Machine Learning System for Natural Language to SQL Translation

Rareş Mihai Dinu
University of Craiova
raresmihai18@gmail.com

Marian Cristian Mihăescu
University of Craiova
cristian.mihaescu@edu.ucv.ro

Traian Eugen Rebedea
University Politehnica of Bucharest
traian.rebedea@cs.pub.ro

ABSTRACT
This paper introduces a system specifically designed for natural language to SQL translation. Leveraging the power of machine learning, the system incorporates deep learning models, namely RAT-SQL and RoBERTa, to improve the accuracy and effectiveness of the translation process. The paper provides an in-depth overview of the system's high-level design, including using RAT-SQL as the core model and integrating the BERT approach for enhanced performance. It also discusses the dataset employed during the system's development and presents the results and conclusions. By utilising the synergy of RAT-SQL and RoBERTa, the system demonstrates promising advancements in the natural language to SQL translation domain, showcasing its potential to simplify and streamline the interaction between human language and structured database queries.

Author Keywords
BERT, Transformer, Spider, Text-to-SQL.

ACM Classification Keywords
H.5.m. Information interfaces and presentation: Miscellaneous.

General Terms
Human Factors; Design; Measurement.

DOI: 10.37789/rochi.2023.1.1.3

INTRODUCTION

We live in the era of technology merging into our day-to-day lives, either personally or professionally; we depend on it in one way or another. SQL is a broadly used language related to databases and commands that allows implicit users to retrieve information based on their input. Despite being present for quite a long time, there are still problems, or better said, difficulties when approaching a larger structure that needs to be tackled. The motivation comes from the desire to understand and further explore the natural language processing domain, a popular topic nowadays that aims to expand and generate better and better results.

The purpose of the application stands in the desire to create an environment meant for translating human-written text into SQL queries. This application can be further used by those who want to create queries based on a provided database input swiftly or for those who wish to explore the structure of the database.

Aside from the practical approach, using different models to achieve the previously stated goal adds to the final value of the project. Having multiple workspaces and understanding such models' structure and behaviour will always be a plus for the intended work.

The system provides a linear implication for the user, from uploading the database to typing in their request. Upon receiving the user's command, the application generates the SQL query through the help of RAT-SQL [1], but not quite fully. In that regard, RoBERTa [2], through its question-answering ability, provides the missing context, filling in the gaps. The resulting output is a fully operational SQL query that can render information in the given environment. It is noted that both mentioned models are based on the BERT [3] architecture.

Considering the specialisation, the presented application is aimed at those who use SQL regularly and want to optimise their efficiency and productivity. Moreover, the machine learning system can also be used as a navigation environment for a data set/database structure unknown to the advanced user. Users can formulate requirements and questions the system will translate into SQL. Thus, those who know the structure given to the system can use the generated queries or, otherwise, better understand the shape of the loaded structure through the queries.

At the core of an application that implies any user's input stands the human-computer interaction baseline. In this situation, the system should be able to handle the barriers of the natural language to which it is exposed. In other words, the natural language input should be broad enough so that the users can express themselves freely, without any occurring issues regarding their input. The text-to-SQL system output should yield the corresponding query to the manual input previously mentioned. Simply put, the system should be designed so that it will not be punishing for the users based on their input and their way of writing prompts; it must broadly understand the concepts of the language and let the inputs be as expressive or redundant as possible.

Technically speaking, the project utilises the Python programming language and a range of specialised libraries,

such as torch,[1] transformers[2], and nltk,[3] for natural language processing and machine learning tasks. Developed in Google Colab Pro, a cloud-based development environment similar to Jupyter, the project focuses on machine learning and data analysis.

The cloud nature of Colab Pro allows for remote accessibility, with virtualised hardware resources, ensuring independence from the local machine. Two models are employed for machine learning: RAT-SQL and RoBERTa Base Squad 2. RAT-SQL is the base for translating natural language to SQL, with preprocessing and training performed using the Spider dataset [4]. While RAT-SQL provides the fundamental query structure, it lacks specific details. To overcome these limitations, RoBERTa Base Squad 2, a pre-trained model specialised in question-answering, is utilised. Combining RoBERTa and code improvements allows the system to generate questions that address RAT-SQL gaps, resulting in a complete and syntactically correct query.

RELATED WORK

The state-of-the-art domain in discussion comprises different concepts and ideas, each having an intricate solution to the problem. The direction of assessment follows the dataset used and the models and techniques involved in their pretraining and evaluation methods. Since there are plenty of approaches, they have advantages and disadvantages.

Deng et al. [5] cover the performance of the supervised structure pretraining framework (StruG) for text-to-SQL. It fundamentally uses RAT-SQL, which, when writing the paper, is the state-of-the-art model according to the official leaderboard. By using the Spider dataset, they proved its outperformance compared to RAT-SQL. Despite the better performance, the paper mentions the complementary matter of the experiment. The core technique used in the previous article is semantic parsing, which stays at the root of the Spider dataset [4]. As mentioned by the paper, the dataset underlines the extensive use of multi-table databases and complex queries. To be noted, Deng et al. [5] used both a manual and automatic approach to achieve higher scores. At the same time, the BERT Large considerably ranks lower due to the non-specificity of the dataset and the "more realistic" alignment settings of the text tables.

Finegan-Dollak et al. [6] cover the generalisation of systems to realistic unseen data. The probation of the performance was based on the variation of the *seq2seq* model, with the implication of the attention technique. The paper's main takeaway results in the current systems' inability to understand the properties of human-generated datasets. Furthermore, the experiments proved the exaggerated perception of the system's ability to generalise to new or unseen data. This particularity defeats the point of the use of such systems. Making good use of the considerations of the users, which are the primary target, implies a low ability to emerge and be used on a large scale, especially in the text-to-SQL approach.

Yu et al. [7] underline using a similar system to SQLNet[4] or TypeSQL[5] to achieve the human interactivity proposed. The natural language to SQL task implies the swift translation of the human-generated text to executable queries, which, in the presented paper, are presented in a general accent. By that, the authors underline that, due to the dataset approach (WikiSQL[6]), the queries generated lack the use of GROUP BY and JOIN keywords, which are vital for more complex alignments with the database. In other words, this particular approach does not fully implement the entire SQL behaviour, the behaviour we would consider in the first place. It is obvious that, due to the limited keywords used and the old-fashioned knowledge-based type-awareness, we would consider a better alternative: the attention mechanism and relational reasoning.

Narechania et al. [8] address the entire application flow regarding the user input to the executed database queries. In other words, the infrastructure proposed designs a debug-like environment where the user can handle different columns and approaches while running the system. A downside to the proposed method is that tokens can only be mapped if they are already applied. In this particularity, a freedom constraint will not allow the users to experiment with the dataset fully. Additionally, the system described, similarly to Yu et al. [7], does not follow the entire SQL vocabulary; keywords such as LIKE and OVER are absent.

Shah et al. [9] presented complies with the architectural progress from the seq2seq to the BERT model. Despite having an interesting elaborative idea, the results yield similar results to RAT-SQL's. The main compression is, in this case, the ready-to-go formality of the application in which, upon entering the desired prompt, the query is executed immediately. In other words, Shah et al. [9] outlines a very similar to RAT-SQL in its final attempts, which is coloured furthermore by an adequate interface for such a task.

---

[1] Torch, https://pytorch.org/, last accessed at 30.06.2023

[2] Transformers, https://huggingface.co/docs/transformers, last accessed at 30.06.2023

[3] NLTK, https://www.nltk.org/, last accessed at 30.06.2023

[4] SQLNet, https://github.com/xiaojunxu/SQLNet, last accessed at 01.07.2023

[5] TypeSQL, https://github.com/taoyds/typesql, last accessed at 01.07.2023

[6] WikiSQL, https://github.com/salesforce/WikiSQL, last accessed at 01.07.2023

An interesting approach presented by Gan et al. [10] implies the usage of two categories: synonym annotations and adversarial training. These approaches create a broader spectrum of understanding for the model when being trained and, in other words, help in better understanding the given context and prompt. This approach is achieved by manually modifying the Spider dataset that offers more than one nuance to a certain schema word. Despite the interesting process that allows the users to have a more colourful vocabulary, most state-of-the-art models have a significantly reduced performance on the modified benchmark presented.

On the visual side and understanding the processes partaken by the models through statistical output, Mitra et al. [11] propose this approach as there are no historical trials for such practices. It suggests that chatbots may create a better understanding environment and augment the current context. The chatbots can further elaborate the prompt by addressing questions or asking for more input. Despite the view-broadening approach, these are not perfect since, by adding another question, the direction might change drastically.

Ning et al. [12] have an interesting concept: to consolidate the human-computer interaction and natural language concept together, the state-of-the-art natural language to SQL models were studied based on the errors made. By that, different methods of error-handling and recovery were conducted and tested via many users. The results show that the handling facilitates the non-experienced users. Still, the mechanisms developed do not significantly improve the quality of the queries generated and the time of completion when the said queries are executed.

In other words, the state-of-the-art regarding text-to-SQL comprises advanced neural networks like transformers and pre-trained models such as BERT to convert natural language queries into structured SQL commands effectively. Despite the said approaches' complexity, limitations and challenges persist, including handling complex queries comprising multiple clauses and joins, model generalisation across databases, language ambiguity and so on.

SYSTEM DESIGN

The application's primary purpose in discussion is to translate natural language into SQL queries through user inputs. The system is based on machine learning models either trained on the Spider dataset or pre-trained and ready to use through different Python libraries.

The system is a command line-based application that requires the user's input, which is both uploading files and typing in text and outputs the query based on the initial circumstances.

The main flow of the application is relatively simple: the user can upload their database structure based on which the files tables.json and dev.json are rerendered. Using RAT-SQL the user can generate SQL queries. RAT-SQL has a flaw: the predicates of the queries are replaced by the word 'terminal', which is fixed through RoBERTa Base SQuAD 2. Figure 1 broadly describes the processes involved.

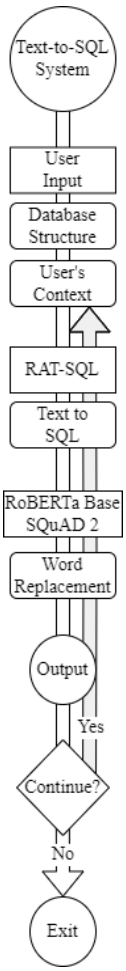

Figure 1. System workflow

RAT-SQL

RAT-SQL is the core functionality of the system, the gate that allows the application to properly translate human-generated text into SQL queries.

RAT-SQL was originally based on Docker images, a technology that Google Colab does not allow. The best solution was manually creating the virtual environment, namely CUDA, line by line in Google Colab for RAT-SQL to work properly. Despite the Docker fashion of the project, the initial phase was solely creating the right dependencies between the libraries.

There were some other issues along the way; namely, the Stanford NLP Server would always run out and errors at the level of RAT-SQL. The first issue was simple to overcome by simply changing the connection port. Updates at the sudo level of Google Colab covered the errors retrieved.

BERT was the decisive point in the training stage of RAT-SQL, representing the project's continuous and integrative part. In other words, BERT was decided to be the technology to be further used during the project's development. The main point of using BERT was the attention module and the higher learning rate provided.

The provided environment implications are mandatory for the smooth running of RAT-SQL, and its primary goal is the query generation at the user's input.

User input

As discussed, the RAT-SQL part represents the system's core functionality, which allows the natural language translation to SQL code. Since the user plays another vital role, we will look closely at their interaction with the application and the steps to orchestrate the communication.

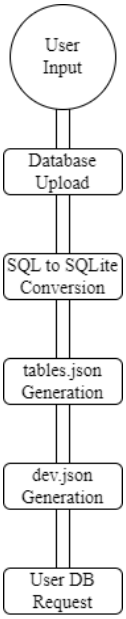

Figure 2. User input flow

Since the SQL code generated is strongly bonded with the files related to the database structure, based on the user input, RAT-SQL has to be configured on newly generated files to only operate on the given set of data.

After entering questions, the user is asked to name and upload a database structure to the system. Upon loading the SQL file, the following documents are created:

- *database_name.sqlite*, *schema.sql*: The schema is the given structure, while the sqlite file is the translated schema in SQLite;
- *tables.json*: The tables of the database in a JSON format;
- *dev.json*: The development file. It is essential for RAT-SQL at any implementation stage.

The files dev.json and tables.json are mandatory for RAT-SQL since they are the basis of the inferring process.

Having both the user input, summarised in figure 2, and RAT-SQL implemented and running, based on a database structure referring to geography, we can ask the following question: "*Show all details regarding cities from Romania.*" RAT-SQL generates the following SQL query based on the given input: "*SELECT * FROM cities WHERE cities.country = 'terminal'*". The schema further exemplifies the system's user input and RAT-SQL flow.

RoBERTa Base SQuAD 2

RoBERTa has been used in the text-to-SQL system to patch the terminal keyword into the appropriate query predicates. The following section will define the procedures considered to achieve the abovementioned goal using its question-answering capabilities.

As mentioned, RoBERTa is mainly used to find the replacing word for the terminal tokens. In this context, a workflow was created so that all terminals were found and replaced by the appropriate words. Both the question and the generated output from the previous section will be considered for better exemplification.

The first step is the tokenisation and normalisation of the given query. By that, we would obtain a list of uncapitalised words that exclude non-primary punctuation marks.

Secondly, the terminal words are to be found and indexed and upon seeing one, a pair of the token and the subject is taken. The subject is the first token that appears right after an SQL keyword. For instance, if considering the cities' query, the pair of subject-terminal would be ['*cities.country = terminal*'].

Next, having the subject, we want to know the quantity of it. This, in most considered cases, is represented by the attribution sign. If any comparison signs were to be executed, the quantity would be proportional to it. "*More than/after*" would be attributed to > since it applies to number and date types.

Having all the components of a question, we can populate the question structure that is based on the proceeding SQL keyword. RoBERTa is provided with the compounded question and the initial question asked by the user. RoBERTa answers it, and the response is further processed according to different nuances.

Having the answers to the generated questions, there are four main points taken into consideration when further processing the answer retrieved, namely, the SQL specific keywords 'LIMIT', 'LIKE' and 'BETWEEN' as well as handling the number conversion from string to integer or float, based on the given context.

The easiest to get around is the keyword '*LIMIT*'. Instead of creating a subject-quantity question, we can simply ask, upon

identifying this keyword, a standard question that allows the answer of limitation.

The keyword 'LIKE' is handled based on context. As we all know, the main areas that LIKE can cover are the words that begin, contain or end in a set of letters or substrings. Considering the words in the context, such as "*start*", "*have*", "*include*", "*end*", etc. we can simply append the necessary percent signs accordingly.

The keyword '*BETWEEN*' covers two terminal tokens. The output is the same whether the answers are withdrawn from the first or the second. The best solution is to discard an answer and split the kept one into two separate ones.

The number translation was based on the library *word2number* and *regex*. Word2number has the slight flaw of summing two consecutive numbers, such as "*one two*" would result in "*3*". Regardless of that, the library only keeps the number-related words, so answers such as "*two large compartments*" would be turned into "*2*". If the answer does not have number-related words, an error is thrown. When this case occurs, regex is used, and the numbers are translated into integers or, finally, float values.

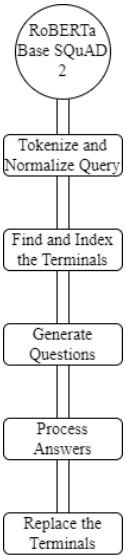

Figure 3. RoBERTa Base SQuAD 2 flow

Having an equal number of terminal tokens and answers, we can now replace them one by one in the initial query, but we have to keep in mind whether or not it is a number or not since quotations are needed only for strings.

Figure 3 describes the processes summarily.

The Dataset

Spider was used as the main source for training RAT-SQL. It is a large-scale, complex, cross-domain dataset excellent for natural language processing (NLP) models and approaches.

As mentioned in the introductory section of the dataset, Spider consists of several files mandatory for training and evaluating machine learning models. As a prime step, we will look at its general structure and then at a more in-depth description of each file.

Spider provides various files meant to be used in the creation of text-to-SQL models. The files cover the main flow for the training and evaluation stages. Right off the bat, the dataset has the following hierarchic structure:

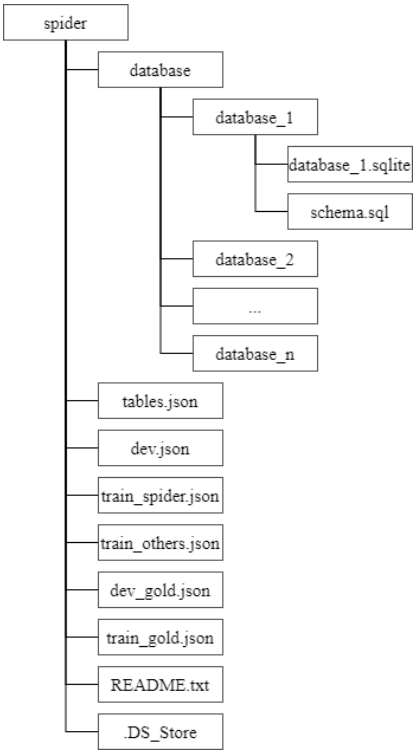

Figure 4. Spider dataset file architecture

Each file has an upright importance and a unique structure , the hierarchal distribution of files being presented in figure 4. Managing to create such a complex environment, the files and their inner format, their fields to be more exact, are an important topic to acknowledge:

- *dev.json*: A JSON file that contains a subset of the Spider dataset delivered as an array specifically designed for development and evaluation purposes;
- *table.json*: A JSON file that provides the structure representation of the tables found in the Spider dataset;
- *train_others.json*, train_spider.json: JSON files that contain a subset of training data of the Spider dataset;
- *dev_gold.sql*, train_gold.sql: SQL files that provide the ideal output for the development and training

stages conducted under the previously mentioned files;

- *database folder*: A folder with the proper dataset as folders containing a number of exactly two files: schema.sql (the database structure) and *database_name.sqlite* (the SQLite equivalent);
- *README.txt*, .DS_Store: Miscellaneous files.

EXPERIMENTAL RESULTS

This section contains the results of testing the models mentioned in the implementation and design section. The outcomes and satisfaction will be covered, as well as the way the testing occurred for each machine learning model.

RAT-SQL – BERT

As mentioned above, due to the outstanding performance, the BERT version of RAT-SQL was used for the further investigations and usage.

Regarding the satisfactory testing and the percentage of adequate results, two main approaches were applied in order to define them:

- Manual testing: By entering inputs on a given data structure;
- Against ChatGPT [13]: By comparing their outputs based on the same data structure and context.

Manual testing and comparison represent the first, most important, and obvious way of testing anything. This means that different inputs were given, and the output queries were tested in an SQL environment in order to output the same components.

The results did not disappoint since RAT-SQL managed to provide excellent results with an accuracy of 70% - 80%, as described in the evaluation stage of the model. The flaw that stood up the most is how RAT-SQL selects the fields based on the input. Unless specifically written, RAT-SQL will not choose the most appropriate displayable field. For instance, if the context does not ask for the name of the products, RAT-SQL can decide to display the code, the production date, or any other field but the name.

The next experimental testing was against the well-known ChatGPT. The experiment followed the outcome of both models when given a database structure and different requests on that particular dataset. Considering that ChatGPT is the reference point for our core component, RAT-SQL, the experiment follows the correct level of the OpenAI model.

Considering that both the database and user-inputs were not changed to better suit any of the two models, the outcome yielded can be showcased using a table. The table below shows the similarities and differences between ChatGPT and RAT-SQL regarding the query structure and quality and the usage of names. The notes are provisory but crucial when comparing the two systems side-by-side.

Topic	RAT-SQL	ChatGPT
Similarities		
Query quality	Both models provided a solid outcome. ChatGPT had a better performance.	
Query structure	Both RAT-SQL and ChatGPT had a very similar approach to computing the query.	
Differences		
Qualified column names	It will provide non-crucial SELECT items unless specifically asked	It will return the best suited fields based on the input.
Table aliases	It will most likely not use aliases for tables when generating queries.	It will most likely use the initial letter(s) of the name of the tables.
Readability	Due to not using aliases, the user can navigate faster through the query.	Having aliases, it becomes difficult to understand longer queries.

Table 1. RAT-SQL vs ChatGPT

In other words, both outcomes are usable and work perfectly well. The sole notable difference that stands between the two is the style preference.

RoBERTa Base SQuAD 2

RoBERTa was used as a base model to revert the incompleteness of the generated output of RAT-SQL, namely, replacing the 'terminal' keywords to the appropriate subject for that particular query.

The main form of testing here was solely the manual one, which implied that the algorithm that englobed RoBERTa over some queries provided by the GOLD files. These were compared and evaluated one by one to sketch out any outliers there was.

As RoBERTa is a question-and-answer model, the results depend heavily on both the question addressed and the context given. Having that in mind, we can already see some flaws in that using a QnA model when replacing certain parts of a query can lead to inconsistencies and wrong answers due to the subjective side of it.

The most predictable errors can be overseen regarding extensive queries with repetitive subjects. A sample context would be to name the nations of which the official languages are English and French. The associated query contains repeated subjects regarding nationality etiquette; thus, RoBERTa is overwhelmed and replies with the same answer.

On top of that, for the dataset above, there is also an attribute that says whether or not a language is official, having the

values '*T*', for true, and '*F*', for false. This leads us in another unknown direction: the lack of knowledge regarding the datatypes and annotations from the dataset. The RoBERTa structure is built so that it only has access to the question asked and the query generated by RAT-SQL. No additional information is provided since that is the only input RoBERTa requires to function.

Last, regardless of how well the query and context are built, there is always room for subjective randomness. Subjective randomness refers to events, or in our case, events that cannot be predicted due to the nature of the model. A slightly different word in inputs can lead to totally different outcomes. The table below shows the context, RoBERTa generated answers and the preprocessed outcome of each concerning the cylinder subject.

Context	RoBERTa (QnA)		Resulting query
What is the maximum miles per gallon of the car with 8 cylinders or produced before 1980 ?	Q	cylinders is what?/ How much?	select max(mpg) from cars_data where cylinders = 'miles per gallon'
	A	miles per gallon	
What is the maximum mpg of the cars that had 8 cylinders or that were produced before 1980 ?	Q	cylinders is what?/ How much?	select max(mpg) from cars_data where cylinders = 8
	A	8	

Table 2. RoBERTa Base SQuAD 2 – flaws

As we can see, the context has a slightly changed structure, while the base query remains the same. By far, this is the most vulnerable point of RoBERTa. We cannot define specific rules for any other context since that would require an in-depth evaluation and testing process. In most cases, RoBERTa excels in replacing and enforcing the correct predicate to its appropriate place.

CONCLUSIONS

This paper presented the flow of an application with the functionality of natural language to SQL translations at its core, along with its design, dataset, and results. The system can be used via the command line by those who want to understand better or generate queries based on a given context.

The HCI aspect revolves around the accessibility approach with regard to the level of experience of the users. This type of system shall be comprised in such a manner that non-experienced users should have access to the structured dataset. In other words, the system should facilitate any type of users, regardless of their background.

The experiments revealed that RAT-SQL requires better accuracy using the BERT approach, while RoBERTa can have some issues in slightly different contexts, namely, subjective randomness. But, having more than decent outcomes, we can justify the coexistence of multiple purpose models in the same system.

The project's current state outlines the usability of different models coexisting within the same environment, namely, the text-to-SQL translation. Despite this successful collaboration, there are still some limitations that persist: the complexity of both the user and database are a direct factor that may lead to erroneous outputs, the vocabulary used cannot be limited, thus the surging numbers of error that may occur and, finally, the word replacement that suffers from the query abstraction and context ambiguity.

As for the foreseeable future, we are going to try to patch as much as possible the aforementioned drawbacks by expanding the training dataset or using several different datasets [14, 15] and improving the query analysis for word replacement.

REFERENCES

1. Wang, B., Shin, R., Liu, X., Polozov, O., & Richardson, M. (2019). Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. arXiv preprint arXiv:1911.04942.
2. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... & Stoyanov, V. (2019). Roberta: A robustly optimised bert pretraining approach. arXiv preprint arXiv:1907.11692.
3. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
4. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., ... & Radev, D. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887.
5. Deng, X., Awadallah, A. H., Meek, C., Polozov, O., Sun, H., & Richardson, M. (2020). Structure-grounded pretraining for text-to-sql. arXiv preprint arXiv:2010.12773.
6. Finegan-Dollak, C., Kummerfeld, J. K., Zhang, L., Ramanathan, K., Sadasivam, S., Zhang, R., & Radev, D. (2018). Improving text-to-sql evaluation methodology. arXiv preprint arXiv:1806.09029.
7. Yu, T., Li, Z., Zhang, Z., Zhang, R., & Radev, D. (2018). Typesql: Knowledge-based type-aware neural text-to-sql generation. arXiv preprint arXiv:1804.09769.
8. Narechania, A., Fourney, A., Lee, B., & Ramos, G. (2021, April). DIY: Assessing the correctness of natural

language to sql systems. In 26th International Conference on Intelligent User Interfaces (pp. 597-607).

9. Shah, D., Das, A., Shahane, A., Parikh, D., & Bari, P. (2021). Speakql natural language to sql. In ITM Web of Conferences (Vol. 40, p. 03018). EDP Sciences.
10. Gan, Y., Chen, X., Huang, Q., Purver, M., Woodward, J. R., Xie, J., & Huang, P. (2021). Towards robustness of text-to-SQL models against synonym substitution. arXiv preprint arXiv:2106.01065.
11. Mitra, R., Narechania, A., Endert, A., & Stasko, J. (2022). Facilitating conversational interaction in natural language interfaces for visualisation. In 2022 IEEE Visualization and Visual Analytics (VIS) (pp. 6-10). IEEE.
12. Ning, Z., Zhang, Z., Sun, T., Tian, Y., Zhang, T., & Li, T. J. J. (2023, March). An empirical study of model errors and user error discovery and repair strategies in natural language database queries. In Proceedings of the 28th International Conference on Intelligent User Interfaces (pp. 633-649).
13. Zhu, J. J., Jiang, J., Yang, M., & Ren, Z. J. (2023). ChatGPT and environmental research. Environmental Science & Technology.
14. Iacob, R. C. A., Brad, F., Apostol, E. S., Truică, C. O., Hosu, I. A., & Rebedea, T. (2020). Neural approaches for natural language interfaces to databases: A survey. In proceedings of the 28th International Conference on Computational Linguistics (pp. 381-395).
15. Brad, F., Iacob, R., Hosu, I., & Rebedea, T. (2017). Dataset for a neural natural language interface for databases (NNLIDB). arXiv preprint arXiv:1707.03172.