IJNRD.ORG ISSN : 2456-4184
INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD) | IJNRD.ORG
An International Open Access, Peer-reviewed, Refereed Journal

QueryPilot: Using NLP and Generative AI for Simplifying Database Queries

Minal Khandare
Assistant professor
Dept. of Computer Science and Engineering
HKBK College of Engineering
Bangalore, India
minalk.cs@hkbk.edu.in

Ashwin Dhungana
Dept. of Computer Science and Engineering
HKBK College of Engineering
Bangalore, India
1HK21CS027@hkbk.edu.in

Atheeq Ur Rehman
Dept. of Computer Science and Engineering
HKBK College of Engineering
Bangalore, India
1HK21CS029@hkbk.edu.in

Junaid Mehraj
Dept. of Computer Science and Engineering
HKBK College of Engineering
Bangalore, India
1HK21CS059@hkbk.edu.in

Mohammed Rayan S
Dept. of Computer Science and Engineering
HKBK College of Engineering
Bangalore, India
1HK21CS097@hkbk.edu.in

***Abstract*—**
Converting natural language queries into SQL statements, QueryPilot is a cutting-edge AI-powered solution that simplifies database interactions. By making it possible for users to enter queries in plain English and effectively retrieve data, this initiative seeks to close the gap between non-technical users and sophisticated database systems. QueryPilot improves productivity, lowers mistakes, and saves time when formulating queries by reducing the requirement for SQL expertise. Its broad range of applications covers a number of industries, such as e-commerce, healthcare, education, corporate analytics, and customer service. QueryPilot has a lot to offer in terms of making data retrieval easy to use and accessible, even with difficulties like guaranteeing translation correctness and managing intricate queries. This application democratizes access to insightful data by enabling individuals to make data-driven decisions with ease.

***Index Terms*—Keywords:**
Natural Language Processing (NLP), SQL automation, database interaction, AI-powered query, data accessibility, non-technical user-friendly, productivity boost, query translation accuracy, data-driven decision-making, industry applications.

I. Introduction

A. QueryPilot

Efficient access and analysis of data is essential in today's data-driven environment. However, SQL is frequently difficult for non-technical users, which makes it difficult to interface with databases effectively. In order to overcome this difficulty, QueryPilot uses artificial intelligence (AI) to convert natural language queries into SQL commands, facilitating simple and easy data retrieval. This solution benefits a number of industries, including business intelligence, healthcare, education, and more, by increasing productivity, decreasing mistakes, and saving time. QueryPilot democratizes access to important data by streamlining database interactions, freeing people to concentrate on insights and decision-making.

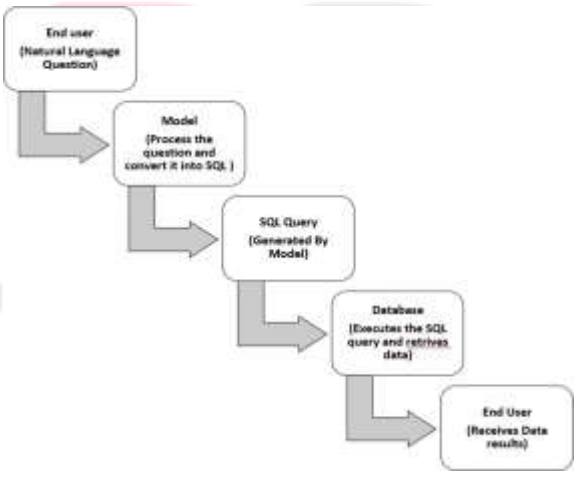

Fig. 1. A flowchart showing how AI translates requests in plain language into SQL queries for convenient database access.

This flowchart makes database access easier for non-technical users by showing how to utilize machine learning to translate a natural language query into SQL. A thorough explanation of each element is provided below:

End User (Natural Language Question): Although the end user may lack technical knowledge of databases or SQL, the process begins with them. They enter their question in natural language or simple English, such ”What are the major cities in the state of Kansas?” The AI model starts with this natural language query and deduces the user’s intention.

Processes and transforms the query into SQL using the model: The model is an AI-powered element that has been taught to comprehend and interpret natural language. It transforms the user’s inquiry into a SQL query by interpreting its meaning. Understanding the question’s structure, identifying entities (such as ”state” or ”cities”), and converting them into SQL syntax that may query the database are all necessary for this conversion. This stage helps to close the gap between technical database operations and non-technical inquiries.

SQL Query (Generated by Model): Following processing, the model uses the natural language input to generate a SQL query. for example-

```
SELECT CITY_NAME
FROM CITY
WHERE STATE_NAME = 'Maharashtra'
AND POPULATION > 1000000;
```

```
CITY_NAME
----------
Mumbai
Pune
Nagpur
Nashik
```

The output listed above This SQL statement may now target specific data in response to the user’s query and interact with the database.

Database (Executes SQL Query and Retrieves Data): The model generates a SQL query, which is then submitted to the database and performed to get pertinent data. The SQL query instructs the database to get the information that matches the parameters of the query, while the database itself holds structured data. For example, it locates Kansas cities with populations over a specific threshold. The database’s result set is then prepared for user return.

End User (Receives Data Results): Lastly, the end user is shown the data in an understandable way. Anyone may easily obtain insights and make data-driven decisions since the user can view the results of their query without requiring SQL knowledge.

B. Datasets

In order to train models that can convert natural language queries into structured SQL queries, a number of datasets have been created for a text-to-SQL machine learning project. Every dataset has distinct qualities and may be used with varying degrees of domain specialization and query complexity. A few well-known datasets for text-to-SQL model training and evaluation are reviewed in this section.

Spider (Yu et al., 2018), a large-scale benchmark created to assess the generalization capabilities of text-to-SQL models across several databases, is one of the most popular datasets. With queries spanning several domains and using intricate SQL procedures like joins, aggregations, and subqueries, Spider offers more than 10,000 examples.This dataset is a great tool for training models that must do cross-database generalization since it is particularly made to assess the model’s capacity to handle both single-table and multi-table queries.

Another well-liked resource, especially for easier, domain-specific tasks, is the **ATIS (Airline Travel Information System)** dataset (Hemphill et al., 1990). Around 5,000 sample queries pertaining to airline and travel-related data, including flight schedules, ticket costs, and aircraft statuses, make up ATIS. The dataset may be used to train models that concentrate on certain domains, particularly in the context of reservation or customer service systems, because each question is paired with its matching SQL query.

To train single-table database-focused models, the **WikiSQL dataset** (Zhong et al., 2017) provides more than 80,000 queries and SQL statement examples.This dataset is commonly used to train models that can convert natural language queries into SQL for situations when just one table is involved. The dataset is useful for building models that manage fundamental SQL operations since it contains a wide range of query types, from straightforward selection queries to more intricate filtering and aggregation activities.

Another important tool for creating SQL queries is the **SQLNet dataset** (Guo et al., 2018). SQLNet’s more than 5,000 examples assist models in concentrating on producing SQL queries in a more methodical and gradual way. With a focus on procedures like SELECT, WHERE, and JOIN, the dataset offers plain language queries in addition to their equivalent SQL queries. To help train models to produce SQL queries step-by-step, SQLNet’s architecture focuses on decomposing the SQL query generation activity into smaller components, such choosing columns and defining filters.

By adding multi-turn conversations, the **CoSQL dataset** (Yu et al., 2019) expands the WikiSQL dataset to handle conversational SQL creation. CoSQL, which has more than 10,000 instances, is made to train models that must manage context-dependent SQL queries, where the query’s meaning might vary depending on previous exchanges in a conversation. Systems that need to create SQL queries in response to back-and-forth discussion, like virtual assistants or customer care bots, are perfect candidates for this dataset.

A knowledge graph that pulls structured data from Wikipedia articles serves as the foundation for the **DBpedia dataset** (Auer et al., 2007). When training models to produce SQL queries that communicate with huge, structured knowledge bases, DBpedia is very helpful. This dataset is appropriate for applications that need to query large, structured knowledge libraries, such question-answering systems, since it includes relationships between entities and enables models to

produce SQL queries that traverse these links.

Furthermore, by adding more intricate and varied query formats, the **ATIS v2 dataset** (Liu et al., 2016) expands upon the original ATIS dataset. Because it adds more query variants, it is a superior choice for training models that must manage a greater variety of phrase forms and query types in the airline industry.

Finally, a dataset for comparing text-to-SQL models is the **TSQL Benchmark** (Iyer et al., 2017). It gives models a wide variety of questions to test their ability to produce SQL statements in various scenarios. Because it includes both simple and complicated queries, the TSQL Benchmark is a valuable tool for assessing the scalability and resilience of models in practical applications.

C. Methods

Models of Sequence-to-Sequence (Seq2Seq) An SQL query is converted from plain language using a Sequence-to-Sequence (Seq2Seq) model. Two components make up the model: an encoder and a decoder. The decoder creates the SQL query once the encoder processes the input, which is your query.

Example: If the input is: ”What are the names of employees in the Sales department?” The model might generate this SQL query:

```
SELECT name FROM employees
WHERE department = 'Sales';
```

Pointer Systems The computer may ”point” to certain terms in the input query with the use of pointer networks, a form of Seq2Seq model. This is helpful for copying certain items (such as the names of tables or columns) straight into the SQL query.

For instance, of the question ”Which employees have worked for more than 5 years?” The SQL may be generated by the pointer network immediately identifying ”employees” and ”5 years” from the natural language query:

```
SELECT name FROM employees
WHERE years_of_service > 5;
```

Transformers Transformers are strong models that focus on several input components simultaneously using attention techniques. Transformers like as T5 or BERT excel at deciphering complicated queries and producing SQL from them.

For instance, if the input is ”Show the total sales by each department for the year 2020,” The SQL query might be produced via a transformer model:

```
SELECT department, SUM(sales)
FROM sales_data
WHERE year = 2020
GROUP BY department;
```

Syntax-Aware Models These models provide particular consideration to the syntax or structure of SQL queries. Because they are aware of SQL-specific criteria, such which clauses belong together (e.g., SELECT, FROM, WHERE), they assist the model in producing more correct SQL.

An example of input might be ”Find all the employees in the HR department.”

```
SELECT name FROM employees
WHERE department = 'HR';
```

Parsing Semantions Instead of only focusing on structure, semantic parsing aims to comprehend the meaning of the natural language query and convert it into SQL based on that meaning.

As an illustration, Answer the following question: ”What is the total revenue for the last quarter?”

```
SELECT SUM(revenue) FROM sales
WHERE quarter = 'last';
```

Reinforcement Education With this method, the model gets feedback on the SQL queries it produces, which helps it learn. Gradually, the model grows better by receiving a ”reward” when the SQL query is right and a ”penalty” when it is incorrect.

As an illustration, a model produces a query:

```
SELECT * FROM sales
WHERE revenue = 'last quarter';
```

SQL-Neural A deep learning method called Neural-SQL was created especially for SQL creation. It functions by concentrating on SQL-specific tasks and producing queries one at a time.

For instance, enter ”How many products were sold in 2023?”

```
SELECT COUNT(*) FROM products
WHERE year = 2023;
```

Models of Pre-trained Language Large volumes of text data are used to train pre-trained language models, such as GPT (Generative Pre-trained Transformer), which may then be optimized to produce SQL queries. These models are excellent because they can produce SQL queries with little training on certain datasets and comprehend a broad variety of queries.

Input: ”List all employees who joined after 2015.” is an example.

```
SELECT name FROM employees
WHERE join_date > '2015-01-01';
```

D. Data Augmentation

Encode Type The techniques used to convert text in natural language into a machine-readable format are referred to

Term	Example
Encode Type	”Show employees from HR.”
Graph-based	”Employees” and ”HR” connected by ”work in.”
Self-attention	”Employees” and ”HR” are linked.
Adapt PLM	Fine-tuning BERT for SQL tasks.
Pre-training	Training GPT-3 on diverse text sources.

TABLE I
TEXT-TO-SQL WITH MACHINE LEARNING (EXAMPLES ONLY)

as encode type. This includes technologies such embedding Word2Vec, GloVe, or FastText, which turn words into numerical vectors that capture semantic content, and tokenization, which divides the text into smaller units like words or sub-words. By using complex encoding techniques to create contextual embeddings, transformer-based models like BERT or GPT are able to comprehend relationships throughout the whole input sequence, which enhances performance for applications like Text-to-SQL.

Graph-based methods use graph structures to represent the relationships between objects, including database tables and their columns. Text-to-SQL allows for the representation of things such as ”employees,” ”HR,” or ”department” as nodes in a network, with edges signifying their connections. By taking into account the links between various entities and characteristics within the dataset, the model is able to provide more accurate SQL queries by better comprehending the structure of the database schema. Learning representations of these graphs is frequently accomplished using methods such as Graph Neural Networks (GNNs).

self-attention as a technique to consider various aspects of an input sequence while generating predictions. This implies that the model may concentrate on pertinent input elements, such as particular entities or actions, in order to produce the appropriate SQL query for text-to-SQL jobs. In the statement ”Show employees from HR,” for example, the model would concentrate on the terms ”employees” and ”HR,” since these are essential to building the query. Self-attention increases the precision of SQL creation by capturing contextual information and long-range dependencies.

Adapting pre-trained language models, such as BERT, GPT, or T5, entails optimizing a model that has already been trained on a sizable corpus of textual data. Focusing on a particular goal, like Text-to-SQL, allows the model to adjust its expertise to the task. As a result, the model can more successfully translate natural language inquiries into SQL queries. A plethora of information on language structure, syntax, and semantics is introduced by pre-trained models, which may then be improved for particular use cases, such as the creation of SQL queries.

Pre-training A language model is first trained on a sizable, varied dataset during the pre-training phase, during which it gains an understanding of language structure, grammar, and word connections. Masked language modeling and causal language modeling are two pre-training tasks that aid in the model’s comprehension of context and its ability to anticipate future or missing words. Following pre-training, the model may be improved to produce SQL queries from natural language inputs by fine-tuning it on task-specific datasets, such as Text-to-SQL. Because of pre-training, the model may use broad language patterns and modify them for particular uses.

II. TEXT-TO-SQL CONVERSION

A. *Install the necessary software and configure your API key.*

Installing the ”google-generativeai” package is the first step in setting up a text-to-SQL conversion model using Google Generative AI. The terminal command ”pip install google-generativeai” may be used to do this. To authenticate your connection with the Google service after the package has been installed, you will need to set up an API key. You will configure this key in your code by executing ”genai.configure(api key=”YOUR API KEY”)”. You may get this key from the Google Cloud Console.

```
pip install google-generativeai

import google.generativeai as genai

# Set the API key
genai.configure(api_key="YOUR_API_KEY")
```

B. *Set up the model’s parameters.*

After that, you put up the model’s parameters, which determine how the AI creates the SQL queries. There are several configuration possibilities for the ”geminipro” model. The model may produce lengthier replies (about 3000–4000 characters) if you specify ‘max_output_tokens=4096‘, and you can set ‘temperature=0.4‘ to guarantee that the generated responses are more concentrated and predictable. With safety settings like ‘safety_settings=”standard”‘, the model is guaranteed not to produce offensive or dangerous material.

```
# Configuration of model settings
model = genai.Model(name="gemini-pro",
temperature=0.4, max_output_tokens=4096,
safety_settings="standard")

model = genai.Model(name="gemini-pro",
temperature=0.3, max_output_tokens=4096,
safety_settings="high")
```

C. *To obtain a SQL query, ask a question.*

After the model is set up, you may ask a straightforward query like ”What are the names and ages of users over the age of 30?” This information will be used by the model to produce the relevant SQL query. ‘SELECT name, age FROM users WHERE age ¿ 30;‘ is an example of the produced SQL that enables you to work directly with your database.

```
# Ask a question
nl_question = "What are the names and ages of
users over the age of 30?"

# Generate the SQL query
response =
model.generate_text(prompt=nl_question)

# Output the generated SQL query
print("Generated SQL Query:",response.text)
```

D. Resolve Complicated Inquiries and Make Adjustments for More Particular Needs

More complicated SQL queries can also be handled using this procedure. For instance, the model may provide a query such as 'SELECT products.product_name, SUM(sales.amount) AS total_sales FROM sales if you ask, "What is the total sales from the sales table for each product, joined with the products table?" GROUP BY products.product_name;'JOIN products ON sales.product_id = products.product_id. To fit your unique requirements, you can modify the query or change the parameters.

```
SELECT products.product_name,
SUM(sales.amount) AS total_sales
FROM sales
JOIN products ON
sales.product_id = products.product_id
GROUP BY products.product_name;
```

These techniques make it simple to turn natural language inquiries into SQL queries, which will help users who might not know how to write SQL by hand. This configuration makes database interactions more efficient and makes database queries simpler for non-technical users.

```
# Ask a complex question about joining
tables
nl_question = "What is the total sales
from the sales table for each product,
joined with the products table?"

# Generate the SQL query
response =
model.generate_text(prompt=nl_question)

# Output the generated SQL query

print("Generated SQL Query:", response.text)

# Ask a more specific question with an
additional condition
nl_question = "What are the names and
ages of users over the age of 30 and
living in New York?"
esponse =
  # Generate the SQL query
model.generate_text(prompt=nl_question)

# Output the generated SQL query
print("Generated SQL Query:", response.text)

--------------------------------------------------
SELECT name, age FROM users
WHERE age > 30 AND city = 'New York';
```

III. Model Instance

"genai" is the line.The 'generativeModel' class is created using the 'google-generativeai' library using the following code: GenerativeModel(model_name = "gemini-pro", generation_config = generation_config, safety_settings = safety_settings)'. By initializing the '"gemini-pro"' model, it is possible to produce text-based answers to input questions. To modify the model's behavior, the 'generation_config' comprises parameters like 'temperature', which regulates output randomness, and 'max_output_tokens', which restricts output length. Through the use of filters that stop risky or destructive replies, the 'safety_settings' make sure the created material is suitable. These parameters work together to build up the model instance with certain output generation and safety settings, enabling it to handle queries and generate safe, regulated outputs.

```
model = genai.GenerativeModel(model_name =
"gemini-pro",
    generation_config = generation_config,
        safety_settings = safety_settings)
```

A. SQL Query Executor

By running SQL queries and publishing the results, the Python function read_sql_query(query) enables communication with a PostgreSQL database. To make sure there isn't a connection open at first, the method starts by setting the connection variable to None. After that, it attempts to connect to the PostgreSQL database using the psycopg2.connect() function, where the required login information is supplied, including the host, database name, user, and password. Following the establishment of the connection, the method generates a cursor object that functions as an interface for database interaction. The method uses cursor.execute(query) to perform the query that was handed in, and cursor.fetchall() to obtain the query's results, which yields a list of every row that the query was able to retrieve. The user is then presented with the query results in an easily accessible way when the function iterates over these rows and outputs each one. The function is encapsulated in a try-except block to provide robust error handling. This block detects any potential exceptions, such as problems with the database connection or query execution. For debugging purposes, the unless block outputs an error message in the event that an error occurs. Whether the query

is successful or not, the finally block is used to ensure that resources are released correctly by closing the cursor and the database connection. By managing failures and making sure that the query is properly cleaned up once it has been run, this method is helpful for rapidly accessing and retrieving data from a PostgreSQL database.

```
import psycopg2

def read_sql_query(query):
    connection = None
    try:
        # Establishing the connection
        connection = psycopg2.connect(
            host="host_address",
            database="database_name",
            user="user",
            password="postgres_pwd"
          )
        cursor = connection.cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        for row in rows:
print(row)
except (Exception, psycopg2.Error)
as error:
print("Something went wrong...", error)
  finally:
 # Closing database connection
if connection is not None:
    cursor.close()
    connection.close()
```

B. Define prompt.

In order to start the process, the question ”What is the most sold product?” is defined and stored in the variable question. The AI model’s primary response will be to this query. The creation of a list named prompt_parts is the following stage. Two sections make up this list: In order to assist the model better grasp the query, the first portion, prompt_parts_1[0], probably contains some background knowledge or pre-existing context. The real question comes in the second section. Both of these can be used to provide the model enough data to produce an accurate and significant response. The code creates the prompt_parts list and then uses the method model.generate_content(prompt_parts) to submit this list to the AI model. The query and context are sent to the model for processing by this function. Following input analysis, the model produces a response depending on the question and the provided context. The variable response is where the produced answer is kept. Use response.text to view the model’s response. This just takes the text of the response, which will give the answer to the inquiry, including information about the best-selling goods. The model produces a more precise and considered response when context and a particular query are combined. With this method, the AI can comprehend the query more easily and, using the input given, produce a clear and helpful response.

```
question ="what is the most sold product?"

prompt_parts = [prompt_parts_1[0],
question]
response =
model.generate_content(prompt_parts)
response.text
```

C. Put things together in Function.

Using a query and some background, the function generate_gemini_response is intended to automate the process of producing an AI answer. It requires two input_prompt parameters (the background or extra information that might assist the model develop a more accurate response) and question parameters (the precise question you want to ask). The code initially merges these two parameters into a list named prompt_parts within the method. The question itself (question) and the background information (input_prompt) are both included in this list. Model.generate_content() is used to provide the list prompt_parts to the AI model once it has been formed. The variable response contains the answer that the model produces after processing the combined input. Depending on the query and circumstances, the AI will provide some text in this answer.

The response.text, which contains the actual substance of the AI’s response, is extracted by the code once it has generated the response and used as input to another method called read_sql_query(). This function probably queries a database for information relevant to the topic and delivers data depending on the AI’s response. The generate_gemini_response function then returns the database query’s results.

When generate_gemini_response(”How many products are there?”, prompt_parts_1[0]) is called, for instance, the function generates an answer using the query ”How many products are there?” and a bit of pre-existing context (prompt_parts_1[0]). After processing this input, the model will query the database and provide pertinent information on the quantity of items in the system.This function automates the process of creating queries and obtaining data based on natural language inquiries, making it simpler to communicate with a database and the AI model. Context is used to make sure the model’s response is precise and pertinent.

```
def generate_gemini_response(question,
input_prompt):
    prompt_parts = [input_prompt, question]
    response =
    model.generate_content(prompt_parts)
    output = read_sql_query(response.text)
    return output

    generate_gemini_response
    ("How many products are there?",
            prompt_parts_1[0])
```

IV. Conclusion

In conclusion, by using AI-powered natural language processing to streamline the querying process, QueryPilot represents a revolutionary change in the way businesses engage with data. It enables users to ask queries in plain English and get correct answers, making data retrieval accessible to everyone, regardless of technical proficiency. Everyone, from business analysts to customer support representatives, can confidently make data-driven choices because to this user-friendliness, which helps close the gap between technical and non-technical users.

QueryPilot's extensive potential is demonstrated by its use in sectors such as business analytics, e-commerce, healthcare, education, and customer service. Businesses can depend on the integrity of their data insights since it not only increases productivity but also drastically lowers mistakes by cutting down on the time spent creating and debugging complicated SQL queries. QueryPilot is a priceless tool for businesses with a variety of data requirements because of its capacity to manage both straightforward and intricate queries.

Additionally, QueryPilot tackles a number of issues related to conventional database querying, including maintaining translation accuracy and handling complex queries. Users may obtain the data they want without having to comprehend the underlying technological aspects thanks to its sophisticated capabilities, which guarantee that even complicated requests are correctly processed and carried out. People at all organizational levels may now access and use data without relying on IT departments or data scientists, paving the way for more inclusive decision-making.

QueryPilot enables consumers to make well-informed decisions more quickly and effectively by democratizing data access. It enables businesses to take use of the full potential of their data while lowering their need on technological know-how. In the end, QueryPilot is revolutionary in its ability to make data analytics more useful, efficient, and accessible, assisting companies in remaining inventive, competitive, and responsive in the data-driven world of today. article

References

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

[2] Singh, G., Solanki, A. (2016). An algorithm to transform natural language into sql queries for relational databases. Selforganizology, 3(3), 100-116. Sripad, Joshi, and Laxmaiah E. n.d. 2013. Survey of Natural Language Interface to Databases.

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

[4] Vig, Jesse, and Kalai Ramea. aˆ€œComparison of transfer-learning approaches for response selection in multi-turn conversations.aˆ€ Workshop on DSTC7. 2019.

[5] Yu, Tao, et al. a‰ŒSyntaxsqlnet: Syntax tree networks for the complex and cross-domain text-to-SQL task.aˆ€ arXiv preprint arXiv:1810.05237 (2018).

[6] Sun, Zeyu, et al. aˆ‰ŒA grammar-based structural CNN decoder for code generation.a‰ŒC Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 33. 2019.

[7] Finegan-Dollak, Catherine, et al. aˆ€œImproving text-to-SQL evaluation methodology.a‰ŒC arXiv preprint arXiv:1806.09029 (2018).

[8] Yu, Tao, et al. a‰ŒSpider: A large-scale human-labeled dataset for complex and cross domain semantic parsing and text-to-SQL task.aˆ€ arXiv preprint arXiv:1809.08887 (2018).

[9] Hwang, Wonseok, et al. aˆ‰ŒA comprehensive exploration on wikisql with table-aware word contextualization.aˆ‰ŒC arXiv preprint arXiv:1902.01069 (2019).

[10] Lin, Kevin, et al. aˆ€œGrammar-based neural text-to-SQL generation.aˆ€ arXiv preprint arXiv:1905.13326 (2019).

[11] Maas, Andrew, et al. aˆ‰ŒLearning word vectors for sentiment analysis.a‰ŒC Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies. 2011.

[12] Xu, Xiaojun, Chang Liu, and Dawn Song. aˆ€œSqlnet: Generating structured queries from natural language without reinforcement learning.aˆ€ arXiv preprint arXiv:1711.04436 (2017).

[13] Gardner, Matt, et al. aˆ‰ŒAllennlp: A deep semantic natural language processing platform.a‰ŒC arXiv preprint arXiv:1803.07640 (2018).

[14] Affolter, Katrin, Kurt Stockinger, and Abraham Bernstein. aˆ€œA Comparative Survey of Recent Natural Language Interfaces for Databases.aˆ€ The VLDB Journal 28.5 (2019): 793aˆ€“819. Crossref. Web.

[15] Sujatha, B., Raju, S. V. (2016). Natural Language Query Processing for Relational Database using EFFCN Algorithm. International Journal of Computer Sciences and Engineering, 4, 49-53.

[16] Sukthankar, N., Maharnawar, S., Deshmukh, P., Haribhakta, Y., Kamble, V. (2017). nQuery-A Natural Language Statement to SQL Query Generator. In Proceedings of ACL 2017, Student Research Workshop (pp. 17-23).

[17] Stefan W., Ellen R., Gabriele S., (1996). Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing, Springer.

[18] T. Ono, H. Hishigaki, A. Tanigami, T. Takagi, (2001), Automated extraction of information on proteinprotein interactions from the biological literature, Bioinformatics. doi:10.1093/bioinformatics/17.2.155.

[19] Warren, D. H., Pereira, F. C. (1982). An efficient easily adaptable system for interpreting natural language queries. Computational Linguistics, 8(3-4), 110-122.

[20] Woods, William A, Ronald M Kaplan, and Bonnie Nash-Webber. (1972) The lunar sciences natural language information system. Bolt, Beranek and Newman, Incorporated.

[21] Xu, X., Liu, C., Song, D. (2017). Sqlnet: Generating structured queries from natural language without reinforcement learning. arXiv preprint arXiv:1711.04436.

[22] ” Yossi Shani, Tal Cohen, and Yossi Vainshtein. (2016) ””Natural Language Interface for ” ”Databases.”” KUERI.ME. 2016. http://kueri.me/. ”

[23] Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T. (2017). Sqlizer: Query synthesis from natural language. Proceedings of the ACM on Programming Languages, 1(OOPSLA), 63.

[24] Zhong, V., Xiong, C., Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv preprint arXiv:1709.00103.

[25] Lin, K., Bogin, B., Neumann, M., Berant, J., Gardner, M. (2019). Grammar-based neural text-to-sql generation. arXiv preprint arXiv:1905.13326.

[26] Zhang, Rui, et al. aˆ‰ŒEditing-Based SQL Query Generation for Cross-Domain Context Dependent Questions.aˆ‰ŒC arXiv preprint arXiv:1909.00786 (2019).

[27] Wang, Bailin, et al. aˆ‰ŒRat-SQL: Relation-aware schema encoding and linking for textto SQL parsers.aˆ€ arXiv preprint arXiv:1911.04942 (2019).

[28] Dar, Hafsa Shareef, et al. aˆ‰ŒFrameworks for Querying Databases Using Natural Language: A Literature Review.aˆ‰ŒC arXiv preprint arXiv:1909.01822 (2019).

[29] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder–decoder for statistical machine translation,” in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1724–1734.

[30] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” arXiv preprint arXiv:1412.3555, 2014.

[31] Z. Yan, J. Ma, Y. Zhang, and J. Shen, "Sql generation via machine reading comprehension," in Proceedings of the 28th International Con ference on Computational Linguistics, 2020, pp. 350–356.

[32] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," 2018. [Online]. Available: https://openreview.net/forum?id=Syx6bz-Ab

[33] L. Dong and M. Lapata, "Language to logical form with neural atten tion," in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 33–43.

[34] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards complex text-to-sql in cross-domain database with intermedi ate representation," in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019, pp. 4524–4535.

[35] Z. Yao, Y. Su, H. Sun, and W.-t. Yih, "Model-based interactive se mantic parsing: A unified framework and a text-to-sql case study," in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), 2019, pp. 5447–5458.

[36] X. V. Lin, R. Socher, and C. Xiong, "Bridging textual and tabular data for cross-domain text-to-sql semantic parsing," in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 4870–4888.

[37] R. Shin, "Encoding database schemas with relation-aware self-attention for text-to-sql parsers," 2019.

[38] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2017, pp. 440–450.

[39] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," Advances in neural information processing systems, vol. 28, 2015.

[40] K. Xuan, Y. Wang, Y. Wang, Z. Wen, and Y. Dong, "Sead: End-to end text-to-sql generation with schema-aware denoising," arXiv preprint arXiv:2105.07911, 2021.

[41] B. Wang, W. Yin, X. V. Lin, and C. Xiong, "Learning to synthesize data for semantic parsing," in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2760–2766.

[42] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine trans-lation models with monolingual data," in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin, Germany: Association for Computational Linguis-tics, 2016, pp. 86–96. [Online]. Available: https://aclanthology.org/P16-1009

[43] Q. Liu, D. Yang, J. Zhang, J. Guo, B. Zhou, and J.-G. Lou, "Awakening latent grounding from pretrained language models for semantic parsing," in Findings of the Association for Computational Linguistics: ACL IJCNLP 2021, 2021, pp. 1174–1189.

[44] A. Kumar, P. Howlader, R. Garcia, D. Weiskopf, and K. Mueller, "Challenges in interpretability of neural networks for eye movement data," in ACM Symposium on Eye Tracking Research and Applications, 2020, pp. 1–5.

[45] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev, "Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task," in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018, pp. 1653–1663.

[46] C. Wang, K. Tatwawadi, M. Brockschmidt, P.-S. Huang, Y. Mao, O. Polozov, and R. Singh, "Robust text-to-sql generation with execution guided decoding," arXiv preprint arXiv:1807.03100, 2018.

[47] C. Wang, M. Brockschmidt, and R. Singh, "Pointing out sql queries from text."

[48] T. Guo and H. Gao, "Content enhanced bert-based text-to-sql genera-tion," 2019. [Online]. Available: https://arxiv.org/abs/1910.07179

[49] T. Yu, C.-S. Wu, X. V. Lin, B. Wang, Y. C. Tan, X. Yang, D. R. Radev, R. Socher, and C. Xiong, "Grappa: Grammar-augmented pre-training for table semantic parsing," in ICLR, 2021.

[50] Y. Zhang, X. Dong, S. Chang, T. Yu, P. Shi, and R. Zhang, "Did you ask a good question? a cross-domain question intention classificatin benchmark for text-to-sql," arXiv preprint arXiv:2010.12634, 2020. JOURNAL OF LATEX CLASS FILES, VOL. 14, NO. 8, AUGUST 2021 18

[51] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[52] D. Choi, M. C. Shin, E. Kim, and D. R. Shin, "Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross domain databases," Computational Linguistics, vol. 47, no. 2, pp. 309 332, 2021.

[53] X. Zhang, F. Yin, G. Ma, B. Ge, and W. Xiao, "F-sql: fuse table schema and table content for single-table text2sql generation," IEEE Access, vol. 8, pp. 136409–136420, 2020