**Text

Description automatically generated**

Artificial Intelligence

BTECH: 2022-23SEMESTER VII

**Conversion of Natural Language to SQL query**

Guide: **Prof. Preeti Kale**

**Group Members:**

**PB02 Charul Jain** (**1032190044@mitwpu.edu.in**)

**PB04 Hrishikesh Vaze** (**hrishikesh.vaze01@gmail.com**)

**PB16 Atharva Wagh (a3wagh@gmail.com)**

**PB19 Zeeshan Mujawar** (**zmujawar2@gmail.com**)

**PB20 Chaitanya Nirfarake (chaitunirfarake@gmail.com)**

INDEX

|  |  |
| --- | --- |
| **SR NO.** | **TITLE** |
| 1. | Abstract |
| 2. | Introduction |
| 3. | Literature Review |
| 4. | Database |
| 5. | Proposed Methodology |
| 5.1 | Algorithm |
| 6 | Result |
| 7 | Comparison to base paper |
| 8 | Conclusion |
| 9 | References |
| 10 | First Page of Base Paper |

**1. ABSTRACT**

This paper outlines a method for efficiently automating the conversion of Natural Language Query to SQL Queries. A strong tool for organizing data is structured query language (SQL), a relational database management system. The correct SQL Query must be entered to obtain or manipulate data. However, people who are unfamiliar with SQL are unable to access the necessary data. We put forward NLSQL, a system in natural language processing to get around this problem and convert natural language queries to SQL queries. This enables users who are unfamiliar with SQL to access the necessary content. Complex queries can also be handled by this system. The system is dynamic and can work with any database. Apart from formerly explored work, we propose state-of-the-art parsing, up to three times, and implementation for JOIN queries with our system’s support for multiple tables.

Keywords: Automation, Natural Language Query, SQL, Query, Dynamic, Database

**2. INTRODUCTION**

A branch of artificial intelligence called natural language processing is used to create intelligent machines that can converse with people in human-like ways. It eliminates the gap between humans and machines. The primary goal of natural language query processing is to enable computer interpretation of English statements. Despite all the difficulties, it is still commonly used for research purposes. By asking questions in natural language and receiving the necessary answers, natural language processing can be utilized to access databases. Natural language queries are a very convenient and simple way to access databases, especially for users who are unfamiliar with complex database-handling query languages like SQL.

This system is designed for anyone and everyone regardless of knowing SQL. The System proposes the architecture for processing the English Query fired to get SQL query using input as the text entered by the users.

**3. LITERATURE REVIEW**

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev worked on the paper ‘Improving Text-to-SQL Evaluation Methodology ’ that evaluates 7 existing datasets and text to SQL systems with the proposition of a new query based dataset split. [1]

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, And Dragomir Radev proposed ‘TypeSQL’ in ‘TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation’ an approach that views Natural language to SQL 4 as a slot filling task and is tested on the ‘WikiSQL’ dataset. But it does not cover some important SQL operators such as JOIN and GROUP BY. [2]

Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee put forward the paper ‘Natural language to SQL: Where are we today?’ that asserts testing current advancements in NLP-to-SQL by a survey of existing methods and performing extensive experiments under their framework using eleven of recent techniques over 10+ benchmarks. Even though there are no consistent and clear superior methods overall benchmarks. [3]

Christopher Baik, H. V. Jagadish, and Yunyao Li put forward the ‘TEMPLAR’ system for ‘ Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases’ that can be used to augment existing NLIDBs which leverages information from the SQL query log of a database to enhance their performance. [4]

Abhilasha Kate, Satish Kamble, Aishwarya Bodkhe, and Mrunal Joshi worked on the paper ‘Conversion of Natural Language Query to SQL Query ’ that talks about the approach to automate the conversion of Natural Language Query to SQL Queries effectively. This model can deal with complex queries. Their implementation might not have covered JOIN statements. [5]

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, And Thomas Dillig, the paper ‘SQLizer: Query Synthesis from Natural Language ’ proposes a new technique for automatically synthesizing SQL queries from natural language. The system boasts a vast Dataset, Hundreds of queries evaluated. But the system might return an empty table when the user's natural language doesn't match the same database or table queries are returned to be selected from. [6]

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz’s ‘Towards a Theory of Natural Language Interfaces to Databases ’ work in this paper is focusing on transforming natural language into SQL queries using natural language processing for relational databases semantic matching technique used to create an intelligent interface that converts natural language to SQL queries using a set of production rules and a data dictionary. [7]

Sarabjit Kaur and Rashmeet Singh Bali worked on ‘SQL Generation And Execution From Natural Language Processing ’ and proposed a system that starts with pre-processing the entered natural language query and then converts the data set into a Python data frame. Then the data is fetched based on a Natural query using the TAPAS module and TensorFlow.The user can visualize their data in a way that will make the results of their query even clearer, Python modules are used for the visualization. [8]

Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake and Qiaofu Zhang’s ‘Natural SQL: Making SQL Easier to Infer from Natural Language. Specifications propose an SQL intermediate representation (IR) called Natural SQL (NatSQL). Specifically, NatSQL preserves the core functionalities of SQL. Experimental results on the challenging Spider benchmark demonstrate that NatSQL consistently improves the prediction performance of several neural network architectures. [9]

Banerjee, Trishali & Bhattacharjee, Upasana & Jansi, K. worked on “Natural Language Querying and Visualization System ''. In today’s world, everything is data-driven but not everyone has the technical knowledge required to build queries and be familiar with various Python tools available for data visualization. The research intends on combining the two systems, an NLP interface to get data from simple English language queries and a second system where the data can be visualized using natural language processing. [10]

Lin, Kevin & Bogin, Ben & Neumann, Mark & Berant, Jonathan & Gardner, Matt’s ‘Grammar-based Neural Text-to-SQL Generation’ which works on making techniques to handle grammar-based decoding problems has shown significant improvement for other semantic parsing tasks. The model made by them which includes identifier linking, link embeddings, and type constraints yielded a 4.5% improvement in denotation accuracy over prior work on the context-dependent ATIS dataset. [11]

Gan, Yujian & Chen, Xinyun & Xie, Jinxia & Purver, Matthew & Woodward, John & Drake, John & Zhang, Qiaofu worked on “Natural SQL: Making SQL Easier to Infer from Natural Language Specifications” which presents the use of Natural SQL (NatSQL), a new intermediate representation that brings a simplified query over other alternatives while preserving a high coverage of the SQL structure which is easier to infer from a natural language representation than a full-fledged SQL. [12]

Simon Erikson worked on “Comparing Natural Language Processing to Structured Query Language Algorithms” in which he compared two of the current most advanced open-source algorithms (SyntaxSQL and TypeSQL). He proposed architecture and test environment for finding the performance of NLP models by testing them on the Spider dataset which is a large NLP2SQL dataset, and it primarily aims at teaching and testing understanding of advanced SQL. [13]

**4. DATA**

For our application, we use our own database of sale of properties. The following are the tables in the database:

I. Homes - This table has the following attributes: id (primary key), owner\_id, address, and rooms column.

II. Owners - The owners table has the following attributes: id (primary key), name, email, phone, and age.

III. Sales - The sales table is made of the following attributes: id (primary key), home\_id, and price.

**5. METHODOLOGY**

User will enter credentials for his database and will start the program.

Our program will only need the following inputs:

1. Database credentials (along with its respective database driver present in the program itself)
2. Natural Query (human input)

For now our program only supports SELECT queries. So users can ask questions like *“show …”, “display …”, “count …”, or “which …”, etc*. To convert natural text given by the user to a proper SQL query, steps used in our program can be classified into mainly four type steps:

A. Tokenization

Here user's input will be converted into a list of tokens. This token list will be fed to lexical analysis.

B. Lexical Analysis

Here tokens will be parsed with the help of a keyword dictionary, and database information. The output of this will be a list of tuples wherein the first element will be the parsed token content and the second element will be the token type.

C. Syntactic Analysis

This step will reorder the tokens and will also filter out unwanted tokens according to the SQL grammar.

D. Generation of SQL

Here parsed final tokens will be passed to the NLSQL database driver where it will be converted to the final SQL query.

Following is the keyword dictionary we have used in our program:

{

"select":{

"SELECT": ["SELECT", "FETCH", "GET", "SHOW", "LIST", "display", "which"]

},

"from":{

"FROM": ["FROM", "IN", "OF"]

},

"selector":{

"\*": [ "ALL"]

},

"where":{

"WHERE": ["WHERE", "IF", "WHEN", "IF", "WHEN", "WHOSE", "WITH"],

},

"keyword" : {

"ORDER BY": ["ORDER BY", "SORT BY", "ORDER", "SORT"],

"NOT": ["NOT", "NEITHER", "NONE OF"],

"LIMIT": ["LIMIT", "TOP", "FIRST"],

"OFFSET": ["OFFSET", "SKIP", "AFTER"],

"GROUP BY": ["GROUP BY", "GROUP"],

"HAVING": ["HAVING"],

"DISTINCT": ["DISTINCT", "UNIQUE"]

},

"option":{

"DESC": ["DESC", "DESCENDING", "DECREASING", "DECREASE"],

"ASC": ["ASC", "ASCENDING", "INCREASING", "INCREASE"]

},

"function":{

"AVG": ["AVG", "AVERAGE", "MEAN"],

"COUNT": ["COUNT", "TOTAL"],

"MAX": ["MAX", "MAXIMUM", "HIGHEST"],

"MIN": ["MIN", "MINIMUM", "LOWEST"],

"SUM": ["SUM", "TOTAL"]

}, "conjuction":{

"AND": ["AND", "BOTH", "TOGETHER"],

"OR": ["OR", "EITHER", "ONE OF"],

},

"operator":{

"LIKE": ["LIKE", "AS"],

"IN": ["IN", "WITHIN"],

"IS NULL": ["IS NULL", "IS EMPTY", "IS BLANK"],

"IS NOT NULL": ["IS NOT NULL", "IS NOT EMPTY", "IS NOT BLANK"],

"BETWEEN": ["BETWEEN", "IN BETWEEN", "WITHIN"],

"=": ["=", "EQUAL TO", "IS"],

">": [">", "GREATER"],

"<": ["<", "LESS"],

">=": [">=", "GREATER THAN EQUAL TO"],

"<=": ["<=", "LESS THAN EQUAL TO"],

"!=": ["!=", "NOT EQUAL TO"],

"\*": ["\*", "MULTIPLY"],

}

}

We cannot use it directly, so we will convert it to this type of dictionary:

{

'select': ('select', 'select'),

'fetch': ('select', 'select'),

'from': ('from', 'from'),

. . .

}

**5.1. ALGORITHM**

**Diagram

Description automatically generated**

**Fig. 1** Flow of our application.

To make a clear understanding of the algorithm we will consider the following query as an example:

“*show homes with sale price greater than 15000*”

Step 1: Fetching the database data

As soon as the program starts, the database driver will retrieve all of the information, including the list of tables, columns in each table along with its pseudo data types, foreign keys, and primary keys.

Here is an example of fetched data:

{

"tables": ["homes", "owners", "sales"],

"columns": {

"homes": [

["id", "number"], ["owner\_id", "number"],

["address", "string"], ["rooms", "number"]

],

"owners": [

["id", "number"], ["NAME", "string"], ["age", "number"],

["phone", "number"], ["email", "string"]

],

"sales": [["id", "number"], ["home\_id", "number"], ["price", "number"]]},

"foreign\_keys": {

"homes": {

"owner\_id": "owners"

},

"owners": {},

"sales": {

"home\_id": "homes"

}

},

"primary\_keys": {

"homes": "id",

"owners": "id",

"sales": "id"

}

}

|  |  |
| --- | --- |
| Original Datatypes | Token Types |
| int, float, double, decimal | number |
| date, DateTime, timestamp | date |
| *rest all datatypes* | string (value of some column) |

Pseudo Data types (used in our program)

Step 2: Setting up the NLSQL engine

Selecting all tables from a database in every SQL query even if it is not needed will create the performance issue. Here we will identify the tables needed in our query and will select them dynamically. This selection will be carried out in *step 5*. So we will initialize the selected\_table dictionary as follows:

selected\_tables\_dict = {

'sales': 0,

'homes': 0,

'owners': 0,

}

Step 3: Taking user Input and cleaning of the text

Now, the program will continuously accept and process user input. The first step is to clean up the text. Some words that are not useful in forming SQL will be ignored here. The following are some of the words used in ignore list in our program: *the, a, are, is, to, in, than*

After cleaning:

“*show homes with sale price greater 15000*”

Step 4: Tokenization

After cleaning the text next essential step is tokenization. Here our text is divided into the list of tokens for upcoming processing.

After tokenization:

Tokens = *['show', 'homes', 'with', 'sale', 'price', 'greater', '15000']*

Step 5: Parsing of tokens

It will mainly consist of 3 steps:

1. Formatting: we use a regular expression to allow only digits, \_, and characters in tokens. Next, we convert all characters to lowercase.

Regular Expression used: *[^0-9a-zA-Z\_]*

1. Lemmatization: It is an essential part of natural language processing. This process reduces the token into its root form ensuring the root form is valid according to the language.
2. Text Enrichment: Here, as the name suggests we will try to enrich the natural text. The program will try to find tokens that are almost similar to the keywords, table names, or column names and will convert them to the exact word.

We will perform parsing three times. Parsing will search the current tokens in the following order:

1. Keywords
2. Table name
3. Column name *for tables which are selected*

So, suppose the column name comes before the table name in the natural query. The parser will not recognize it. So, whenever we are not able to find them in the above order, we leave the token as it is and continue to parse the next tokens. So, the next time we parse the unparsed tokens from the token list, we will be having the tables selected and so the program will easily identify the column if it is present in the natural query. Suppose we don't recognize the token in the following order even twice than in the third parse, we will consider it as a value to any column in the selected tables which will be handled by the SQL query generator module in the future. This process is called *Text enrichment*.

After parsing of tokens:

Tokens = *[('select', 'select'), ('homes', 'table'), ('where', 'where'), ('sales', 'table'), ('sales.price', 'column'), ('>', 'operator'), ('15000', 'number')]*

Selected Tables = *{'homes': 1, 'owners': 0, 'sales': 1}*

Step 6: Reordering and filtering tokens

Reordering

This step is important. Let's consider a query - “*show names of students in descending order*”. Here if we see the word “*descending*” will be mapped to the SQL keyword “*DESC*” and the word “*order*” will be mapped to the SQL keyword “*ORDER BY*”. If we see the sequence of these tokens then “*DESC*” comes before “*ORDER BY*” which is in reverse order according to SQL syntax. So the program will reorder these types of tokens if they are present in the token list.

Filtering

SQL has rules like *table name* should not occur after “*where*” or “*group by*” keywords. *Selectors (‘\*’, etc)* should not occur after the “FROM” keyword. The program will filter these tokens out.

After Reordering and Filtering:

Tokens = *[('select', 'select'), ('homes', 'table'), ('where', 'where'), ('sales.price', 'column'), ('>', 'operator'), ('15000', 'number')]*

Step 7: Generating Query

So the tokens have been properly parsed, filtered, and reordered based on the syntax of the selected database. The only process left is the creation of a SQL query. Because the output of this step will differ depending on the database, this will be handled by a custom-made database driver.Table

Description automatically generated

**Fig. 2** Driver is an abstract class

**Joining Selected Tables**

Limitation of the program is that it only supports “INNER JOINS”. Until now we have selected two tables, i.e. sales, and homes. Let us see how the program converts them to join statement string. From the database information fetched before the first step, we will use information about foreign keys and primary keys.

Input = *[‘sales’, ‘homes’]*

Intermediate result:

Joining\_dictionary = *{ 'sales': [ ('homes', 'sales.home\_id') ] }*

Here we can clearly see that we have sales joined with one table of homes with key sales.home\_id as foreign key referencing the primary key “*id*” of the referenced table “*homes*”.

And final join phrase:

“ *sales INNER JOIN homes ON sales.home\_id = homes.id*”

Step 8: Executing the query

In addition, after executing the generated query on the database, our program will also show the result.

NLSQL Class:

This class will handle full NLSQL conversion. It will need one database driver.

nlsql\_engine = NLSQL(db\_driver=db\_driver)

**6. RESULTS**

We implemented the program NLSQL. Here are some of the results. We have executed many queries, here are some of the queries.

Query 1) “display all homes”

Text

Description automatically generated

Query 2) “show names of all owners with age greater than 40”

Text

Description automatically generated

Query 3) “show names of all owners with age greater than 40 and less than 50”

Text

Description automatically generated

Query 4) “count number of homes”

Text

Description automatically generated

Query 5) “show addresses of homes with sale price greater than 15000 and rooms greater than 3”

Text

Description automatically generated

**7. COMPARISON TO BASE PAPER**

The algorithms can generally be classified as either syntax- or semantics-based. The most well-known and traditional NLP2SQL technique, for instance, is known as keyword detection. The main concept is to build a SQL query using a big corpus that can categorize which words are significant in a sentence. This approach belongs to the syntax-centered group because it only examines individual words and ignores the sentence's overall meaning. The algorithm will normally employ machine learning to comprehend the semantics of the question. Therefore, we use a common metric offered by the Spider dataset to compare such models. Where the Spider dataset is a large NLP to SQL dataset with the primary goal of teaching and testing understanding of advanced SQL.

The metrics used are Component Matching, Exact Matching, and Execution Accuracy. The Component Matching metric checks a model’s performance on different SQL components. Where each component is a SQL Keyword, like *SELECT*, *WHERE*, and *GROUP BY*. Each component is divided into sets of sub-components which the algorithm then checks if it matches the expected sets of sub-component. Each component is evaluated independently and has slightly different requirements. For example, some components have order constraints while others do not. The exact matching metric checks if every part of the SQL is correct. This occurs when all the components in the query match expectations. The Execution accuracy checks if the generated SQL produces the correct results. It is important to check alongside the other metrics as there are often multiple different SQL queries that produce the correct answer. However, sometimes it can be misleading as incorrect queries could also produce the correct result. For example, when a query is supposed to return NULL.

Spider dataset also divides the questions into four categories: easy, medium, hard and extra hard. The difficulty classification is based on the number of SQL components, selections, and conditions.

|  |
| --- |
| Diagram  Description automatically generated |

**Fig. 3** Architecture to find the performance of an NLP2SQL Model

The results would be in the format of

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DB | Category | Count | Accuracy | Parse Time |
| Spider dataset | Easy |  |  |  |
| Medium |
| Hard |
| Extra Hard |

**8. CONCLUSION AND FUTURE WORK**

We have proposed a new methodology for synthesizing SQL queries from natural language. Starting with initial program tokens generated using lexical parsing, our approach enters an iterative process of parsing the tokens, and reorders and filters the unwanted tokens. Our method also uses dynamic table selection, where the program will only select those tables which it finds appropriate to include in the query.

The proposed methodology has been implemented as a tool called NLSQL, which will provide an end-to-end solution for generating SQL queries from natural language. Unlike other tools, this program will automatically retrieve the necessary data from the database. Users will be required to provide database credentials. The program will also run the query and display the results.

Our solution will only work with number and string data types and is not yet optimized for dates and therefore can be a prospective work hereafter. This system can be better visualized using a GUI.

**9. REFERENCES**

[1] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. [Improving Text-to-SQL Evaluation Methodology](https://aclanthology.org/P18-1033). In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

[2] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. [TypeSQL: Knowledge-Based Type-Aware Neural Text-to-SQL Generation](https://aclanthology.org/N18-2093). In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), pages 588–594, New Orleans, Louisiana. Association for Computational Linguistics.

[3] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2021. Natural language to SQL: where are we today? Proc. VLDB Endow. 13, 10 (June 2020), 1737–1750. https://doi.org/10.14778/3401960.3401970

[4] C. Baik, H. V. Jagadish and Y. Li, "Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases," 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 374-385, doi: 10.1109/ICDE.2019.00041.

[5] A. Kate, S. Kamble, A. Bodkhe, and M. Joshi, "Conversion of Natural Language Query to SQL Query," 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), 2018, pp. 488-491, DOI: 10.1109/ICECA.2018.8474639.

[6] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. Proc. ACM Program. Lang. 1, OOPSLA, Article 63 (October 2017), 26 pages. <https://doi.org/10.1145/3133887>

[7] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In Proceedings of the 8th international conference on Intelligent user interfaces (IUI '03). Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/604045.604070

[8] Kaur, Saravjeet and R. S. Bali. “SQL GENERATION AND EXECUTION FROM NATURAL LANGUAGE PROCESSING.” (2012).

[9] Singh, G., & Solanki, A. (2016). An algorithm to transform natural language into SQL queries for relational databases. Selforganizology, 3, 110-126.

[10] Banerjee, Trishali & Bhattacharjee, Upasana & Jansi, K.. (2021). Natural Language Querying and Visualization System. 10.3233/APC210251.

[11] Lin, Kevin & Bogin, Ben & Neumann, Mark & Berant, Jonathan & Gardner, Matt. (2019). Grammar-based Neural Text-to-SQL Generation.

[12] Gan, Yujian & Chen, Xinyun & Xie, Jinxia & Purver, Matthew & Woodward, John & Drake, John & Zhang, Qiaofu. (2021). Natural SQL: Making SQL Easier to Infer from Natural Language Specifications.

[13] Eriksson, Simon. "COMPARING NATURAL LANGUAGE PROCESSING TO STRUCTURED QUERY LANGUAGE ALGORITHMS." (2019).

**10. FIRST PAGE OF BASE PAPER**

|  |
| --- |
| Text  Description automatically generated |