-to-end text-to-SQL system incorporating values on the challenging Spider dataset. The main idea of our approach is to use not only metadata information about 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 64% and 60% accuracy, respectively, for translating from text to SQL, even when applying this more difficult metric than used by previous work.

PVLDB Reference Format:
Ursin Brunner and Kurt Stockinger. ValueNet: A Neural Text-to-SQL Architecture Incorporating Values. *PVLDB*, xxx(xxx): xxxx-yyyy, xxx.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Proceedings of the VLDB Endowment, Vol. xxx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Building text-to-SQL systems has been a long-standing challenge of 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[25].

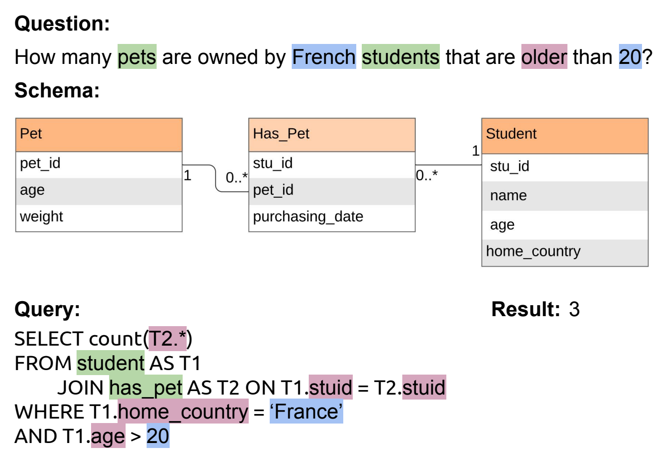


Figure 1: A typical text-to-SQL system synthesizes and executes a SQL query given a natural language question and a database schema. Besides synthesizing a valid SQL sketch (non-highlighted words), the system has to select the correct tables *(green)*, columns *(red)* and values *(blue)*

.

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.

In Figure 1 we see the typical text-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 column and base data value* (e.g. 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 is an important open gap in the current state-of-the-art of neural network-based text-to-SQL systems.

Over the last years, a large body of research focused on new text-to-SQL architectures [1]. With the introduction of large-scale datasets like *WikiSQL*[46], advanced neural approaches started to become popular. These approaches often used an encoder-decoder architecture to train a neural network end-to-end on a large amount of question/query pairs.

WikiSQL comprises 80,000 question/query pairs and enables the use of such neural architectures. However, *WikiSQL* lacks the complexity of real world scenarios. With only one table per database (and therefore no `JOIN`s), as well as the lack of advanced operators like `ORDER BY`, `GROUP BY` or nested queries, simple sketch-based decoders such as e.g. SQLova[19] started soon to reach accuracy scores of 90% and more on the test set.

The *Spider*[42] challenge released by the end of 2018 poses a new interesting challenge in the field of text-to-SQL. Not only does *Spider* overcome the shortcomings of *WikiSQL* with respect to complex queries, it also requires the participants to implement a text-to-SQL system handling unseen databases. Providing a total of 200 databases over 138 domains, the participants have to avoid overfitting on specific domains or known database schemas in order to generalize well on unseen questions and database schemas.

The release of *Spider* motivated several research groups to provide contributions in 2019/2020 and has become the de-facto standard for evaluating such systems. While classical neural architectures[13, 38] achieved rather low accuracy scores on the *Spider* leaderboard[2] (4.8%, 12.4%), more advanced approaches like *IRNet*[16] and *RAT-SQL*[35] 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.

- **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 text-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.

By today, all *Spider* contributions focus on the *Exact Matching Accuracy* evaluation (visible on the "Exact Set Match without Values" leaderboard). While a good score on this leaderboard is definitely an achievement (it requires a system to not only predict a complex SQL-sketch, but also to select the right tables and columns) it is still a simplification from a fully fledged text-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 the paper.

In this paper we make the following contributions:

- To the best of our knowledge, we introduce the first text-to-SQL system to synthesize a full query including values on the challenging Spider dataset. Our approach is thus a step forward in building an end-to-end-system that is not only of theoretical value but also applicable for real-world scenarios. We achieve an accuracy of 63% on the Spider *Execution Accuracy* metric. Current top solutions on the easier *Exact Matching Accuracy* metric achieve results in the range of 60% - 65%.

- We provide two novel text-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) *Value Net* - An end-to-end architecture which extracts and generates value candidates given only natural language questions and the database content. *Value Net* 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 review the related work. In Section 3 we define the problem of text-to-SQL in more detail. In Section 4 in describe our end-to-end neural architecture and focus on ValueNet light and ValueNet in Section 5. Our experimental results are given in Section 6. Finally, we draw conclusions in Section 7.

2. RELATED WORK

2.1 Text-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 [36, 2, 24] focused on

[1]Select-Project-Join
[2]https://yale-lily.github.io/spider

rule-based approaches with handcrafted features. Later systems enabled users to query the databases with *simple keywords* [30, 4, 3]. 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[31, 14]. A comprehensive overview of these 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.

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. *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*[19] is the first work to use a transformer-based encoder [32] 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 text-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[35] is another text-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.

2.2 Transformer Architectures

Based on the well-known paper "Attention is all you need" [32], transformers - a special type of neural network - started in mid 2017 to become one of the most promising techniques for natural language processing (NLP). One of the earliest systems was BERT [12] which combined the transformer architecture with massive unsupervised pre-training. BERT was the first approach to achieve state-of-the-art results in a large number of NLP tasks. Succeeding works by [39, 22] achieved even higher results on NLP benchmarks, though mainly by using more training data, larger neural network architectures and novel pre-training approaches.

While transformers are undeniably among the most significant achievements in the recent NLP landscape, they have also started a controversy about the *more data/larger models/more computational power* mentality. Therefore, in addition to increasing model sizes like the authors of [29] did, research started also to develop leaner transformer models, which can be used on mobile devices or non-GPU servers at inference time [27, 21].

Transformer architectures have been used for different tasks such as natural language translation [32], natural language generation [26] and recently also for entity matching as part of data integration [6].

2.3 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 [40] 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 text-to-SQL system incorporating values. Unfortunately most works [16, 35, 5] 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 text-to-SQL system incorporating values and hope to motivate further work to solve this challenge.

3. PROBLEM DEFINITION

3.1 Intermediate Representation

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 these 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 to directly synthesize a SQL query from it. Hence, abstracting the details of SQL enables the system to understand the question/query pairs more reliably as shown previously [17, 7]. Another advantage is that intermediate representations 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*.

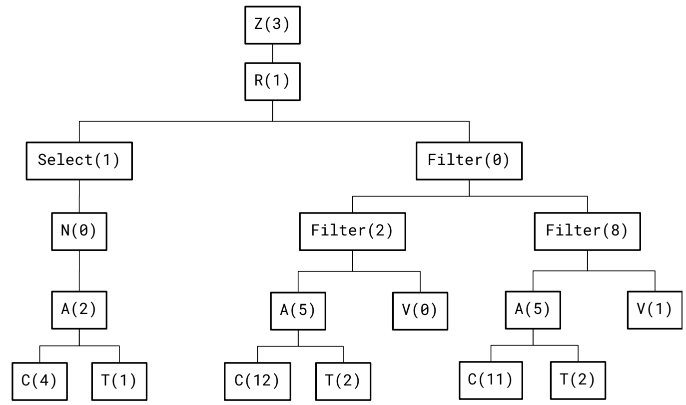
Figure 2 shows the complete grammar for SemQL 2.0 that can handle the major relational operators such as *select, project, join, union, intersect*, etc. We extend the grammar introduced in *IRnet*[16] mainly by the value representation *V* - highlighted in the figure in green.

Figure 3 shows the abstract syntax tree representing our example query. In the inner nodes the AST query contain what we call the *SQL sketch* - the logical form of the query. It contains information about when to *filter, group, select* and other operations. In the leaf nodes, the tree contains the information about which table/column/value to select at which point.

As an example see Figure 3 where we use the value representation *V* to refer to the values *20* and *France*. For instance, in the right branch of the tree we see the *Filter(8)* which represents the SQL constraint `WHERE age > 20` of our initial example. The child nodes *C(11)* and *T(2)*, refer to

```
           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
```

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



Index	Table
0	Pet
1	Has_Pet
2	Students

Index	Column
4	Has_Pet.*
11	age
12	home_country

Index	Value
0	"France"
1	20

Legend:
Z = Multi Query Root
R = Query Root
N = Number of Attributes
A = Attribute
C = Column
T = Table
V = Value

Figure 3: The example query from Figure 1 as AST with SemQL 2.0 grammar. The numbers next to each action refer to the implementation of this action. For instance, *Filter(0)* refers to an *AND*-filter. The index of each leaf node refers to a table, column or value, respectively.

the column *age* of table *Students*. The child node *V(1)* of this filter refers to the value *20*.

Note that V always represents a *value candidate*, i.e. a potential value that is contained in the base data, and is mostly used in filter criteria to compare a value to a column. We further use V in the *Superlative* action, for example, if someone asks the question *"List the three highest mountains in the Netherlands"*. In that case we need V to represent the value *3*.

Finally, we also extended SemQL by the *Distinct*-keyword to be able to correctly answer questions such as *"What are all the different book publishers?"*.

3.2 Problem Statements

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) are 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* is all entries of the green table show in Figure 3. The set of options for predicting *columns* is all entries of the red table. Both sets of options are created by iterating through the database schema. The blue table though, the set of options for predicting *values*, cannot just be looked up in the database schema. To understand this difference we will first analyze how columns, tables and the SQL-sketch are predicted. Afterwards we describe how predicting values deviates from this approach.

3.2.1 Predicting Columns, Tables and SQL-Sketches

For predicting columns and tables, the set of options is given by the database schema. For any small or mid-sized database, 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.

3.2.2 *Predicting Values*

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."

Figure 4: 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.

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 4 we see three examples of values which are not typically directly derivable from the text. The terms *French student* are most probably referring to a student with the home country *France*. The term *female* might refer to students whose sex is equal to *'F'* and the forth-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."

Figure 5: 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 5 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.

4. END-TO-END ARCHITECTURE

In this section we describe the architecture that serves as the foundation for both *ValueNet light* and *ValueNet* to translate natural language questions to SQL.

Figure 6 shows the end-to-end process of our text-to-SQL system. The architecture essentially consists of 3 major components: (1) Pre-Processing (2) Neural Architecture (3) Post-Processing. The main contributions of our paper take place in the pre-processing step (highlighted in red). As input our system expects a question in natural language, the schema of the database, and access to the content of the database, i.e. the base data. The output is a SQL statement, that once executed, returns the data the user asked for.

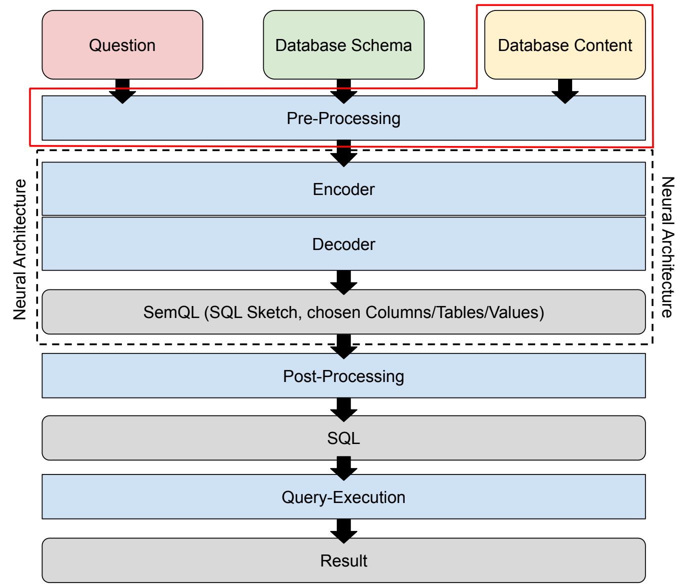


Figure 6: Architecture Overview. The main contributions of *ValueNet* take place in the pre-processing step.

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 previous systems that were evaluated against the Spider dataset did 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. In short, these systems did not generate a completely correct SQL statement.

4.1 Pre-Processing

In the pre-processing step, we try to find simple matches between tokens in the natural language question and the corresponding tables and columns of the *database schema*. In our example, such a simple match would be between the question token *pets* and the corresponding table *Pet*.

Different to previous work, our approach not only uses the database schema but also the *database content* (base data) to find matches. In our example, the system finds the value e.g. *20* in the table/column *Student.age*.

In order to evaluate the query tokens "French students", we further use *value candidates* to find correct matches in the database. This allows us to find in which tables and columns a value candidate is identified. In our example, the correct table is "home_country" and the correct value is "France".

The matching information of value candidates is made available to the neural network , i.e. the subsequent components of our architecture, as an additional source of information besides the 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.

4.1.1 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 7 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 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 input knowledge to the neural model. More complex relationships, like e.g. the fact that the token *older* in Figure 7 refers most probably to the column *age* in table *Student*, must be established by the neural network.

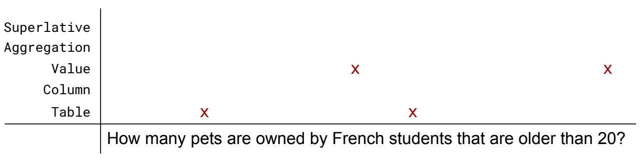


Figure 7: Classify question tokens by finding matches in the schema (for columns/tables) or database content (for values)

4.1.2 Schema Hints

Schema hints are basically the inverse version of the *Question Hints*. We want to know if each column or table matches with a token in the question. The intuition is again give hints to the neural network about the importance of a column/table. See Figure 8 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"* in the initial example. As we found it in column *Student.age*, we classify this column as a *value candidate match*.

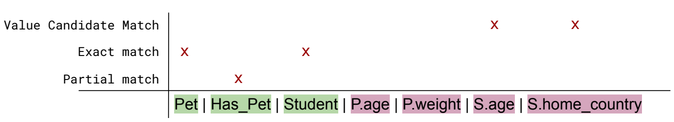


Figure 8: Schema Hints, based on matches between tables/columns and the tokens in the question. This example refers to a subset of the schema in Figure 1.

4.2 Neural Architecture

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

The encoder tries to *encode information about the question and the schema* into a low dimensional space. The decoder 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*.

4.2.1 Encoder

Our encoder is based on a *pre-trained transformer architecture* [32] which is commonly used in recent text-to-SQL systems [19, 16, 35]. 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.

A critical task when using transformer encoders is to correctly encode the input. See Figure 9 for an example. The input in our case is the question (already tokenized), the schema (table- and column names already tokenized).

4.2.2 Decoder

The decoder receives the question/table/column/value-encodings from the encoder as input and outputs a query represented as an Abstract Syntax Tree (AST).The decoder consists of an LSTM architecture in combination with multiple Pointer Networks[33] to choose tables, columns and values.

In more detail, the decoder model synthesizes an AST query by sequentially choosing from a set of possible action options. The action options are given by the SemQL syntax defined in Figure 2. For an example, let us have a look at the AST in Figure 3: At time step 0, the decoder can choose between all possible implementations of the Z action. The Z action represents a *multi query root* which is needed if a query contains a top level operation like a *union*, an *intersect* or an *except*. As this query does not require any of them, the decoder should with high probability choose *Z(3)* which is a query Root R without any top level operation. At time step 1 and with the history of choosing action R before, the decoder has to choose one out of all six possible action implementations for R. An example implementation of R is e.g. *Select Order Filter*, for all others see Figure 2. If the decoder performs correctly, it chooses *R(1)*, which is *Select Filter*.

This process continues until all branches of the abstract syntax tree are closed with a leaf node. In our example in Figure 3 these leaf nodes are for e.g. the left branch a column C and a table T. Afterwards, for each leaf node the decoder has to select an element: if the leaf node e.g. is a table T, the decoder has to select from the set of tables provided by the encoder. The same happens for columns C and values V.

In Figure 3 this selection is visible in each leaf node of the tree. The index of e.g. leaf node *V(1)* in the bottom right is referring to the value with index 1 in the blue table.

The probability of synthesizing the correct query z can be formalized as:

$$p(z|q,s,v) = \prod_{i=1}^{T} p(a_i|q,s,v,a_{<i}),$$

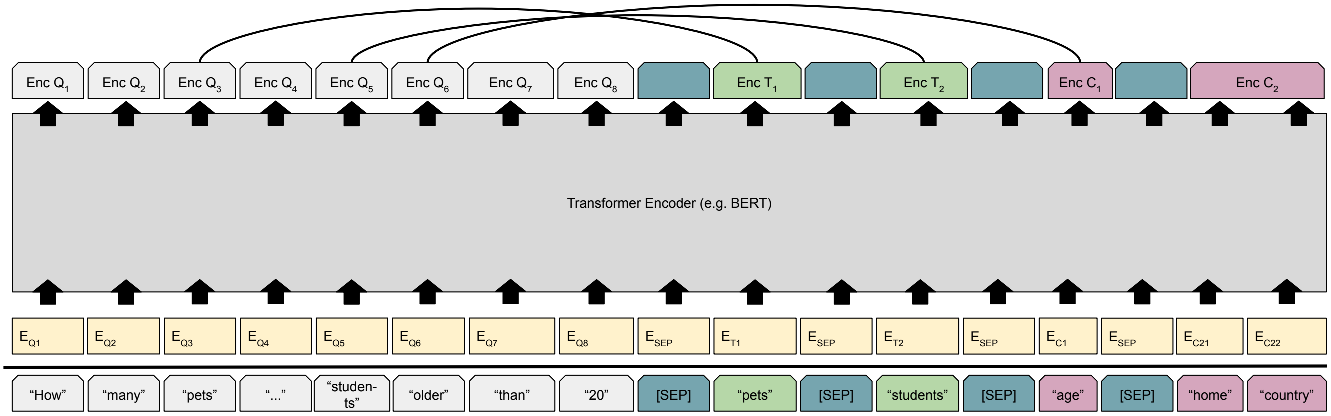


Figure 9: Encoding of natural language question and database schema tokens. Each color visualizes a token type: light gray for question tokens, light green for table tokens, red for column tokens and dark green for special tokens such as separators. The black connectors on top visualize the attention built up during encoding between question- and schema tokens, e.g. between question token "older" and column token "age". The yellow boxes represent word embeddings.

where q and s are the encoded question and schema, v represents the values, a_i is the action chosen at time step i and $a_{<i}$ is the history of actions up to time step i.

We approximate this equation similar to *TRANX*[41] and *IRNet*[16] through an LSTM network. As input, the network has access to the question encodings and the question hints (see Section 4.1.2). At each time step the hidden state of the LSTM network is a combination of the last chosen action (a_{i-1}), an activated attention vector over the input encodings ($\widetilde{s}_{i-1}$) and a parent feeding vector p_i. Interested readers can refer to *TRANX*[41] for more details about *parent feeding* and the decoder LSTM in general.

4.2.3 Encoding Schema- and Question Hints

One interesting detail is how to encode the *question hints* and *schema hints* (see Section 4.1.1 and 4.1.2) in such a way, that the neural model can use them best in its decision making process.

The *IRNet*[16] implementation adds the token classification information (revisit Figure 9) as a simple additional input token to the encoder (e.g. instead of *"pets"* it would be *["table", "pets"]*) and averages the two outputs of the encoder to get one encoding for the token *"pets"*.

In our experiments we found that it is superior to add the token class information only after the encoding with a trainable embedding with a length of 5 (one for each class). That way the neural network can decide itself during training if it uses these "hints" and how much weight to put on them. The neural network can even learn to not use these hints at all by setting all five embeddings to the same value during training.

Similar to *question hints*, the *schema hints* are added to the table/column encodings only after the encoder. Also here we use trainable embeddings so the neural network can decide itself how much it incorporates these hints.

4.3 Post-Processing

The post-processing step is mainly a deterministic SemQL-to-SQL transformation. We also have to incorporate the selected values, which we will explain in more detail in Section 5.

4.3.1 Translating SemQL to SQL

Transforming SemQL to SQL is done by traversing the SemQL tree from its root to leaf nodes (see Figure 3 for an example AST). Most actions of the SemQL grammar can be transformed directly to their SQL equivalent. An *Order* action, for example, is transformed simply to an `ORDER BY` SQL clause. The more abstract *Superlative* action is transformed into `ORDER BY Col1 DESC LIMIT n`. The *Filter* action is slightly more complicated. If there is no aggregation involved, the filter is transformed into a `WHERE` clause (nested if necessary). If the filter is applied to an aggregated column, it is transformed into a `HAVING` clause. As a rule of thumb, an aggregation causes all other columns in the *Select* to be included in a `GROUP BY` clause.

4.3.2 Relationships

Note that 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. To do so, *IRNet* suggested transforming 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 done that way 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.

5. VALUENET

In this section we will elaborate on how to handle questions with values using a neural network architecture. In particular, we will introduce *ValueNet light* and *ValueNet* – two novel text-to-SQL systems which incorporate values. 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* take place during pre-processing and encoding (see Figure 6).

5.1 ValueNet light

Let us assume the true values in the database in the first sample sentence of Figure 5 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 *JFK* (the value option from the oracle) and *John F Kennedy International Airport* (the tokens in the question) than *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 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 10 (blue value encodings). 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 datatype 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 %.

5.2 ValueNet

In contrast to *ValueNet light* we propose with *ValueNet* an end-to-end text-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. item *Neural Architecture:* Continuing similar to *ValueNet light* based on the architecture described in Section 4.

5.2.1 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[32] architecture leveraging the popular Transformers[37] 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 data sets, 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[18] *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 containing 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.

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

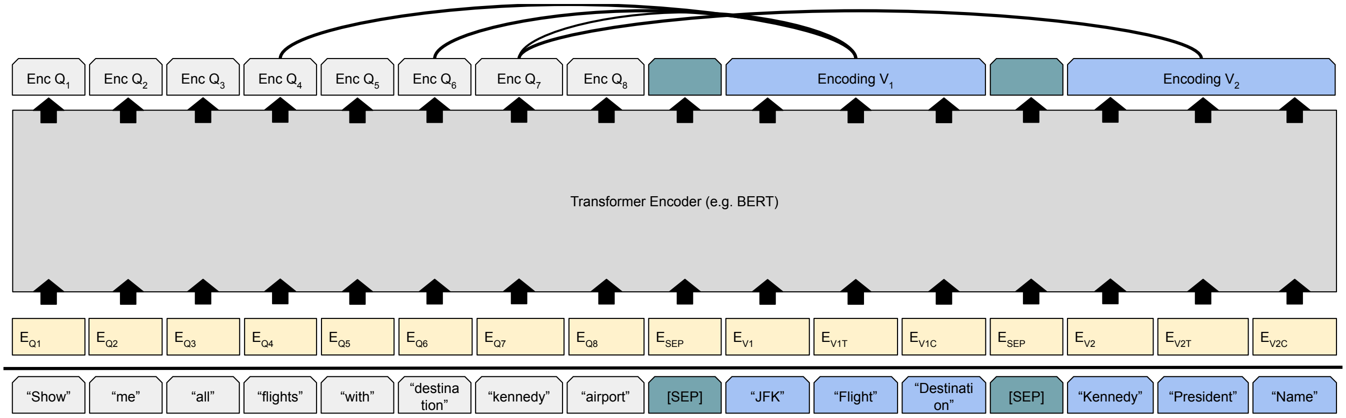


Figure 10: Encoding of value candidates and question. The black connectors on top visualize the attention built up during encoding between question - and value tokens. The large blue value encodings at the output of the encoder visualize the summarizing of a value together with its location (in a certain table column) by an LSTM. The encodings of columns and tables have been omitted for better readability.

5.2.2 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[34] or distances based on word embeddings[23]. 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 DamerauLevenshtein[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 more than one token. For example, a value like *'Kennedy International Airport'* generates one trigram, two bigrams, and three single words as value candidates.

5.2.3 Value Candidate Validation

Depending on the similarity thresholds, the number of values extracted from the natural language question, and the 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 6, 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.

5.2.4 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 10. 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[28] segmentation algorithm. The input for the encoder is then a list of pre-trained embeddings, one for each word piece.

5.2.5 Neural Architecture

After encoding the value candidates we continue similar to *ValueNet light* in Section 5.1. We use the neural architecture explained in Section 4.2.

6. 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 text-to-SQL systems: (1) How well do our approaches *ValueNet* and *ValueNet light* perform on the text-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 a simpler task using the *Exact Matching Accuracy*, which does not incorporate values.

6.1 Data Set

For our experiments we use the Spider[42] data set which contains more than 10,000 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.

We further analyzed the value distribution in the Spider data set. We focused on the *train* split, which contains exactly 7,000 of 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 11 for a distribution of the values.

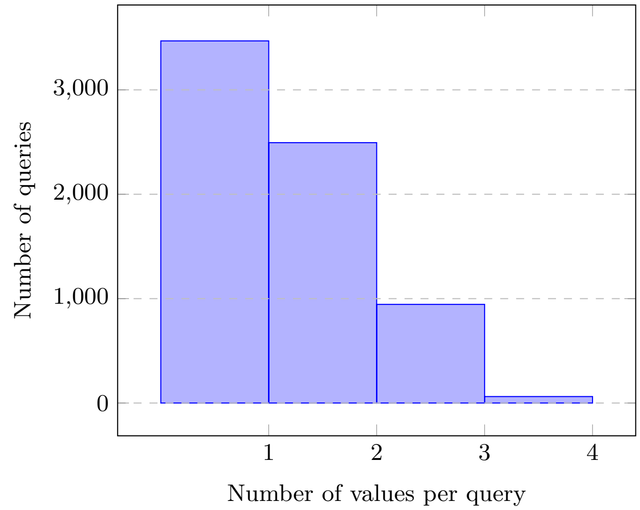


Figure 11: Value distribution in the Spider data set. 3,469 samples contain no values. 2,494 samples contain one value, 945 two values, 62 three values and 30 samples contain 4 values.

6.1.1 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`.

6.1.2 Value Types

We find the Spider dataset to contain an wide range of different values. The data set 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

We consider the variability despite some missing value types (e.g. phone numbers) to be realistic to a real world environment.

6.2 Evaluation Metrics

The Spider challenge[4] comes with two different evaluation metrics: *Exact Matching Accuracy* and *Execution Accuracy*. As we briefly explain in Section 1, *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 text-to-SQL system which is evaluated via *Execution Accuracy*.

6.3 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.

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 4.2) we use the popular Transformer[37] library. For validation we use the official Spider validation script[5].

Implementation: In our implementation we provide a transformer encoder which can be configured to use any modern pre-trained transformer model like RoBERTa[22] or XLNet[39]. 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 4.2.1) with a dimensionality of 300. We use the same dimensionality for the decoder-network described in Section 4.2.2.

Moreover, we use an Adam[20] 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[6].

6.4 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 we introduce, to the best of our knowledge, the first text-to-SQL system that synthesizes a full query including values on the Spider data set, there are currently no direct competitors using the *Execution Accuracy* metric (see the empty Spider leaderboard[7] for the latest results). We therefore compare our approach to the open source implementation of *IRNet + Bert*[16] which was the only top 10 contribution to release the code while this paper was being written. We further add the performance of *RATSQL v2 + BERT* as a single value, which was, at the time of writing, the top contribution. Note that both, IRNet and RATSQL v2 + BERT use the less challenging *Exact Matching Accuracy* metric as we described in Section 6.2. In other words, both of our approaches solve a harder problem than our competitors.

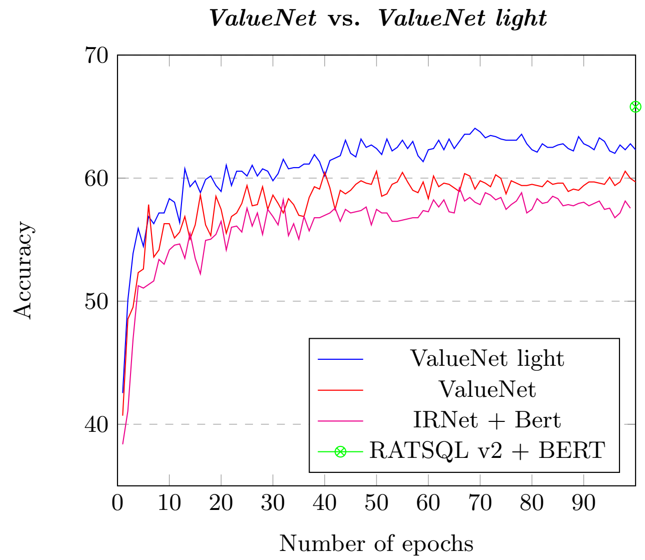


Figure 12: The performance of *ValueNet light* and *ValueNet* on the Spider data set using the *Execution Accuracy* evaluation metric. To compare with existing implementations, we also visualize the *IRNET + BERT* implementation using the *Exact Set Matching* evaluation metric. We further show the performance of *RATSQL v2 + BERT*, the current leader of that evaluation metric by a single green dot. We only visualize accuracy scores in range of 35 to 70% to emphasize the difference. The reported numbers are an average of five runs.

As we see in Figure 12 both *ValueNet* and *ValueNet light* outperform the *IRNet + Bert* implementation, even though we evaluate our system using a more difficult evaluation metric.

6.4.1 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 5. 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 dev split of the dataset.

Referring to the value difficulty of Section 6.1.1 we found that almost all of the remaining 10% not found values belong to the difficulty classes *Hard* and *Extra Hard*. For instance, we failed in the question *"What are the full names of all*

[4]https://yale-lily.github.io/spider
[5]https://github.com/taoyds/spider
[6]https://github.com/brunnurs/valuenet
[7]https://yale-lily.github.io/spider

left handed players?" to extract the value *'L'* which would match the table/column *players.hand*.

(2) *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 11, 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.

6.4.2 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.

(I) 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 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 [35], might help to prevent such errors.

(II) 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`).

(III) 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.

(IIII) 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.

6.4.3 Interpretation of the results

A first interesting insight is the relatively small difference between *ValueNet light*, *ValueNet* and the traditional approaches *IRNet* and *RAT-SQL*. As we see in Figure 12 all accuracies lie in an interval of 6%. This is therefore surprising since we know that the share of samples containing values is around 50%, so in half of the cases *ValueNet light*/ *ValueNet* have to solve a quite different task from e.g. *IRNet*. We therefore conclude that incorporating values is possible without reducing the prediction power of the model.

A second insight is the relatively small difference (4%) between *ValueNet light* and *ValueNet*. The conclusion here is that if we manage to come up with the correct value candidates (even if we propose some value candidates which are incorrect) the neural model is capable of selecting the correct one with a high probability.

7. CONCLUSIONS & FUTURE WORK

In this work we propose *ValueNet light* and *ValueNet* – two end-to-end text-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. Despite using a more difficult evaluation metric on the Spider challenge than most recent works, we achieve state-of-the-art results with our systems.

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 using the available data from the database.

8. ACKNOWLEDGMENTS

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

9. 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):171176, 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. 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] M. Honnibal and I. Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

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

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

[21] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. 09 2019.

[22] 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.

[23] 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.

[24] 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 141es, USA, 2004. Association for Computational Linguistics.

[25] 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 149157, New York, NY, USA, 2003. Association for Computing Machinery.

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

[27] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

[28] M. Schuster and K. Nakajima. Japanese and korean voice search. pages 5149–5152, 03 2012.

[29] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism.

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

[31] 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.

[32] 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.

[33] 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.

[34] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168173, Jan. 1974.

[35] 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.

[36] D. H. D. Warren and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *Comput. Linguist.*, 8(34):110122, July 1982.

[37] 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.

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

[39] 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.

[40] 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.

[41] P. Yin and G. Neubig. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *CoRR*, abs/1810.02720, 2018.

[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.