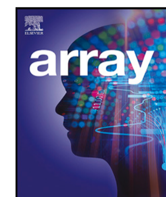
Array 24 (2024) 100368

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/locate/array



Combining computational linguistics with sentence embedding to create a zero-shot NLIDB

Yuriy Perezhohin [a,b], Fernando Peres [a,b], Mauro Castelli [b,*]

[a] MyNorth AI Research, Alameda Bonifácio Lázaro Lozano nº15- 1ºC, 2780-125, Oeiras, Lisboa, Portugal

[b] NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Campus de Campolide, 1070-312, Lisboa, Portugal

ARTICLE INFO

Dataset link: https://yale-lily.github.io/spider, https://github.com/yuriyvnv/Neural-Rule-Based-model-for-Text-to-SQL

Keywords:
Text to SQL
Natural language processing
Computational linguistics
Sentence embeddings

ABSTRACT

Accessing relational databases using natural language is a challenging task, with existing methods often suffering from poor domain generalization and high computational costs. In this study, we propose a novel approach that eliminates the training phase while offering high adaptability across domains. Our method combines structured linguistic rules, a curated vocabulary, and pre-trained embedding models to accurately translate natural language queries into SQL. Experimental results on the SPIDER benchmark demonstrate the effectiveness of our approach, with execution accuracy rates of 72.03% on the training set and 70.83% on the development set, while maintaining domain flexibility. Furthermore, the proposed system outperformed two extensively trained models by up to 28.33% on the development set, demonstrating its efficiency. This research presents a significant advancement in zero-shot Natural Language Interfaces for Databases (NLIDBs), providing a resource-efficient alternative for generating accurate SQL queries from plain language inputs.

1. Introduction

Natural Language Processing (NLP) has been extensively studied since the end of World War II. NLP focuses on facilitating interaction between humans and computers by enabling the understanding and generation of human language. A sub-field of NLP, Natural Language Interfaces (NLI), deals with both unstructured and structured data [1]. NLI for structured data began to gain significant attention in the 1990s when Relational Database Management Systems (RDBMS) became widespread and were adopted by many businesses globally. To extract information from these databases, end users need to be familiar with SQL commands such as SELECT, FILTER, and HAVING clauses, which posed challenges for non-technical users. Over the past two decades, Natural Language Interfaces for Databases (NLIDBs) have become a hot research topic [2,3], with numerous text-to-SQL models being developed to meet the growing demand for accessible data retrieval. As artificial intelligence (AI) technology continues to evolve rapidly, NLIDBs are being adapted for use in various domains to address business needs. However, despite the progress in this area, there are still significant opportunities and challenges in enhancing NLIDBs to meet user expectations, and address issues related to language context and understanding. A competent NLIDB system for an inexperienced user must be capable of interacting with data and dealing with several challenges, such as answering questions with minimal information or adapting to new domain terminology. Consequently, current systems still face several limitations. The most significant problem is the unexplicit requests: the ability to process and understand decreases significantly when the user's question lacks complete context [4]. Current NLIDB systems are trained on benchmarks that do not tackle natural language's ambiguity, leading these models to primarily understand only clear and specific questions. Another major challenge faced by most NLIDBs is poor cross-domain adaptability. Despite advances in deep learning techniques and the availability of cross-domain benchmarks for model training, these systems remain too shallow for industrial use [5], resulting in poor generalizability [6]. Another important aspect is the accessibility of deep-learning models. Nowadays, most text-to-SQL models are trained using deep learning techniques that operate as "black boxes", making it difficult for users to modify components when necessary [7,8].

To address these challenges, in this work, we have developed a method to create a text-to-SQL model that does not require training. Our central hypothesis is that by combining rule-based techniques with a handwritten vocabulary and pre-trained models fine-tuned for semantic search, we can correctly translate a natural language question into SQL code. The core idea is to create a zero-shot NLIDB that can generalize to new domains without any prior training, relying solely on linguistic and handwritten rules. Currently, the only zero-shot models capable of tackling the complex task of text-to-SQL, and available on academic benchmarks, are provided by technological companies. These models, such as GPT-3 [9], employ prompting techniques to input

* Corresponding author.
E-mail addresses: yperezhohin@novaims.unl.pt (Y. Perezhohin), fperes@novaims.unl.pt (F. Peres), mcastelli@novaims.unl.pt (M. Castelli).

https://doi.org/10.1016/j.array.2024.100368
Received 9 February 2024; Received in revised form 9 September 2024; Accepted 9 October 2024
Available online 24 October 2024
2590-0056/© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

all the database information and requests to translate the text into SQL code [10]. However, these models require significant time and resources to train.

Considering the existing scenario and the challenges identified by several authors [1,4,11], this work aims to address the following research questions:

- **RQ1** - Can a combination of linguistic rules and pre-trained models for semantic search accurately identify the necessary columns and tables to answer specific requests?
- **RQ2** - Is it possible to translate plain language requests into SQL code across different database domains without training the text-to-SQL model?

The main contribution of this work is the development of a novel, resource-efficient, and domain-flexible approach for NLIDBs. This approach has a significant potential for further improvement and advancement within this line of research. The rapid pace of AI advancements driven by the private sector, with its vast resource investments, has made it challenging for academic researchers to compete globally. However, the absence of extensive training requirements in our approach allows researchers to explore innovative ideas and techniques, potentially leading to breakthroughs in solving the text-to-SQL problem.

The rest of the paper is organized as follows: Section 2 presents existing work in the NLIDB field and highlights some problems discussed by distinct authors. The Section 3 overviews the methodology to build and evaluate the proposed NLIDB. The evaluation of the model, given different pre-trained models for sentence embeddings, is presented in the Section 4. Finally, the Section 5 discusses the main contributions of the proposed model and establishes a direction for future work.

2. Related work

NLI are systems designed to facilitate human–machine interaction through natural language. NLIDBs have a more specific focus: allowing users to interact with databases. Users can provide input in the form of voice commands or textual questions, and the system responds with actions, data frames, text, or numerical outputs.

Their use emerged in the late 1960s with the development of systems like LUNAR, SHRDLU, or MARGIE [12]. These systems were designed for narrow and specific domains, enabling them to produce accurate outputs. Nowadays, several intelligent systems, such as ALEXA, SIRI, or CORTANA, can handle questions from different domains. All of those interfaces address the challenge of transforming any user request into a machine-understandable language and, subsequently, executing the corresponding action. Thus, the primary advantage of these interfaces is that they allow users with little or no technical expertise to perform specific tasks. However, a significant limitation is that these systems cannot retrieve information or execute tasks from unrecognizable commands. For example, in an interface that accesses information about patients, a user might ask, "*Why the patient has died?*". The meaning of this question can be simple, but naturally, the interface was not designed to answer such queries.

RDBMS [13] are system used to maintain relational databases, a technology still widely in use today. The standard users for this type of system must have appropriate knowledge in SQL [14], as it is the core language for interacting with the database. Due to the shortage of a knowledgeable audience, the development of NLIDBs was crucial to tackling the problem of transforming natural language into SQL.

2.1. Classical approaches

The task of transforming natural language requests into SQL is a long research line of NLIDB, beginning with the LUNAR system. Originally developed for geologists, LUNAR allowed access to chemical analyses of lunar rocks and soil collected during the Apollo moon missions [15]. This system employed a parsing-based method to achieve its goals. Since then, numerous other approaches have been developed for NLIDB, including:

- **Keyword-Based** - The fundamental idea of this method is to find keywords in the users' questions and match them with the database and its metadata. A clear example of this technique is the Search Over Data Warehouse [16] (SODA) system, which matches keywords with the database, produces a graph from metadata with all the nodes found, and, finally, assigns a score to each node to find the appropriate table/column of the database that can be used to answer the user's request. This method is also employed in other systems such as NLP-Reduce [17] and QUEST [18]. Each system has a different approach to the problem, but the fundamental idea is to match tables, columns, or operations with keywords. The keyword-based approach is flexible and simple, making it easy to use. On the other hand, its main disadvantage is that it cannot generate more complex queries, such as those involving aggregations.
- **Pattern-based** - In this method, the system compares human-created patterns and rules to an input sentence and verifies if the pattern matches. If a match is found, it applies a predefined rule to formulate the appropriate query clause. For example, this method can use word patterns like "*by*" or "*per*" to indicate that an aggregated output is required. Thus, if the system matches these keywords in the user's request, it adds a *Group By* clause. These patterns are not limited to keywords; they can also be sequences of part-of-speech tags or other syntactic representations. An illustrative implementation of this method is the system Savvy [19].
- **Parsing-based** - This method is divided into two approaches: syntactic and semantic-grammar. The syntactic method tries to generate a syntactic structure of the user's input tokenizing it and analyzing the grammatical rules. This process creates a parsing tree where some nodes are mapped into their semantic labels and then to SQL components. By analyzing this structure, it is possible to visualize how column names can relate to aggregation words such as "*by*" or "*per*". This method is used to implement the NALIR [20] system, where the tree nodes are mapped into SQL component rules to match inputted questions. The semantic-grammar parsing method is similar to the syntactic one, but instead of having a full tree, it eliminates or groups together insignificant nodes, reducing the tree's complexity. In this approach, nodes are classified based on function or meaning rather than syntactic categories. It is also possible to assign a specific name to each node to reduce the ambiguity of inputted questions, but this requires prior knowledge of the elements in the domain. A system built using this method is LADDER [21].
- **Grammar-based** - This method relies on a set of predefined rules to define the questions the system can answer. As the user types a question, the system suggests the next words to form a valid input question, thus reducing ambiguity. A relevant disadvantage of these systems is their strong dependency on the domain-specific rules. An example of this method is TR Discover [22], which uses the auto-suggestion mode for the user input, so when the user writes "*p*", it will complete the word by selecting "*profession*" and then try to predict the next word from the known grammar.

All of the methods mentioned have been widely used over the last five decades, and Table 1 displays several NLIDBs for SQL distinguished by the method used. Nowadays, the problem of text to SQL has earned new and more innovative approaches, including the use of deep learning models to generate SQL queries from natural language questions [23].

Table 1
Classical NLIDBs distinguished by the method used to tackle the problem of text to SQL.

Method	Model
Keyword	SODA [16] NLP-Reduce [17] Précis [24] Keyword++ [25] QUEST [18]
Pattern	Savvy [19]
Parsing	Lunar [15] LADDER [21] WASP [26] USI Answers [27] NALIR [20] ANTHENA [28] SQLizer [29] BioSmart [30]
Grammar	TR Discover [22]

2.2. Deep learning approaches

Deep Learning approaches have been used in various areas, such as Computer Vision and NLP, yet they require a large amount of data for the training phase. Recent advancements in this field were prompted by the availability of complex datasets like Spider [31], BIRD [32], and WikiSQL [33], and the task of text-to-SQL has reached state-of-the-art performance. In particular, several deep learning approaches were fundamental for the developments in the text-to-SQL task:

- **Sequence to Sequence** - One of the first deep learning-based approaches to appear was "End-to-End" or seq2seq [34], where the model is trained on input–output pairs of statements in the task of machine translation. This approach utilizes a multi-layered LSTM [35], encoding the input sequence into a fixed dimensions vector and then decoding it into target sentences using the same technique. Following this idea, a model that applied the seq2seq approach to the text-to-SQL task was developed [33]. It proposed an Encoder–Decoder system with an attention layer and a reward mechanism in the query generation process. However, models trained with the seq2seq approach face a significant challenge related to the order of elements: the models are sensitive to the specific order in which SQL clauses are presented during training. For example, the training question "*What are the salaries where jobs are Data Scientist and Research Scientist?*" would formulate the correct SQL clause. However, if the question's order is changed to "*Where jobs are Data Scientist and Research Scientist, what are the salaries?*" the model struggles because it has never seen the second question, although it has the same meaning. Another notable method is the integration of database schema information into the training phase. The model X-SQL [36] included the encoded schema information with a sequence encoder to improve the decoding step with contextual information from the schema.
- **Sequence to Set** - To address the order problem, a new method was developed where, instead of directly predicting the target SQL query, the question is divided into parts. Each part is then assigned to a respective clause (i.e., Select, Where, Aggregation clauses). SQLNet [37] was proposed as a sequence-to-set (seq2set) approach. This model employs a sketch-based and slot-filling architecture, aligning slots naturally to the syntactical structure of the SQL query. The neural network is then trained using an attention mechanism to predict each slot. With the development of the SPIDER dataset, increasing attention was given to the database schema during the evaluation phase, particularly when dealing with complex and unseen databases. A Graph Neural Network [38] (GNN) was developed to encode the database schema to later feed this representation into an Encoder-Decoder architecture [39]. However, this method may encounter a context-ignoring issue, as the model cannot reason over database metadata and may pick irrelevant or wrong columns when the user's intent is not clear. The proposed solution [40] implements a graph convolution network [41] to predict the relevance of a column from the global context of the question and the database.

Table 2
Deep learning models employed to build NLIDBs, trained on the WikiSQL and Spider datasets. The development and test sets accuracies are represented as exact set matches without values.

	Model	Architecture	Dev	Test
WikiSQL	Seq2SQL [33]	Seq2Seq	60.8	59.4
	PT-MAML [47]	Seq2Seq	68.3	68.0
	SQLNet [37]	Seq2Seq	69.8	68
	TypeSQL [48]	Seq2Set	74.5	73.5
	X-SQL [36]	Seq2Seq	89.5	88.7
	HydraNet [49]	Seq2Set	89.1	89.2
	SeaD [50]	Seq2Seq	90.2	90.1
	SeaD+Execution Guiding decoding [50]	Seq2Seq	92.9	93.0
Spider	GrammarSQL [51]	Seq2Set	34.8	33.8
	GNN [39]	Seq2Set	40.7	39.4
	IRNet [52]	Seq2Set	53.2	46.7
	RATSQL [53]	Seq2Seq	62.7	57.2
	Photon [54]	Seq2Seq	63.2	–
	ValueNet [55]	Seq2Set	–	62.0
	RASAT+PICARD [42]	Seq2Seq	75.3	70.9
	T5-3B+PICARD [56]	Seq2Seq	75.5	71.9
	LGESQL+ELECTRA [57]	Seq2Seq	75.1	72.0
	REDSQL-3B+NatSQL [58]	Seq2Set	80.5	72.0
	S^2SQL + ELECTRA [59]	Seq2Seq	76.4	72.1
	N-best List Rerankers+PICARD [60]	Seq2Set	76.4	72.2
	SHIP+PICARD [61]	Seq2Seq	77.2	73.1
	Graphix-3B+PICARD [62]	Seq2Seq	77.1	74.0

The methods mentioned above have been vastly explored. Even though they present certain challenges, researchers have tried to overcome them by incorporating more complex structures into existing models. For example, the RASAT model [42], which integrates a seq2seq architecture based on the T5 model [43], replaces the self-attention layers in the encoding phase with relation-aware layers, resulting in two additional relation embedding lookup tables. Consequently, it converts the input sentences into an interaction graph by adding the relations through relational propagation. Once the model is trained, it retrieves relation embeddings from the lookup tables via the interaction graph. This results in a model with almost all word relations, schema linking system, and syntactic dependency embedded into one relation representation. Many other models were proposed, including those that allow user feedback to verify and refine generated queries [44]. In this case, the model generates an answer, and if it is incorrect, the user can provide additional input to reformulate the query. Table 2 displays most of the models used in the text-to-SQL task. Despite the recent progress in the text-to-SQL models and the increase in accuracy achieved in several benchmarks, significant limitations remain, such as domain adaptation and training costs [45,46].

2.3. Embeddings

Human language is often ambiguous or vague, leading to the development of several neural models designed to extract the meaning from user input, which has also driven the evolution of NLIDBs. The models are trained with different techniques to produce embedded vectors in an N-dimensional space, allowing them to learn features from words or sentences. Many approaches create word embeddings from text corpora, and it is crucial to refer to them as they are a core component in this work. Embeddings allow for calculating the

similarity between pairs of words or sentences by considering their context and meaning. For instance, the words "*jobs*" and "*professions*" are more similar to each other than to two other words, "*year*" or "*date*", and this similarity is used to map the SQL clauses correctly. The Word2Vec [63] model represented a significant breakthrough in word representations through unsupervised learning. Its architecture allows it to use either a Skip-Gram or Common Bag of Words (CBOW) model based on a feed-forward neural network. The main goal of Word2Vec is to learn the semantic meaning between words by analyzing their context, a significant advantage over traditional models such as Bag of Words [64] or Term Frequency Inverse Document Frequency [65].

The creation of word embeddings has its limitations, such as the challenge of out-of-vocabulary words and the restricted semantic context, which is limited to the available training datasets. Pre-trained Language Models have earned popularity because of their ability to transfer existing knowledge to specific tasks, achieving state-of-the-art performance in many cases. BERT [66] is one of the most well-known pre-trained language models, trained using the concept of self-supervising learning. Other models, such as the Robustly Optimized BERT Pretraining Approach (RoBERTa) [67] and A Little BERT (ALBERT) [68], were developed to improve the original BERT model. While RoBERTa adds more training data and parameters, ALBERT reduces the parameters and introduces a new architecture.

In this work, we create linguistic rules that use lexical features and relationships between words to identify specific patterns in user requests. These patterns and the metadata from the database, as table and column names, are embedded into N-dimensional vectors using pre-trained models. A ranking algorithm then compares the similarity between these embeddings and identifies the column or table with the highest similarity for each pattern. We also create a set of structuring rules and a vocabulary to assemble the query. To prove the resource effectiveness and adaptability of our model, we conducted an experimental phase considering several databases (from different domains) within the SPIDER benchmark.

3. Materials and methods

This section presents the methods and materials used to develop and evaluate the proposed NLIDB system. The methodology comprises three main phases. The first phase involves preparing the NLIDB by performing pre-defined activities to set up the environment for text-to-SQL tasks. In the second phase, natural language questions are translated into SQL queries. The third phase executes the generated SQL query. Fig. 1 illustrates the whole methodology.

3.1. Data and evaluation

The SPIDER [31] dataset is a complex benchmark used to evaluate the performance of the most prominent semantic parsers in the text-to-SQL task. It consists of 200 databases, each with multiple tables, and includes 10.181 questions corresponding to 5639 complex and unique SQL queries. We rely on the SPIDER benchmark because it covers various domains and contains a wide range of query types that cover the majority of SQL instructions (e.g., "GROUP BY", "ORDER BY", "SUB-SELECT", "HAVING"). It is important to highlight that many databases from this benchmark do not follow the general RDBMS naming conventions [69]; for example, the "twitter" database includes columns named "f1" and "f2". For this reason, instead of focusing on the whole benchmark, specific databases (in which the primary and foreign keys are clearly identified and all columns are meaningfully described) have been selected. In our experimental phase, we excluded questions that require "EXCEPT", "INTERSECT", and "UNION" SQL clauses, because the main goal is to prove the generalization capabilities of the algorithm and its effectiveness without a training phase.

The pre-selected databases from the training and development (i.e., test) sets of the benchmark contain 429 and 120 questions, respectively, spanning over 19 domains. The questions in the training set were employed to refine the linguistic rules. On the other hand, the databases in the development set were used to assess the algorithm's effectiveness in generating correct and working SQL queries on unseen databases. The two metrics used to evaluate the text-to-SQL algorithm are execution accuracy and the exact set match accuracy. Execution accuracy measures how well the algorithm performed by comparing the outputs of the target and the predicted SQL queries, while exact set match accuracy compares the structure of the target and predicted SQL queries without considering their outputs.

We use the SQLite database engine to execute the generated queries because all the databases in the benchmark are in SQLite format.

3.2. Proposed NLIDB

In this section, we will present a detailed explanation of the proposed NLIDB system, covering the methods used to embed database metadata, the translation process from natural language to SQL through linguistic and syntactic approaches, and the structuring rules applied to generate precise SQL queries based on user inputs.

Formally, the proposed NLIDB system can be defined as $\mathcal{N}(\mathcal{M}, \Sigma, \mathcal{Q}, \mathcal{D})$, where:

- $\mathcal{N}$ represents the NLIDB algorithm itself.
- $\mathcal{M}$ is the embedding model used to represent the database metadata and natural language queries in a vector space.
- Σ is the configuration set, $\Sigma = \{\mathcal{A}, \mu, \mathcal{V}\}$, where:
  - $\mathcal{A}$ represents various models, different from the embedding model, used for generating dependencies and Part-of-Speech (POS) tags.
  - μ represents the similarity measure (i.e., cosine similarity or dot product).
  - $\mathcal{V}$ is a set of predefined keywords and synonyms corresponding to SQL functions.
- $\mathcal{Q}$ is the natural language question to be translated into an SQL query.
- $\mathcal{D}$ is the database containing the tables and columns.

3.2.1. NLIDB preparation

To translate a natural language statement to SQL code, it is essential to understand its meaning. In particular, it is necessary to address potential issues like ambiguities. Therefore, this phase involves several tasks aimed at reducing ambiguities and identifying relevant elements within the database.

This preparation phase is denoted as a function $\Phi(\mathcal{D}, \mathcal{M}) \rightarrow \mathcal{E}$ where:

- $\mathcal{D}$ represents the database.
- $\mathcal{M}$ denotes the model used for embedding.
- $\mathcal{E}$ represents the set of embeddings for tables and column names, denoted as, $\{T_1[C_1, C_2, \dots, C_n], T_2[C_1, C_2, \dots, C_k], \dots\}$, where T_i is a table and C_i is a column.

To extract the semantic meaning of column and table names in each database $\mathcal{D}$ used in this experiment, we embed them into N-dimensional vectors. For this task, we chose four pre-trained models, as detailed in Table 3, which presents their performance on the semantic search task. These models, publicly available on the *Hugging Face* platform,[1] use two similarity metrics: the dot product [70] and cosine [71] similarity. We then compute the similarity between the patterns extracted using linguistic rules and the resulting embeddings.

[1] https://huggingface.co/models.

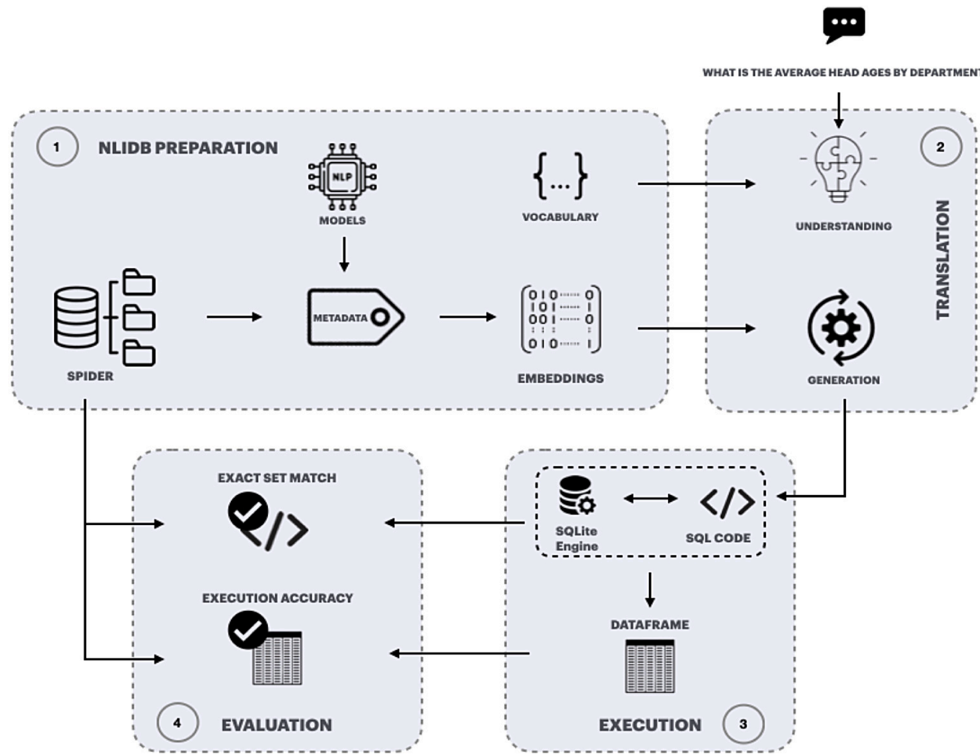


Fig. 1. Overview of the methodology.

Table 3
Performance of fine-tuned models for semantic search. Values taken from https://www.sbert.net/docs/pretrained_models.html.

Fine tuned model	Base model	Performance	Embedding dimension
multi-qa-mpnet-base-dot-v1	*MPNet* [72]	57, 60	768
all-mpnet-base-v2	*MPNet* [72]	57, 02	768
multi-qa-distilbert-cos-v1	*DistilBERT* [73]	52, 83	768
multi-qa-MiniLM-L6-cos-v1	*MiniLM* [74]	51, 83	384

Many SQL functions are represented by single keywords: for example, “mean” corresponds to the AVG() function, and words like “where” or “whose” are often interpreted as a WHERE clause in most questions. Given this scenario, we have created an extensive vocabulary $\mathcal{V}$, with synonyms and keywords that maps all possible SQL functions appearing in the questions under analysis. Both the metadata embeddings ($\mathcal{E}$) and the vocabulary ($\mathcal{V}$) are provided as input to the Translation step, where the algorithm generates the SQL query from the natural language question.

3.2.2. Text to SQL translation

The first step consists of transforming a given natural language question $\mathcal{Q}$ into a more manageable form, simplifying the translation process. This step is formally defined as $\Psi(\mathcal{Q}, \mathcal{A}) \rightarrow \mathcal{T}$, where:

- $\mathcal{Q}$ is the natural language question.
- $\mathcal{A}$ is the model used for *POS* tags and Dependencies.
- $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ is the set of tokens extracted from $\mathcal{Q}$ with respective *POS* tags.

The question ($\mathcal{Q}$) is processed using the Stanford Core NLP [75] pipeline ($\mathcal{A}$), which extracts the Part-of-Speech (POS) tags and dependencies between words. The function Ψ yields a set of tokens from the original question, each annotated with its corresponding POS tags and dependencies.

When researching possible ways of translating SQL queries from natural language, we observed a recurring pattern. Specifically, column names in most tables were either a *Noun* or a *Proper Noun*. The DBTagger [76] research also pointed to this finding, referring to the fact that *Noun* words are matched to the table or column names. Based on this observation, the next step employs the function $\Gamma(\mathcal{T}) \rightarrow \mathcal{D}_L$, which extracts dependencies from $\mathcal{T}$ between words tagged as *Noun* or *Proper Noun* during the POS tagging process and outputs a list of dependencies ($\mathcal{D}_L$). $\mathcal{D}_L$ is then forwarded to the lexical pattern matcher, denoted as $\Lambda(\mathcal{T}, \mathcal{D}_L) \rightarrow \mathcal{F}_L$, where:

- $\mathcal{T}$ is the set of tokens extracted previously
- $\mathcal{D}_L$ is the list of dependencies.
- $\mathcal{F}_L$ is the list of patterns.

The lexical pattern matcher (Λ) is responsible for combining tokens from $\mathcal{T}$ that match a set of predefined linguistic rules, using the dependencies ($\mathcal{D}_L$). For example, the sequence “head names” in the question “What are the head names?” corresponds to a two-word pattern where the word “names” is a noun and also a compound of the word “head”. The compound dependency indicates that one word is syntactically dependent on the other, which could also be a verb or a number. In this case, those two words are coupled together as they can represent the semantic meaning of a column in the database.

To retrieve the table and column name with the highest similarity scores, two functions are used: the embedding and extraction functions. The first is denoted as: $\Xi(\mathcal{M}, \mathcal{F}_L) \rightarrow \mathcal{E}_{PM}$, where:

- $\mathcal{M}$ is the model used for embeddings.
- $\mathcal{F}_L$ is the list of patterns.
- $\mathcal{E}_{PM}$ is the set of embeddings for the patterns.

All patterns ($\mathcal{F}_L$) obtained from the lexical matcher (Λ) are embedded using the chosen models $\mathcal{M}$ (see Table 3), resulting in $\mathcal{E}_{PM}$. The extraction function is defined as $\Upsilon(\mathcal{E}_{PM}, \mathcal{E}, \mu) \rightarrow \mathcal{LTC}$, where:

- $\mathcal{E}_{PM}$ is the embedding set from the pattern matcher.

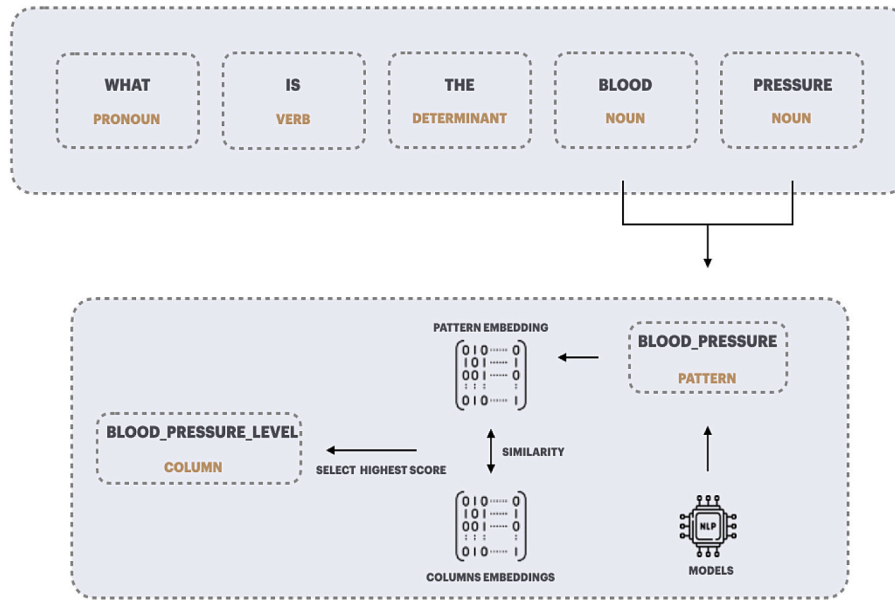


Fig. 2. Illustration of the extraction of two *NOUN* words, which are joined and embedded to compare the similarity between the database metadata embeddings. Lastly, the column with the highest score is selected.

- $\mathcal{E}$ is the embedding set from database metadata.
- μ is the similarity metric.
- $\mathcal{LTC}$ is the list of tables and column names identified.

The similarity scores between $\mathcal{E}_{PM}$ and $\mathcal{E}$ are calculated using μ. The table or column name with the highest similarity score is then retrieved and stored in $\mathcal{LTC}$ for use in structuring the SQL query. Fig. 2 displays the entire process.

The following step uses the function $\Omega(\mathcal{T}, \mathcal{V}) \rightarrow \mathcal{TS}$ where:

- $\mathcal{T}$ is the set of extracted tokens.
- $\mathcal{V}$ is the predefined Vocabulary.
- $\mathcal{TS}$ is the set of tokens corresponding to SQL clauses.

The vocabulary $\mathcal{V}$ determines whether a specific keyword or synonym in $\mathcal{T}$ corresponds to any SQL functions. If a match exists, the corresponding SQL function is applied to modify the original token, generating a new set of tokens ($\mathcal{TS}$) with SQL Clauses.

To aid in constructing the final query, the resulting sets $\mathcal{LTC}$ and $\mathcal{TS}$ are combined, using the function $\Pi(\mathcal{T}, \mathcal{LTC}, \mathcal{TS}) \rightarrow \mathcal{LPQ}$ where:

- $\mathcal{T}$ is the set of tokens from the question.
- $\mathcal{LTC}$ is the set of tables and column names identified.
- $\mathcal{TS}$ is the set of tokens related to SQL clauses.
- $\mathcal{LPQ}$ is the pseudo query.

The function Π iterates over $\mathcal{T}$, replacing tokens with the corresponding table names ($\mathcal{T}_i$), column names ($\mathcal{C}_i$), or SQL clauses, thereby preserving the natural order of the question ($\mathcal{Q}$), which is crucial for generating the final query. In the final step, the function $\Theta(\mathcal{LPQ}) \rightarrow \mathcal{Q}_{\text{final}}$ applies a set of structuring rules to determine which columns or values should be included in each SQL clause to generate the final SQL query $\mathcal{Q}_{\text{final}}$. Fig. 3 illustrates the steps involved in the translation phase.

Several classes of structuring rules correspond to the SQL clauses that may appear in a question. For instance, the question, "What are the best-paid jobs?", refers to the "jobs" column and the "MAX" component of the "salaries" column. Thus, the algorithm will employ only the classes of structuring rules related to the "SELECT" and the "MAX" clauses to formulate the query.

The algorithm analyzes the processed question and determines the input for the "SELECT" clause, whether it is a column or a mathematical calculation (count, average, etc.). In this experiment, we implemented a rule that would break the question into two parts if it found the adposition "of" (the rest of the structuring rules are in the supplementary material). For example, in the question "What is the average salary of the Data Scientists?", the salary column is inputted into the "SELECT" clause because it appears before the adposition "of". Afterward, the algorithm searches for the remaining clauses (in this case, for the "WHERE" corresponding to "Data Scientist"). One of the structuring rules contemplates the "JOIN" clause, which is the most important as it allows the concatenation of different tables within the database. Consequently, the model identifies if the question references more than one table, retrieves each table's column identifiers (primary and foreign keys), and forwards this information to the rules responsible for assembling the query (further details are provided in the supplementary material). Fig. 4 shows the process related to the "JOIN" clause.

The following pseudo-code outlines all the necessary steps of the proposed text-to-SQL algorithm. The first step involves extracting the POS tags and truncating the tokens corresponding to predefined linguistic rules. Next, the algorithm iterates over the saved tokens and executes two functions to predict the correct table and corresponding column based on specific patterns. In the third step, the algorithm replaces predefined keywords in the vocabulary with the respective SQL functions. The last step is responsible for assembling the final SQL query, executing it, and storing the output.

4. Results and discussion

This section presents and discusses the results of the experiments conducted in this research, along with the factors that influence the performance of the proposed text-to-SQL algorithm.

4.1. Execution accuracy

Table 4 presents the execution accuracy achieved on the training and development sets by considering different pre-trained models for semantic search with cosine and dot product similarity. The results show that the two best-performing models are multi-qa-mpnet-base-dot-v1 and multi-qa-MiniLm-L6-cos-v1: the former achieved the highest accuracy on the training set, while the latter performed best on the development set.

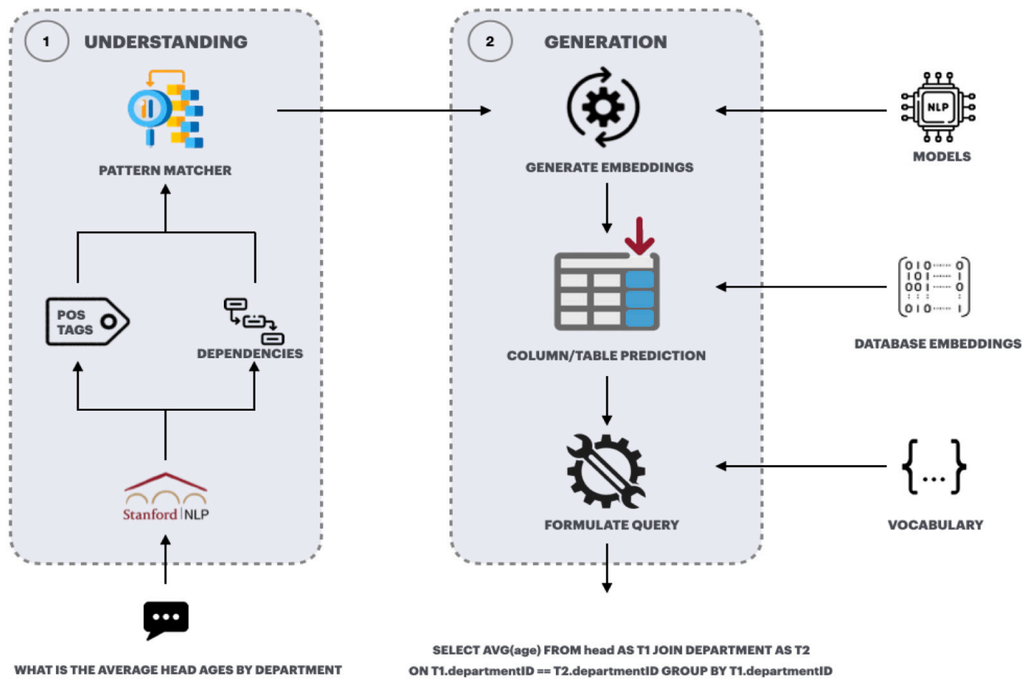


Fig. 3. Walk-trough of the understanding and the generation steps, starting from the inserted question and finishing with the generated query.

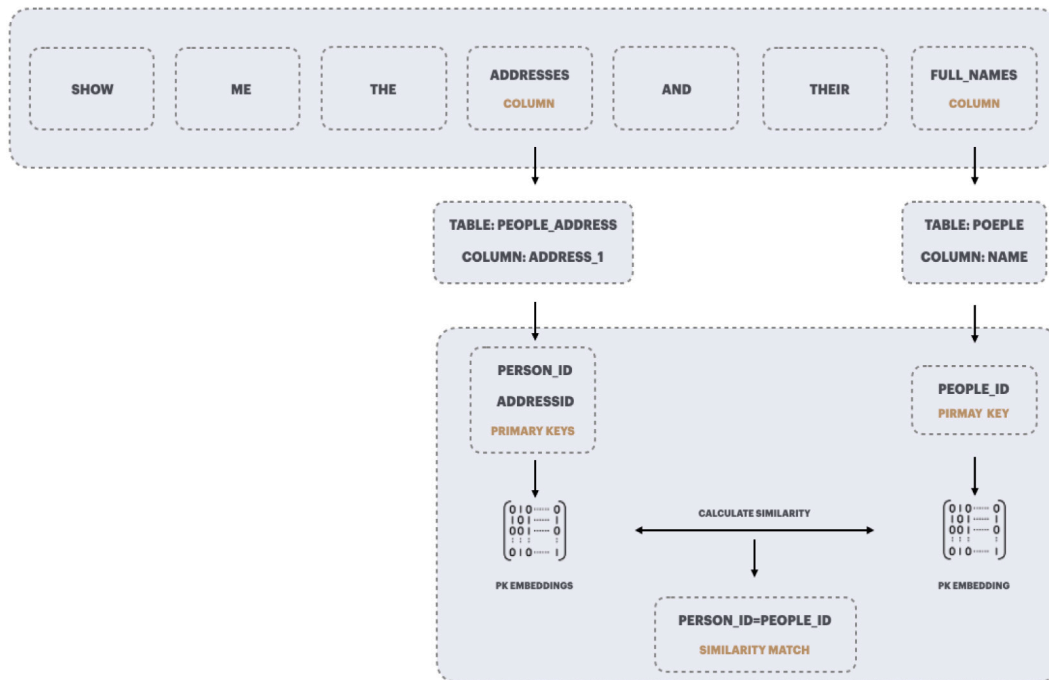


Fig. 4. Overview of the join operation.

The table shows that there are two models which performed similarly. Thus, we rely on model size to determine the best option for this task. Of the two top models, multi-qa-mpnet-base-dot-v1 is the larger, with a size of 420 megabytes and producing 768-dimensional vectors, while multi-qa-MiniLM-L6-cos-v1 is only 80 megabytes and creates 384-dimensional vectors. The larger model could be more suitable for broader scale applications where rare database domains might appear, as it has greater capacity to find related words. However, in this case, the smaller model is preferable due to its lower computational cost. The large model takes approximately 17 min and 39 s to process the questions in the training set and build the corresponding SQL queries (on average, 4.1 s for each question), while the smaller model takes 15 min and 40 s, with an average of 3.58 s per question using the cosine similarity metric. Switching to the dot product metric reduces the time required to construct the SQL queries for the training set. The large model's time decreases slightly to 17 min and 30 s, while the

Algorithm 1: Neural Rule-Based algorithm.

```
Input  : Question, Database Metainformation Embeddings,
         Vocabulary
Output: Formulated SQL query, SQL output dataframe
1  final ← [];
2  pseudoQuery ← [];
   // Step 1: Extract POS tags and truncate
      dependencies
3  for token in tokenize(question) do
4  |  pseudoQuery.append(extractPOS(token));
5  end
6  pseudoQuery ← truncateDependencies(pseudoQuery);
   // Step 2: Given the linguistic rules embed the
      patterns and extract table/columns
7  for token in pseudoQuery do
8  |  if token ∈ predefinedRules then
9  |  |  tableColumn ← extract_table(token);
10 |  |  final ← extract_column(tableColumn);
11 |  end
12 end
   // Step 3: Find SQL functions and store the
      information for assembling of the query
13 for token in pseudoQuery do
14 |  aggregationsFunction(token, pseudoQuery, final);
15 |  basicSQLfunctions(token, pseudoQuery, final);
16 |  havingFunctions(token, pseudoQuery, final);
17 end
   // Step 4: Assemble the query and execute it
18 final ← assemble_query(final);
19 result ← execute_query(final);
```

Table 4
Execution accuracy achieved by the proposed algorithm on training and development sets using different sentence embeddings and with distinct similarity metrics. Bold text denotes the best results.

Pre-trained models	Cosine similarity		Dot product similarity	
	Train	Dev	Train	Dev
multi-qa-mpnet-base-dot-v1	**72,03**	**70,0**	**72,03**	**70,0**
all-mpnet-base-v2	62,94	59,16	62,47	60,0
multi-qa-distilbert-cos-v1	68,53	64,16	68,06	65,0
multi-qa-MiniLM-L6-cos-v1	**69,46**	**70,83**	**69,46**	**70,83**

smaller model's time decreases to 13 min and 55 s.[2] Thus, based on the experimental results, we can state that the multi-qa-MiniLM-L6-cos-v1 model coupled with the dot product metric is the best choice for this specific task.

4.2. Exact set match accuracy

The structuring rules are a crucial part of the proposed algorithm, as they are essential for building the final query. The following assessment aims to understand where the query formulation failed and how it can can be improved. To perform this analysis, based on the previous considerations, we rely on the multi-qa-MiniLM-L6-cos-v1 model with the dot product similarity metric. Specifically, the study compares the components of the resulting queries (i.e., those produced by our algorithm) against the target queries from the benchmark. Table 5 presents the exact set match accuracy divided into three categories: *simple*, *join*, and *sub-select*. The *simple* category comprises queries that do not involve joins or sub-selects, while the *join* category requires at least one join operation. The *sub-select* category involves only nested queries. For each category, Table 5 shows all the query types appearing in the training and development sets.

Table 5
Exact set match of the predicted queries for training and development sets with every possible clause combination.

Query	Query type	Train		Dev	
		Total	Correct	Total	Correct
Simple	Select	81	70	22	21
	Where	55	33	19	13
	Where & Group by	2	2	0	0
	Where & Order by	2	0	0	0
	Order by	71	47	19	13
	Order by & Group by	31	23	8	7
	Group by	24	15	8	5
	Having & Group by	20	17	2	1
	Having & Group by & Where	1	1	0	0
Join	Simple join	25	15	6	4
	Join & Order by	27	23	8	5
	Join & Group by	8	2	2	0
	Join & Order by & Group by	10	3	4	0
	Join & Group by & Having	16	4	7	5
	Join & Where	28	17	9	5
Sub-select	Simple sub-select	26	10	6	2
	Sub-select & Where	2	0	0	0
		429	**282**	**120**	**81**

4.3. Discussion

The number of correct queries, based on the execution accuracy, totaled 298 for the training set and 85 for the development set. However, when focusing on the exact set match accuracy, Table 5 shows a decrease in the number of correct queries, with 282 correct queries in the training set and 81 in the development set. After analyzing the mismatched queries, the following observations were made:

- Ten queries from the training and two from the development sets were missing the *ASC* clause. However, the absence of this clause did not impact the output, as the query engine automatically orders the column alphabetically.
- Three queries from the training and one from the development sets included an unnecessary *GROUP BY* clause. The presence of the keyword "*each*" in the original natural language questions triggered a grouping operation. However, the output was unaffected because no repeated values were inside the primary key column.
- One of the questions from the training set refers to two different tables, leading the algorithm to execute a *JOIN*. However, this operation is unnecessary because the columns needed to answer the question are present only in one table. Nonetheless, the content remained unchanged because the values in the primary and foreign keys were identical.
- The remaining cases in the training set consisted of one question where the missing *BETWEEN* clause was replaced with a double *WHERE* clause, generating the same output. Another case involved a missing *ORDER BY* clause, where the algorithm failed to identify the column to sort by. However, the column values were already correctly ordered, producing the same result. In the development set, the remaining mismatched question featured an additional *ASC* clause in the predicted query. The natural language question explicitly requested the column to be ordered

[2] Every experiment was executed on a MacBook Pro laptop with an M1 pro CPU chip and 16 GB of RAM.

in ascending order. Thus, the resulting query fully reflects the natural language request. As in the previous case, the column was already correctly ordered, and both queries generated the correct output.

Among the issues discussed, only two have a significant impact. The first is the unnecessary *JOIN* operation, which can lead to different results in different questions. The second issue involves the missing *ORDER BY* clause, which is crucial for aligning the output with the user's specific request. The remaining issues, such as having an extra *ASC* clause, do not affect the final output, even if some clauses differ.

Upon careful examination, it is possible to state that the algorithm mainly struggles to replicate certain types of join and nested queries. An analysis of the errors associated with each query type revealed the following findings:

- Simple Queries — In the category of simple queries, the most common mistakes appeared within the *WHERE* and *ORDER BY* clauses. Difficulties in the "*WHERE*" clause arise when the algorithm fails to identify values to filter due to linguistic patterns not accounted for in the algorithm. For instance, in the question, "*What are the names of actors who have been in the musical titled The Phantom of the Opera*", the algorithm was unable to recognize that the pattern "*The Phantom of the Opera*" represents a value, as it was not enclosed in quotation marks and consisted of a combination other than just proper nouns. Regarding the *ORDER BY* clause, there are ten cases where adding an *ASC* clause would have produced a correct query. Moreover, within the "*GROUP BY*" query type, some queries failed to produce the correct result due to a linguistic pattern where the term "*different*" can be associated with both the "*DISTINCT*" clause and the "*GROUP BY*" clause. However, the vocabulary used in the proposed algorithm categorizes it as a "*DISTINCT*" clause, leading to inaccuracies.
- Join Queries — The main challenge of the *JOIN* queries arises in both simple operations and those involving additional clauses such as *GROUP BY* and *HAVING*. In particular, when a question mentions multiple tables, the join operator attempts to combine them. However, the problem occurs when more tables are mentioned in the question than are actually necessary to build the query, and they do not fall under linguistic exceptions. For example, the question "*Show the name and number of employees for the departments managed by heads whose temporary acting value is 'Yes'?*" identifies three tables "*departments*", "*heads*", and "*management*", while the "*heads*" table should be ignored. Additional issues emerge when a query requires a join and other clauses such as *ORDER BY* and *GROUP BY*. The complexity of structuring an accurate query increases as more clauses are incorporated.
- Sub-Select - The *Sub-Select* clause is the least appearing clause in both sets, but it is also the most difficult to assemble. The linguistic patterns in this clause are challenging to distinguish from others. For example, the word "*have*" can initiate three different clauses: *HAVING*, *ORDER BY*, or *Sub-Select*. Due to this complexity, the algorithm often struggles to correctly identify when a *Sub-Select* clause is needed.

Another relevant challenge is the overlap of structuring rules. The rules for assembling the query in the proposed algorithm are intrinsically related to linguistic patterns. When a question is very long and contains diverse patterns that can be matched, some of these patterns may overlap. For example, in the question "*Of all the contestants who got voted, what is the contestant number and name of the contestant who got least votes?*", instead of selecting the column *contestantNumber*, the algorithm added a *COUNT* operator to the *SELECT* clause. In this example, a keyword from the vocabulary triggered a specific operator, which overlapped with the correct column selection. Despite these challenges, the Neural Rule Based algorithm demonstrated the ability to correctly identify which columns to select and what operations to perform in most of the questions within the training and the development sets.

4.4. Comparison with other methods

We compared our text-to-SQL algorithm to three models trained on the SPIDER benchmark, using the same databases we pre-selected for our experiment.

For this purpose, we considered *SyntaxSQLNet*, *SyntaxSQLNetAugmented*, and *ValueNet* [55]. The first two models are architecturally similar to SQLNet [77] and TypeSQL [48]. However, in their research, the authors proposed using a syntax tree-based decoder to generate SQL queries. This decoder divides the process into nine modules, each responsible for predicting a distinct SQL clause. The augmented version differs by incorporating a cross-domain data augmentation method, providing more training samples for the model. The *ValueNet* implements an additional step beyond the training phase. This step employs the extracted database's attributes to identify candidate values in the natural language question (i.e., words that match the database's attributes). Once this step is complete, all the information, such as the question and the candidate values, is encoded and fed into the neural model.

Our system outperformed *SyntaxSQLNet* and *SyntaxSQLNetAugmented*, specifically chosen for comparison due to their syntax tree-based decoder, which is similar to our structural rules for SQL generation. On the development set for the pre-selected databases, our model achieved an execution accuracy of 70.83%, compared to 42.5% for *SyntaxSQLNet* and 47.5% for *SyntaxSQLNetAugmented*, surpassing their performance by 28.33% and 23.33%, respectively. The analysis revealed that our system more accurately identifies the correct column to select, even when dealing with semantically similar column names. Additionally, *SyntaxSQLNet*, in contrast to our approach, struggled to place the correct values in *WHERE* clauses and accurately structure *JOIN* operations, leading to less precise query formulation in these areas.

The *ValueNet* model outperformed our system in terms of execution accuracy by 15.17% on the development set. The model better identified the correct values to be placed in the *WHERE* clauses and the relationships between different tables during *JOIN* operations. However, this model required extensive training using a Tesla V100 GPU (32 GB memory) with an Intel(R) Xeon(R) CPU E5-2650 v4 (4 cores) and 16 GB unified memory. Considering that our approach does not require any training phase and can be run on a simple laptop, we regard its performance as satisfactory and the method as a valid alternative when extensive training is not feasible. At the moment, there is no specific benchmark to test zero-shot models in the text-to-SQL task, and there are two main constraints for achieving better results on the *SPIDER* benchmark:

- Database Representation — The proposed system has many rules that structure SQL queries without altering the table or column names. This means that the databases from the benchmark must have simple and concise schemas. Otherwise, the text-to-SQL algorithm may select incorrect columns or tables.
- Natural Language Diversity — The benchmark has many diverse questions, paraphrased several times to augment the number of possible queries. Since our system has not been trained on these questions, it may struggle to capture all the different linguistic structures.

5. Conclusion and future work

In this research, we developed a system to address the text-to-SQL task. The system is adaptable to new databases across different domains and effectively answers natural language questions requiring join operations across database tables. In particular, to tackle the cross-domain adaptability and the computational resources necessary to train text-to-SQL models, the proposed technique combines a set of

structuring and linguistic rules with a handwritten vocabulary and pre-trained embedding models to efficiently generate correct queries from natural language questions.

The results demonstrate the effectiveness of the proposed system. The performance differences compared to other methods stem from our approach's elimination of the training phase. Thus, it could not capture some of the relations between natural language questions and the SQL query from the selected benchmark. On the other hand, the models trained on this benchmark suffer from poor domain adaptability and high computational costs.

The experiment conducted on the pre-selected databases from the benchmark allowed us to answer both research questions posed in the introduction:

- RQ1 — The execution accuracy provided in Table 4 confirms that pre-trained models play an important role in the performance of the proposed NLIDB . The combination of pre-trained models with linguistic rules demonstrated that the proposed text-to-SQL algorithm can effectively identify the necessary tables and columns to answer the natural language question.
- RQ2 — The analysis of the exact set match for query types in Table 5 confirms that ruled-based techniques, coupled with a handwritten vocabulary, can generate working queries. Moreover, the experimental findings demonstrate that it is unnecessary to train a model on vast amounts of data or use extensive computational resources to develop a cross-domain model capable of generating correctly structured SQL queries.

The main limitation of the text-to-SQL algorithm is its difficulty in generating correct queries when dealing with databases that do not follow the standard SQL naming conventions. Another issue arises when dealing with linguistic patterns not covered in the algorithm. Given these limitations, we propose several directions for future work, such as:

- Create a General Entity-Value Recognition — One way to deal with linguistic patterns not covered in this work is to create an algorithm similar to the Named Entity Recognition [78] but specifically for databases. This algorithm would help identify the correct column to answer a natural language question. For instance, if the user asks: "What is the year of creation of the AC/DC album?", the algorithm would recognize that AC/DC is a value from the "*bands*" column. Such an algorithm could correct errors related to the *WHERE* clause.
- Hybrid Approach — Another way to handle unknown linguistic patterns is to employ a hybrid approach that includes a training phase where the model learns to understand the user's question and partitions it into several clauses. For instance, the model could identify which part of the question corresponds to the *SELECT* clause and which parts pertain to filtering clauses. Combining this model with pre-trained models for semantic search and rule-based techniques for query structuring could generate a robust system capable of accurately identifying the necessary SQL clauses while maintaining domain adaptability.
- Automatic Column Labeling — To address poor column naming, an interesting approach would be to develop a model that analyzes column names and proposes changes if they are not semantically clear or understandable.
- Other fundamental SQL operations - While this study focused solely on the *Read* operation, extending the existing algorithm to include the other essential SQL operations — Create, Update, and Delete — would be highly beneficial.

CRediT authorship contribution statement

Yuriy Perezhohin: Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Fernando Peres:** Writing – original draft, Methodology, Conceptualization. **Mauro Castelli:** Writing – original draft, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare no competing interests.

Acknowledgments

This work was supported by MyNorth AI Research. This work was partially supported by national funds through the FCT (Fundação para a Ciência e a Tecnologia) by the project UIDB/04152/2020 - Centro de Investigação em Gestão de Informação (MagIC)/NOVA IMS.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.array.2024.100368.

Data availability

The data used in this work are publicly available at https://yale-lily.github.io/spider. Github Link to the project: https://github.com/yuriyvnv/Neural-Rule-Based-model-for-Text-to-SQL.

References

[1] Majhadi K, Machkour M. The history and recent advances of natural language interfaces for databases querying. In: E3S web of conferences. Vol. 229, EDP Sciences; 2021, p. 01039.

[2] Affolter K, Stockinger K, Bernstein A. A comparative survey of recent natural language interfaces for databases. VLDB J 2019;28:793–819.

[3] Őzcan F, Quamar A, Sen J, Lei C, Efthymiou V. State of the art and open challenges in natural language interfaces to data. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data. 2020, p. 2629–36.

[4] Abbas S, Khan MU, Lee SU-J, Abbas A, Bashir AK. A review of NLIDB with deep learning: findings, challenges and open issues. IEEE Access 2022.

[5] Nascimento ER, Garcıa GM, Feijó L, Victorio WZ, Izquierdo YT, de Oliveira AR, Coelho GM, Lemos M, Garcia RL, Leme LAP, et al. Text-to-SQL meets the real-world. In: 26th int. conf. on enterprise info. sys. 2024.

[6] Qin B, Hui B, Wang L, Yang M, Li J, Li B, Geng R, Cao R, Sun J, Si L, et al. A survey on text-to-SQL parsing: Concepts, methods, and future directions. 2022, arXiv preprint arXiv:2208.13629.

[7] Li H. Deep learning for natural language processing: advantages and challenges. Natl Sci Rev 2018;5(1):24–6.

[8] Tan Z, Chen T, Zhang Z, Liu H. Sparsity-guided holistic explanation for llms with interpretable inference-time intervention. In: Proceedings of the AAAI conference on artificial intelligence. Vol. 38, 2024, p. 21619–27.

[9] Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, et al. Language models are few-shot learners. Adv Neural Inf Process Syst 2020;33:1877–901.

[10] Liu A, Hu X, Wen L, Yu PS. A comprehensive evaluation of ChatGPT's zero-shot text-to-SQL capability. 2023, arXiv preprint arXiv:2303.13547.

[11] Quamar A, Efthymiou V, Lei C, Özcan F, et al. Natural language interfaces to data. Found Trends Databases 2022;11(4):319–414.

[12] Winograd T. Five lectures on artificial intelligence. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE; 1974.

[13] Codd EF. A relational model of data for large shared data banks. Commun ACM 1970;13(6):377–87.

[14] Chamberlin DD, Boyce RF. SEQUEL: A structured english query language. In: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on data description, access and control. 1974, p. 249–64.

[15] Woods WA. Progress in natural language understanding: an application to lunar geology. In: Proceedings of the June 4-8, 1973, national computer conference and exposition. 1973, p. 441–50.

[16] Blunschi L, Jossen C, Kossman D, Mori M, Stockinger K. Soda: Generating sql for business users. 2012, arXiv preprint arXiv:1207.0134.

[17] Kaufmann E, Bernstein A, Fischer L. NLP-reduce: A naive but domainindependent natural language interface for querying ontologies. In: 4th European semantic web conference ESWC. Springer Berlin; 2007, p. 1–2.

[18] Bergamaschi S, Guerra F, Interlandi M, Trillo Lado R, Velegrakis Y, et al. QUEST: a keyword search system for relational data based on semantic and machine learning techniques. Proc VLDB Endow 2013;6:1222–5.

[19] Johnson T. Natural language computing: the commercial applications. Knowl Eng Rev 1984;1(3):11–23.

[20] Li F, Jagadish HV. NaLIR: an interactive natural language interface for querying relational databases. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. 2014, p. 709–12.
[21] Hendrix GG, Sacerdoti ED, Sagalowicz D, Slocum J. Developing a natural language interface to complex data. ACM Trans Database Syst 1978;3(2):105–47.
[22] Song D, Schilder F, Smiley C, Brew C, Zielund T, Bretz H, Martin R, Dale C, Duprey J, Miller T, et al. TR discover: A natural language interface for querying and analyzing interlinked datasets. In: International semantic web conference. Springer; 2015, p. 21–37.
[23] Zhang B, Ye Y, Du G, Hu X, Li Z, Yang S, Liu CH, Zhao R, Li Z, Mao H. Benchmarking the text-to-sql capability of large language models: A comprehensive evaluation. 2024, arXiv preprint arXiv:2403.02951.
[24] Simitsis A, Koutrika G, Ioannidis Y. Précis: from unstructured keywords as queries to structured databases as answers. VLDB J 2008;17(1):117–49.
[25] Ganti V, He Y, Xin D. Keyword++ a framework to improve keyword search over entity databases. Proc VLDB Endow 2010;3(1–2):711–22.
[26] Wong YW, Mooney R. Learning for semantic parsing with statistical machine translation. In: Proceedings of the human language technology conference of the NAACL, main conference. 2006, p. 439–46.
[27] Waltinger U, Tecuci D, Olteanu M, Mocanu V, Sullivan S. Usi answers: Natural language question answering over (semi-) structured industry data. In: Twenty-fifth IAAI conference. 2013.
[28] Saha D, Floratou A, Sankaranarayanan K, Minhas UF, Mittal AR, Özcan F. ATHENA: an ontology-driven system for natural language querying over relational data stores. Proc VLDB Endow 2016;9(12):1209–20.
[29] Yaghmazadeh N, Wang Y, Dillig I, Dillig T. Sqlizer: query synthesis from natural language. Proc ACM Program Lang 2017;1(OOPSLA):1–26.
[30] Jamil HM. Knowledge rich natural language queries over structured biological databases. In: Proceedings of the 8th ACM international conference on bioinformatics, computational biology, and health informatics. 2017, p. 352–61.
[31] Yu T, Zhang R, Yang K, Yasunaga M, Wang D, Li Z, Ma J, Li I, Yao Q, Roman S, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. 2018, arXiv preprint arXiv:1809.08887.
[32] Li J, Hui B, Qu G, Yang J, Li B, Li B, Wang B, Qin B, Geng R, Huo N, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. Adv Neural Inf Process Syst 2024;36.
[33] Zhong V, Xiong C, Socher R. Seq2sql: Generating structured queries from natural language using reinforcement learning. 2017, arXiv preprint arXiv:1709.00103.
[34] Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. Adv Neural Inf Process Syst 2014;27.
[35] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput 1997;9(8):1735–80.
[36] He P, Mao Y, Chakrabarti K, Chen W. X-SQL: reinforce schema representation with context. 2019, arXiv preprint arXiv:1908.08113.
[37] Xu X, Liu C, Song D. Sqlnet: Generating structured queries from natural language without reinforcement learning. 2017, arXiv preprint arXiv:1711.04436.
[38] Gori M, Monfardini G, Scarselli F. A new model for learning in graph domains. In: Proceedings. 2005 IEEE international joint conference on neural networks, 2005. Vol. 2, IEEE; 2005, p. 729–34.
[39] Bogin B, Gardner M, Berant J. Representing schema structure with graph neural networks for text-to-SQL parsing. 2019, arXiv preprint arXiv:1905.06241.
[40] Bogin B, Gardner M, Berant J. Global reasoning over database structures for text-to-sql parsing. 2019, arXiv preprint arXiv:1908.11214.
[41] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. 2016, arXiv preprint arXiv:1609.02907.
[42] Qi J, Tang J, He Z, Wan X, Zhou C, Wang X, Zhang Q, Lin Z. RASAT: Integrating relational structures into pretrained Seq2Seq model for text-to-SQL. 2022, arXiv preprint arXiv:2205.06983.
[43] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, Zhou Y, Li W, Liu PJ, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res 2020;21(140):1–67.
[44] Elgohary A, Hosseini S, Awadallah AH. Speak to your parser: Interactive text-to-SQL with natural language feedback. 2020, arXiv preprint arXiv:2005.02539.
[45] Katsogiannis-Meimarakis G, Koutrika G. A survey on deep learning approaches for text-to-SQL. VLDB J 2023;1–32.
[46] Stojkovic J, Choukse E, Zhang C, Goiri I, Torrellas J. Towards greener LLMs: Bringing energy-efficiency to the forefront of LLM inference. 2024, arXiv preprint arXiv:2403.20306.
[47] Huang P-S, Wang C, Singh R, Yih W-t, He X. Natural language to structured query generation via meta-learning. 2018, arXiv preprint arXiv:1803.02400.
[48] Yu T, Li Z, Zhang Z, Zhang R, Radev D. Typesql: Knowledge-based type-aware neural text-to-sql generation. 2018, arXiv preprint arXiv:1804.09769.
[49] Lyu Q, Chakrabarti K, Hathi S, Kundu S, Zhang J, Chen Z. Hybrid ranking network for text-to-sql. 2020, arXiv preprint arXiv:2008.04759.
[50] Xuan K, Wang Y, Wang Y, Wen Z, Dong Y. Sead: End-to-end text-to-sql generation with schema-aware denoising. 2021, arXiv preprint arXiv:2105.07911.
[51] Lin K, Bogin B, Neumann M, Berant J, Gardner M. Grammar-based neural text-to-sql generation. 2019, arXiv preprint arXiv:1905.13326.
[52] Guo J, Zhan Z, Gao Y, Xiao Y, Lou J-G, Liu T, Zhang D. Towards complex text-to-sql in cross-domain database with intermediate representation. 2019, arXiv preprint arXiv:1905.08205.
[53] Wang B, Shin R, Liu X, Polozov O, Richardson M. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. 2019, arXiv preprint arXiv:1911.04942.
[54] Zeng J, Lin XV, Xiong C, Socher R, Lyu MR, King I, Hoi SC. Photon: a robust cross-domain text-to-SQL system. 2020, arXiv preprint arXiv:2007.15280.
[55] Brunner U, Stockinger K. Valuenet: A natural language-to-sql system that learns from database information. In: 2021 IEEE 37th international conference on data engineering. ICDE, IEEE; 2021, p. 2177–82.
[56] Scholak T, Schucher N, Bahdanau D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. 2021, arXiv preprint arXiv:2109.05093.
[57] Cao R, Chen L, Chen Z, Zhao Y, Zhu S, Yu K. LGESQL: line graph enhanced text-to-SQL model with mixed local and non-local relations. 2021, arXiv preprint arXiv:2106.01093.
[58] Li H, Zhang J, Li C, Chen H. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In: Proceedings of the thirty-seventh AAAI conference on artificial intelligence. AAAI, 2023.
[59] Hui B, Geng R, Wang L, Qin B, Li B, Sun J, Li Y. S^{2SQL}: Injecting syntax to question-schema interaction graph encoder for text-to-SQL parsers. 2022, arXiv preprint arXiv:2203.06958.
[60] Zeng L, Parthasarathi SHK, Hakkani-Tur D. N-best hypotheses reranking for text-to-sql systems. In: 2022 IEEE spoken language technology workshop. SLT, IEEE; 2023, p. 663–70.
[61] Zhao Y, Jiang J, Hu Y, Lan W, Zhu H, Chauhan A, Li A, Pan L, Wang J, Hang C-W, et al. Importance of synthesizing high-quality data for text-to-SQL parsing. 2022, arXiv preprint arXiv:2212.08785.
[62] Li J, Hui B, Cheng R, Qin B, Ma C, Huo N, Huang F, Du W, Si L, Li Y. Graphix-T5: Mixing pre-trained transformers with graph-aware layers for text-to-SQL parsing. 2023, arXiv preprint arXiv:2301.07507.
[63] Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. Adv Neural Inf Process Syst 2013;26.
[64] Harris ZS. Distributional structure. Word 1954;10(2–3):146–62.
[65] Sparck Jones K. A statistical interpretation of term specificity and its application in retrieval. J Doc 1972;28(1):11–21.
[66] Devlin J, Chang M-W, Lee K, Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018, arXiv preprint arXiv:1810.04805.
[67] Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. Roberta: A robustly optimized bert pretraining approach. 2019, arXiv preprint arXiv:1907.11692.
[68] Lan Z, Chen M, Goodman S, Gimpel K, Sharma P, Soricut R. Albert: A lite bert for self-supervised learning of language representations. 2019, arXiv preprint arXiv:1909.11942.
[69] Papamichail A, Zarras AV, Vassiliadis P. Do people use naming conventions in SQL programming? In: SOFSEM 2020: theory and practice of computer science: 46th international conference on current trends in theory and practice of informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, proceedings 46. Springer; 2020, p. 429–40.
[70] Axler S. Linear algebra done right. Springer Science & Business Media; 1997.
[71] Salton G, Buckley C. Term-weighting approaches in automatic text retrieval. Inf Process Manag 1988;24(5):513–23.
[72] Song K, Tan X, Qin T, Lu J, Liu T-Y. Mpnet: Masked and permuted pre-training for language understanding. Adv Neural Inf Process Syst 2020;33:16857–67.
[73] Sanh V, Debut L, Chaumond J, Wolf T. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. 2019, arXiv preprint arXiv:1910.01108.
[74] Wang W, Wei F, Dong L, Bao H, Yang N, Zhou M. MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. 2020, arXiv:2002.10957.
[75] Manning CD, Surdeanu M, Bauer J, Finkel JR, Bethard S, McClosky D. The stanford corenlp natural language processing toolkit. In: Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations. 2014, p. 55–60.
[76] Usta A, Karakayali A, Ulusoy Ö. Dbtagger: Multi-task learning for keyword mapping in nlidbs using bi-directional recurrent neural networks. 2021, arXiv preprint arXiv:2101.04226.
[77] Xu X, Liu C, Song D. Sqlnet: Generating structured queries from natural language without reinforcement learning. arxiv 2017. 2017, arXiv preprint arXiv:1711.04436.
[78] Rau LF. Extracting company names from text. In: Proceedings the seventh IEEE conference on artificial intelligence application. IEEE Computer Society; 1991, p. 29–30.