QPE: A SYSTEM FOR DECONSTRUCTING SQL QUERIES

by

Connor D. Bullard

May, 2023

Director of Thesis: Dr. Venkat Gudivada

Major Department: Computer Science

Research on the topic of converting natural language to machine-readable code has experienced great interest over the last decade, however studies into converting machine-readable code into natural language are sparse. The applications of translating spoken or written languages into code are well-established, such as allowing a more novice or non-technical user to interact with a program or database with ease. The benefits of such applications are readily observable and are likely to grow as software systems continue to increase in complexity and capability. Likewise, parsing code to natural language produces certain benefits from which the potential gain in utility and knowledge has yet to be fully realized. This thesis identifies opportunities for deploying solutions that provide a natural language explanation of programming languages, specifically with Structured Query Language (SQL) and database interfacing. A novel solution is proposed in the form of an application named Query Purpose Extractor (QPE), which utilizes existing open-source libraries to aid in the process of translating SQL statements into English sentences.

QPE: A SYSTEM FOR DECONSTRUCTING SQL QUERIES

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

by

Connor D. Bullard

May, 2023

QPE: A SYSTEM FOR DECONSTRUCTING SQL QUERIES

by

Connor D. Bullard

APPROVED BY:

DIRECTOR OF THESIS:	Dr. Venkat Gudivada
COMMITTEE MEMBER:	Dr. Nic Herndon
COMMITTEE MEMBER:	Dr. Rui Wu
CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:	Dr. Venkat Gudivada
INTERIM DEAN OF THE GRADUATE SCHOOL:	Dr. Kathleen T. Cox

DEDICATION

To my wife, Stephanie: without you none of this would have been possible. Thank you for your steadfast support and unwavering faith in me to pursue this opportunity. You are my treasure.

To my children, Clara and August: you give me endless joy and drive me to be the best version of myself. Thank you for bringing me to heights of life I had never imagined.

ACKNOWLEDGMENTS

I would like to thank all the staff of ECU's Department of Computer Science for the exceptional education and support I have received in pursuit of my degree. In particular, to Dr. Gudivada: thank you for your guidance and encouragement as I sought to complete this research. To Drs. Herndon and Wu: I am sincerely grateful for your dedication to my success as a student in your courses and for your input throughout this research.

Additionally, I would like to thank Dr. Hills for encouraging me to apply to the Master of Science in Computer Science program in 2020. Without his considerate advice I would not have had the courage to seek enrollment in the first place.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

In the existing literature exploring the relationship between Structured Query Language (SQL) statements and natural language, most research has been focused on the transforming of natural language input into a SQL output. Little research has been conducted on the opposite direction in this relationship: transforming a SQL statement into natural language. It appears that there are two obvious motivators that have led to research being focused in its current manner:

1. The value of parsing spoken or written input from a user into SQL statements, which in turn leads to
2. Interfacing with a relational database becoming more approachable for non-technical users.

At first glance, moving to study in the opposite direction may appear trivial; an individual capable of constructing a SQL statement would inherently have some understanding of what the statement accomplishes as well as the data with which it interacts. Therefore, if a tool existed that could deconstruct a SQL statement into a natural language output, such a technical individual would at face value have no need for it.

But what about the non-technical user? There exists a population of such individuals who are exposed through their professions or recreationally to relational databases and who could stand to benefit from the existence of such a tool [31]. These individuals are present in many areas of academia and enterprise organizations–for example, a business analyst at a large company may work on a daily basis with software developers and data scientists who make use of SQL statements in their work. The business analyst may have limited firsthand experience with the querying language and therefore may be unable to make sense of SQL statements when they are presented. A solution developed to provide a mechanism to parse meaning from SQL statements would prove useful in this situation, as a deep understanding of SQL may be outside the scope of the business analyst's position, but surely, he or she would benefit from knowledge of the SQL statements to which they are exposed, and as a result the business or organization would grow in cohesiveness. A similar benefit could be observed for a solution that could also parse meaning from NoSQL statements into natural language text. For the purpose of this study, ANSI SQL will be the language of focus, but it is worth acknowledging that a similar solution could be beneficial in other querying languages.

1.2 Contribution

The research focuses on the following key objectives:

1. Analysis of existing literature concerning conversion between SQL and natural language, specifically English.

2. Identification of current research and solutions for extracting and representing the abstract syntax tree of a SQL statement.

3. Development of a proof-of-concept application which demonstrates the feasibility of deconstructing SQL statements into natural language using parse trees.

In the existing research between SQL and natural language, a common theme is the utilization of machine learning models and natural language processing to obtain an accurate, repeatable and reliable output, be it into SQL or English. When the research focuses on converting SQL to natural language, processing the SQL statement directly via machine learning models is a common approach. The research in this thesis aims to propose a viable method that does not involve the use of machine learning and highlights the tenability of extracting a parse tree from which to derive a meaningful description. An application, Query Purpose Extractor (QPE), has been developed in conjunction with this research to demonstrate this approach as feasible. It is available for use and testing in the form of a web application[1].

[1]Currently accessible at `https://query-pe.dev`

1.3 Thesis Outline

The thesis is organized into the following chapters: chapter 2 comprises a background of the subject matter along with a review of current literature on the relationship between natural language and SQL statements. Beginning in chapter 3 possible approaches to the problem of this paper are discussed, and in chapter 4 the novel approach Query Purpose Extractor (QPE), which utilizes available open-source libraries to create a parsing tool, is reviewed. Chapter 5 covers a discussion of the strengths and weaknesses of the application, and implications and future works are covered in chapter 6.

Chapter 2

SQL Statements and Natural Language

A bidirectional relationship exists between SQL statements and natural language: it is possible to traverse from one state to the other in a method of translation. Traveling from natural language to SQL statements typically requires the use of machine learning and natural language processing in order to convert the input, whether spoken or typed, into accurate and effective SQL statements. Research with this focus typically aims to produce the most accurate SQL statement representing the intentions of the end user which can then be used to interact with a modern database management system. Traveling from SQL statements to natural language can be done by several approaches. A robust, repeatable, and precise method involves extracting the syntax tree from the SQL statement and assigning meaningful identification or phrases to relevant data points in the tree. In either direction of the travel between the two, the goal is to educate end users on better interacting with database management systems and enabling them to become more familiar with SQL and querying.

For the purposes of this research, the issue of converting SQL into natural language will be referred to as the SQL-to-NL problem.

2.1 Related Works

2.1.1 Translating Natural Language to SQL

The notion of interfacing with a database system using simple English (or other spoken or written language) commands has existed for many years. In fact, the concept is not limited solely to database interactions but is an area of interest for general-purpose programming languages [8] [30]. Programming languages and their abstractions arose due to the precise requirements a computer demands, creating a divide between the non-technical user and granular computational control. Attempts to bridge this divide date back even to the same decade as the creation of SQL [6]: in their 1979 study, Chang and Ke [7] described an extensible query language (XQL) to handle "fuzzy" queries from users. Designed to allow a less formal approach when extracting data from databases, the research outlined an "intelligent coupler" tasked with manipulating the user's requests into a format the database system can execute. The authors highlighted their choice of the word "coupler" as intentional, as it implies a connection between the user and the system. The language and the study are early examples of a practical endeavor that is abundantly observed today.

Annamaa et al. [2] proposed a novel approach to validating SQL statement syntax prior to execution in a Java environment. They identified the problem of writing embedded SQL queries that could not be tested until the program had been executed and the query executed, as until this point it is operating within the Java runtime as opposed to a database management system. The proposed SQL syntax analyzer utilizes the Java Development Tools from the Eclipse IDE[1] to obtain abstract syntax trees from the source code and is equipped to determine any issues prior to query execution. This study identifies validation as a valuable outcome from observing and monitoring the abstract syntax trees of SQL statements.

Elgohary et al. [10] identified a potential shortcoming of semantic parsing systems concerning the correction of interpretation when an utterance is incorrectly mapped to a value.

[1]`https://www.eclipse.org/ide/`

Their study intentionally provided an incorrect or inconclusive argument (speech or utterance from a user) that initially is interpreted incorrectly and translated into a wrong SQL statement. It is then corrected by additional speech in the form of feedback to the model, which in turn generates the corrected SQL statement as a result. In this paper the authors identify the usefulness of user interaction in the validation step of converting natural language to SQL, in particular highlighting the benefit of explaining the SQL statements in a way users can understand. Part of their study involved the step of deconstructing the model-generated SQL statements into short paraphrases in English for explaining the generated output the model created after interpreting the utterances of the user.

Kumar et al. [17] provide another example of an investigation into converting spoken English into SQL for interfacing with a relational database management system. Utilizing acoustic and language models, their work encodes spoken commands into tokenizable data using inspiration from the Hidden Markov Model concept. This concept assists in accounting for imperfections in the phonetics observed from the raw input caused by the natural variations among English speakers. In this paper the process taking spoken input to machine-executable code is broken down into each individual step, providing a deeper insight into the effort and complexity of such a task.

In a similar manner, Timbadia [24] devises a system that accepts the input natural language in the format of a text (.txt) file. The operation of the proposed application is similar in function to previously-described natural language to SQL solutions, but of note in this study is the inclusion of a database scanning step. This step analyzes the tables, columns, and keys present in the database to ensure the validity of translated SQL statements for the purposes of the attached database.

Yaghmazadeh et al. [28] also sought to develop a tool for parsing natural language to SQL, however their approach took both the user-created request as well as the underlying database schema into consideration as input to ensure a more accurate and executable output using a system named SQLizer.

Guo and Gao [13] developed a similar approach to translating English to SQL using a dataset called WikiSQL [32], which is described as "a dataset of 80,654 hand-annotated examples of questions and SQL queries distributed across 24,241 tables from Wikipedia."[2] This dataset has been widely studied and used in various natural language processing initiatives, and in this study, the authors generate a SQL statement from a given prompt or context to accurately query a table that has been previously identified. A similar study also referencing WikiSQL was conducted by Bai et al. [4] which utilized a compound point-wise reward to provide better feedback to the learning model used for the purpose of generating SQL from natural language. The current best-performing model using the WikiSQL dataset is TaPEx (Table Pre-Training via Execution), a pre-training approach used to mimic executable SQL query processes in order to develop a deep understanding of the database schema created by Liu et al. [19].

Brunner and Stockinger [5] identify potential areas for improvement in natural language to SQL (NL-to-SQL) research, primarily in providing detailed system architecture and source code for the solution developed. This paper discussed the creation of the ValueNet system, designed as an end-to-end architecture which implements a neural network to encode and decode user input into abstract syntax trees for the purpose of interacting with a database. Because the system is designed to be end-to-end, candidate queries are created using database content to ensure better accuracy with each request. In a similar vein, Qin et al. [23] identify the benefits of building a database query from the database to the request as opposed to the traditional request to database direction. By starting with the knowledge base of all relations in a given database, a candidate query graph is generated which provides greater accuracy than attempting to parse meaning from the user's request and then associate it with expected table or column names.

Other studies have been conducted that extend the benefits of natural language to SQL conversion into more complex applications. Liu et al. [18] contributed to addressing the

[2]Wikisql: Datasets at Hugging Face `https://huggingface.co/datasets/wikisql`

serious threat of SQL injection attacks by developing a tool named DeepSQLi. This tool extends common SQL injection defenses such as automated testing of potential malicious user inputs by implementing a natural language model that adjusts a training dataset into novel test cases. These test cases are able to be fed back into the language model as needed to continually create more sophisticated attacks in order to address database security concerns before they are exploited by bad actors.

Cognitive computing is another application that research into this topic could positively impact. In their study on implementing artificial intelligence to a relational database, Neves and Bordawekar [20] examine the outcome of applying cognitive intelligence queries into a traditional workflow of extracting data from a SQL-based database. Cognitive intelligence queries are defined as complex queries that implement semantic matching and inductive reasoning for the purposes of business analytics. The paper highlights another area of data storage, retrieval, and management that could benefit from a multilateral approach to SQL and natural language conversion.

2.1.2 Translating SQL to Natural Language

As aforementioned, research into the parsing of SQL statements into natural language text is limited. Of the available studies, the one most similar in nature and function to this research was conducted by Paira and Chandra in 2019 [22] in which a parser (*SQL_NL*) was developed to identify SQL syntax, tokenize keywords, and output natural language. A primary difference between this study and this thesis research is the process by which the SQL statement is deconstructed. *SQL_NL* makes use of regular expressions (regex), whereas the proposed system Query Purpose Extractor sources the natural language from the syntax tree of a given SQL statement.

A study conducted by Xu et al. [27] in 2018 also approached the subject of translating SQL to natural language by the use of a graph-to-sequence model [26]. As opposed to isolating the abstract syntax tree for extraction of meaning and descriptions, a graphical representation of the query was generated to preserve its components in logical groupings. Using an approach utilizing the type of Recurrent Neural Network (RNN) Long-Short Term Memory (LSTM), the authors applied natural language processing to predict the next token given all previous words in the generated description. The researchers tested their model on two datasets, one being the aforementioned WikiSQL set with the other a dataset from the website StackOverflow. Results demonstrated the success of the novel approach which, compounded with the work of this thesis and the study by Paira and Chandra, exemplifies the validity of multiple approaches to decoding SQL statements. Unfortunately, only part of the code was published publicly at the time of this writing[3], and the model could not be implemented as a demonstration to include with this research.

At a more fundamental level, Yellin and Mueckstein [29] investigated the notion that conversion from one language into another language using a certain set of rules implies that translating in the opposite direction is possible. The study identifies a method used to invert attribute grammars and demonstrated its effectiveness on SQL queries, outputting English paraphrases of what the query was seeking to accomplish. The authors note a potential use-case for providing these paraphrases could be the feedback it provides to a novice or occasional user of relational databases, allowing the user to examine the accuracy and precision of the queries that have been created.

[3]Per this opened issue: `https://github.com/IBM/SQL-to-Text/issues/5`

Chapter 3

SQL, Parse Trees, and Abstract Syntax Trees

The most common way to interact with a relational database management system (RDBMS) is via SQL statements. SQL itself comes in many flavors in addition to its standardized ANSI[1] and ISO[2] forms, but in order for the written code to be executable by the RDBMS it must first be converted into a meaningful format.

Specific to the popular RDBMS PostgreSQL, once a user enters the SQL statement and executes it the database parser first determines if the syntax is valid. If it is valid, the parser then returns the parse tree of the query which is used to complete the request[3]. This process, the parse tree, and the abstract syntax tree will be explored in greater depth in the following section.

[1]A summary of ANSI standards for SQL can be found at: `https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/`

[2]Published ISO standards for SQL can be found here (relevant links begin with "ISO/IEC 9075"): `https://www.iso.org/committee/45342/x/catalogue/p/1/u/0/w/0/d/0`

[3]`https://www.postgresql.org/docs/current/parser-stage.html`

3.1 Background

3.1.1 Syntax Trees

An abstract syntax tree, also simply called a syntax tree, is a data structure comprising a tree representation of syntactic details and information of code or text. Because it is abstract, not all details of the code are included in the data structure, and as it is a tree it can be traversed and manipulated. A concrete syntax tree, also known as a parse tree, is the implementation of the abstract syntax tree and represents the actual input used in code execution [12]. The parse tree is a more complex rendering of the given data that includes details regarding the position and purpose of each term or expression. Because of this, the abstract syntax tree can be thought of as a minimalist implementation of the parse tree that is not concerned with the specific grammar of the given programming language [25]. Abstract and concrete syntax trees are used in a variety of programming languages, including SQL, and are important structures used by the compiler as an intermediate representation of data [1] [3] [11]. A compiler first tokenizes input source code which is then used to generate the parse tree. This in turn is sent through to be consumed by the remaining compilation steps as the compiler executes the source code [15].

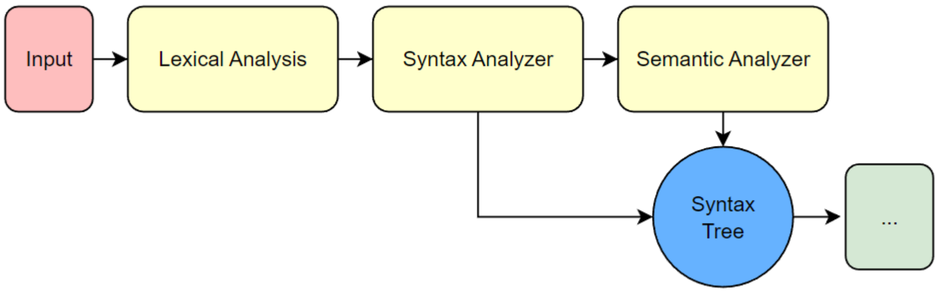


Figure 3.1: A simplified depiction of the initial stages of compilation, including generation of the parse tree.

Because the abstract syntax tree can vary considerably based on the source code provided, it offers a flexible and durable way to represent the underlying meaning behind the code. Figure 3.2 represents an example abstract syntax tree, while Figure 3.3 represents a parse tree. Both trees in these figures are generated from the below simple expression: $1 \times 2 + 3$.

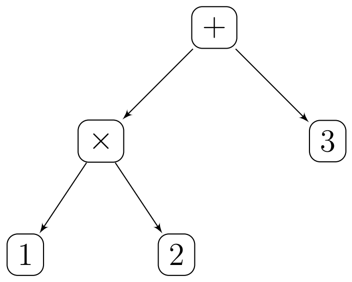


Figure 3.2: An abstract syntax tree example.

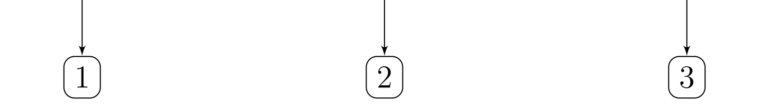


Figure 3.3: A concrete syntax tree (parse tree) example.

Slight variations in the input can cause greater changes in the parse tree generated. To demonstrate using the previous example, the expression is modified slightly to change the calculation order via addition of parentheses: $1 \times (2 + 3)$. The new abstract and concrete syntax trees can be observed in Figures 3.4 and 3.5, respectively.

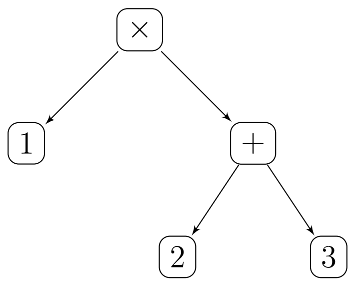


Figure 3.4: The updated abstract syntax tree.

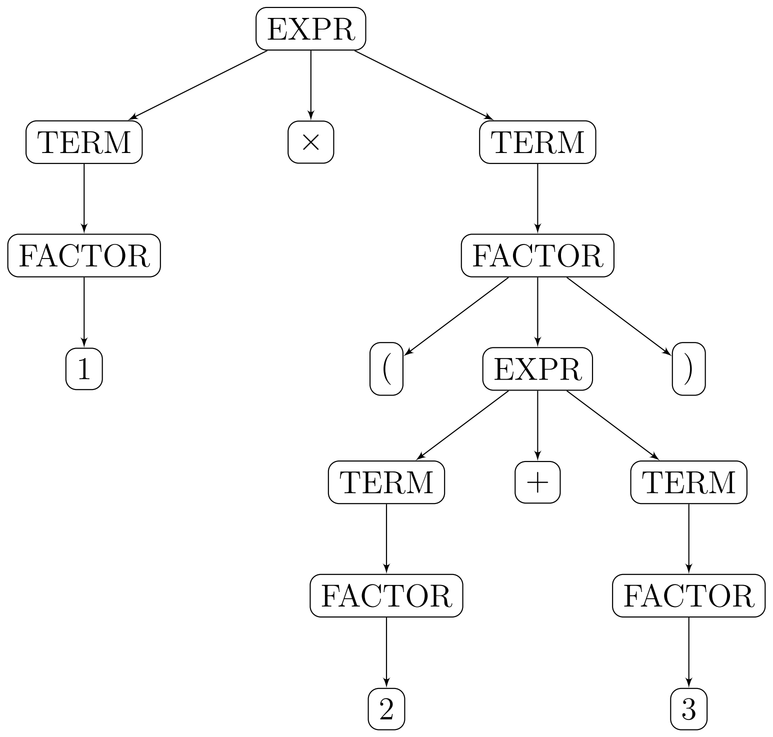


Figure 3.5: The updated concrete syntax tree (parse tree).

3.1.2 Further Considerations with Relational Algebra

The SQL language is intended for interactions with relational databases, and therefore queries written in SQL are also able to be expressed using relational algebra. Tools exist that are able to demonstrate this conversion and can help users understand the relations between the points of contact within a query[4].

Understanding this underpinning concept for SQL is essential for addressing the SQL-to-NL problem and developing a meaningful solution, as the relational algebraic representation of the query should be expressed identically in both the original SQL and the parsed output.

An example of a simple SQL query represented using relational algebra can be seen in Figure 3.6 for the below query:

```
SELECT e.name, e.title, c.salary
FROM employees e
INNER JOIN compensation c
ON c.employee_id = e.employee_id
```

$$\Pi_{\text{e.name, e.title, c.salary}}(\rho_e employees \bowtie_{\text{e.employee\_id = c.employee\_id}} \rho_c compensation)$$

Figure 3.6: Relational algebraic representation of a SQL query.

[4]An example SQL to relational algebra converter can be found at `http://www.grammaticalframework.org/qconv/qconv-a.html`

Solving the SQL-to-NL problem requires retention of the underlying logic of the relational algebra for a SQL statement. This can be conveyed accurately with the usage of the abstract syntax tree, because this relational data is exposed to the compiler or interpreter in a detailed medium.

SQL is by its very nature highly relational [9], making it a prime language for translating to English or another natural language. The tight coupling of data it references and the declarative nature of the querying language support its candidacy for being the subject of a system designed to logically derive meaning with little subjectivity. No additional context is required with SQL aside from the query at hand in order to accurately produce understandable explanations for a non-technical user to comprehend.

3.2 Methodology

3.2.1 Outline of Requirements

A systematic approach to deconstructing the syntax of the statement is required to consistently achieve the result of a parsed English translation from a SQL statement. The approach to solving the SQL-to-NL problem should have the following characteristics:

1. Repeatable: the solution should be query-agnostic and able to algorithmically derive meaning from a varied assortment of queries.

2. Accurate: the solution should accurately represent the purpose behind a given query using readily understandable paraphrases.

3. Robust: the solution should withstand varying degrees of complexity with the presented queries and responsively produce meaningful output.

For these reasons, it was determined that for this study that the best method by which to extract meaning from SQL statements involved using the parse tree of the query. This object provides the necessary data and flags for procuring meaningful results for the user. Additionally, it allows a representation of the relational algebra underpinning each SQL statement as closely as possible.

3.2.2 Demonstration of Approach

The abstract syntax tree allows for a parser to express the intentions of a SQL query in an appropriate grammar for consumption by the database optimizer. In open-source relational database management systems such as PostgreSQL, it is possible for the abstract syntax tree to be extracted from a query and represented in various data formats. Certain open-source libraries have been created to aid in this process, and will be of use for the purposes of this paper. Of particular note for is *pg_query*[5], a library developed by the pganalyze[6] team.

[5] `https://github.com/pganalyze/pg_query`

[6] `https://pganalyze.com/`

pg_query was developed as a Ruby gem[7] and allows analysis of how PostgreSQL analyzes a SQL statement for execution.

For the application developed for this study, a derivative library *pglast*[8] was used. *pglast* is built from *pg_query*, but designed for use with the Python programming language. Through the use of *pglast*, it is possible to extract an parse tree for a given SQL statement with the results output in JavaScript Object Notation (JSON)[9] format. This allows for easier consumption by the solution built for this research, and because of the structure of the parse tree, it allows for the reliable execution and consistent deconstruction required by such a task. Examples of SQL input and JSON output for the purposes of this study can be observed in Figures 3.7 and 3.8.

In the simple query in Figure 3.7, it can be observed that several flags have been specified. The *targetList* field identifies the target results of the query, namely what is present in the *SELECT* clause. As this query simply selects the integer 1, no other values are present in the *targetList* field.

The query used for Figure 3.7 is simply *SELECT 1*.

[7]`https://guides.rubygems.org/what-is-a-gem/`
[8]`https://github.com/lelit/pglast`
[9]`https://www.json.org/json-en.html`

```
"stmt": {
    "SelectStmt": {
      "targetList": [
        {
          "ResTarget": {
            "val": {
              "A_Const": {
                "val": {
                  "Integer": {
                    "ival": 1
                  }
                },
                "location": 7
              }
            },
            "location": 7
          }
        }
      ],
      "limitOption": "LIMIT_OPTION_DEFAULT",
      "op": "SETOP_NONE"
    }
  }
```

Figure 3.7: JSON representation of the parse tree of a simple SQL query.

A more involved query is provided in Figure 3.8 for demonstrating the relative quickness with which the output JSON parse tree grows in complexity. Even for a query only using *SELECT*, *FROM*, and *WHERE* clauses, the output can easily become difficult to navigate.

The query used for Figure 3.8 is:

```
SELECT b.book
FROM books b
WHERE b.year > 2000
```

To address the need for consistently extracting data as queries change, the solution builds around the expected key and value pairings to meaningfully describe the data sought by the SQL query as well as the tables, columns, and values traversed as part of this data retrieval.

```
"stmt": {
  "SelectStmt": {
    "targetList": [
      {
        "ResTarget": {
          "val": {
            "ColumnRef": {
              "fields": [
                {
                  "String": {
                    "str": "b"
                  }
                },
                {
                  "String": {
                    "str": "book"
                  }
                }
              ],
              "location": 7
            }
          },
          "location": 7
        }
      }
    ],
    "fromClause": [
      {
        "RangeVar": {
          "relname": "books",
          "inh": true,
          "relpersistence": "p",
          "alias": {
            "aliasname": "b"
          },
          "location": 19
        }
      }
    ],
    "whereClause": {
      "A_Expr": {
        "kind": "AEXPR_OP",
        "name": [
          {
            "String": {
              "str": ">"
            }
          }
        ],
        "lexpr": {
          "ColumnRef": {
            "fields": [
              {
                "String": {
                  "str": "b"
                }
              },
              {
                "String": {
                  "str": "year"
                }
              }
            ],
            "location": 33
          }
        },
        "rexpr": {
          "A_Const": {
            "val": {
              "Integer": {
                "ival": 2000
              }
            },
            "location": 42
          }
        },
        "location": 40
      }
    },
    "limitOption": "LIMIT_OPTION_DEFAULT",
    "op": "SETOP_NONE"
  }
}
```

Figure 3.8: The parse tree of a slightly more complex SQL query represented in JSON.

Chapter 4

Query Purpose Extractor (QPE)

A novel approach to completing the translation from SQL to natural language has been developed as part of this research. The system is called Query Purpose Extractor (QPE) and makes use of the *pglast* library previously described in chapter 3 to parse the abstract syntax tree from a query for further analysis. Source code and documentation for QPE can be found on GitHub[1].

The structure of QPE can be observed in Figure 4.1. The process of SQL-to-NL can be defined as having five primary steps:

1. A SQL statement is provided as the input.

2. The SQL statement is validated.

3. Parsing of the abstract syntax tree from SQL is conducted.

4. Data is extracted from the parse tree and the output is paraphrased into English.

5. The paraphrased result is returned as the output.

[1]https://github.com/cdbullard/query-pe

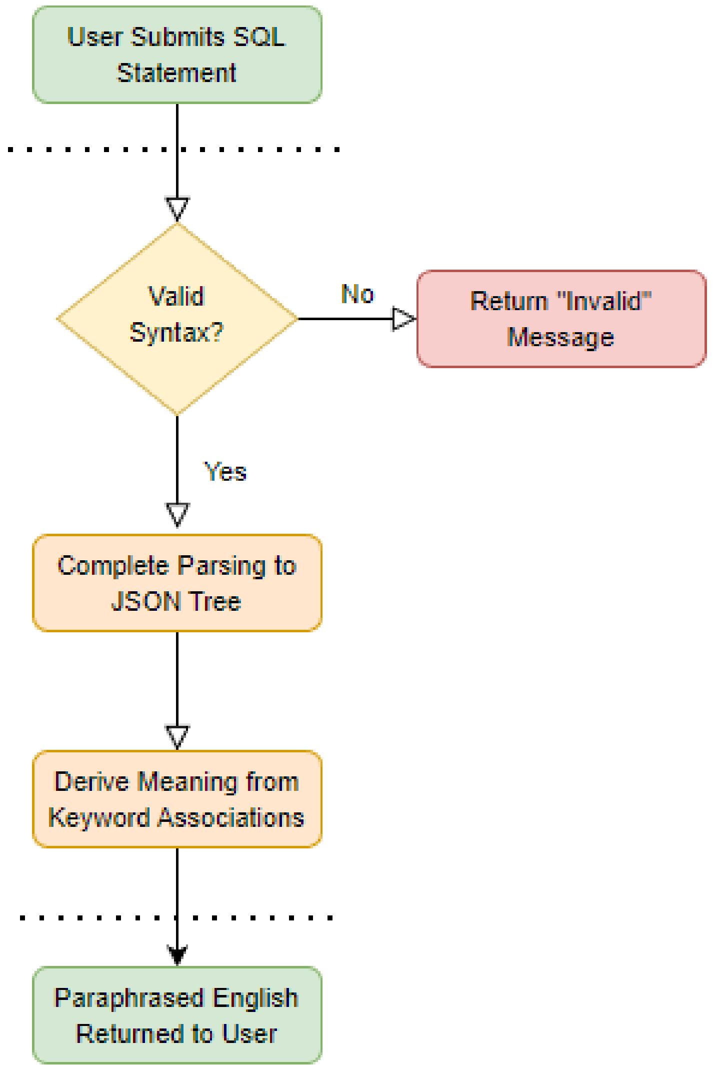

User Submits SQL Statement
Valid Syntax?
No
Return "Invalid" Message
Yes
Complete Parsing to JSON Tree
Derive Meaning from Keyword Associations
Paraphrased English Returned to User

Figure 4.1: The process of QPE.

4.1 QPE Overview

QPE is a web application developed primarily using React and Python. A user, upon navigating to the application, is greeted with a prompt seen in Figure 4.2. The prompt contains an area for inputting SQL statements along with a toggle switch to retrieve either the parsed English meaning from the statement or the JSON representation of the parse tree. The output of the JSON representation is generated from the *pglast* library similar to Figures 3.7 and 3.8. An example of the 'Generate Parsed Results' output in English paraphrases can be seen in Figure 4.3.

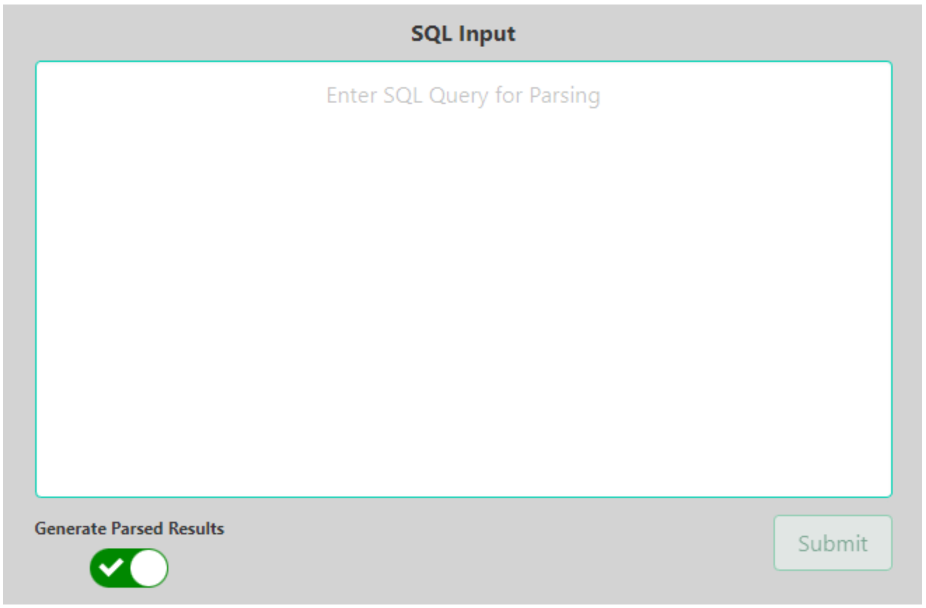
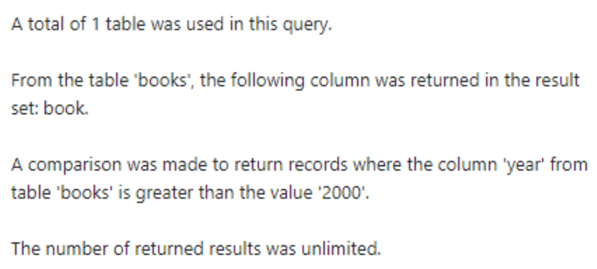


Figure 4.2: The main interface for QPE.



A total of 1 table was used in this query.

From the table 'books', the following column was returned in the result set: book.

A comparison was made to return records where the column 'year' from table 'books' is greater than the value '2000'.

The number of returned results was unlimited.

Figure 4.3: Sample results using the query in Figure 3.8.

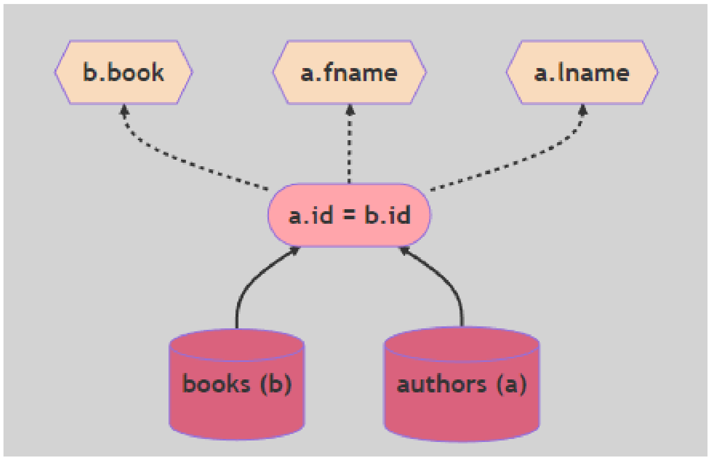


Figure 4.4: Graph depicting the output of the given sample query.

As an additional feature to more clearly demonstrate the relationship between the various clauses in the SQL statement, a graph is generated dynamically using the Mermaid.js library[2] when a query is entered that contains a table reference with linked values in the SELECT clause. Join conditions are also detected and represented by the graph, and in Figure 4.4 an example can be seen for the SQL query below:

```
SELECT b.book, a.fname, a.lname
FROM books b
INNER JOIN authors a
ON a.id = b.id
```

This graphical feature functions as an additional tool to help users understand the relationships between selected data points and any joins that may constrain them. It is designed to be a simplified version of the query's parse tree, serving as a dynamic visual aid in the process of extracting meaning from SQL statements.

[2] `https://mermaid.js.org/`

```
{
    "dict": {
        "Distinct": false,
        "From": {
            "joinvalue": [
                [
                    "=",
                    "a",
                    "id",
                    "b",
                    "id",
                    "JOIN_INNER"
                ]
            ],
            "relname": [
                [
                    "b",
                    "books"
                ],
                [
                    "a",
                    "authors"
                ]
            ]
        },
        "Group": {
            "groups": []
        },
        "Having": {
            "conditions": []
        },
        "Limit": "Unlimited results",
        "Select": {
            "targets": [
                [
                    "b",
                    "book"
                ],
                [
                    "a",
                    "fname"
                ],
                [
                    "a",
                    "lname"
                ]
            ]
        },
        "Where": {
            "conditions": []
        }
    },
    "output": "A total of 2 tables were used in this query.\n\nFrom the table 'books', the
         following column was returned in the result set: book.\nFrom the table 'authors', the
         following columns were returned in the result set: fname, lname.\n\nThe table 'authors'
         was joined with 'books' where 'id' from both tables are equal.\n\nThe number of returned
         results was unlimited."
}
```

Figure 4.5: Sample API result for the query "*SELECT b.book, a.fname, a.lname FROM books b INNER JOIN authors a ON a.id = b.id*" using the paraphrased English output.

Figure 4.5 shows a sample result body in JSON format for the API endpoint[3] in QPE responsible for parsing the given input. The API will always return two objects: the simplified dictionary ("dict") containing the extracted, relevant details from the parse tree, and the chosen response ("output"), be it the parse tree itself or the paraphrased English results.

The "dict" object in the API response body displays the intermediary information necessary for QPE to complete its task. This object is produced from traversing the extracted parse tree of the SQL query and provides a much simpler structure to be further examined in the next step of the process. It is a collection of the values associated with various SQL clauses and demonstrates the approach of QPE to map data from the SQL query in a logically-sound manner.

In the example "output" object given in Figure 4.5, the newline character "\n" can be seen in several instances. This is purely for formatting purposes when the result is displayed on the front-end of the web application.

[3]`https://qpe.onrender.com/parse`

4.2 QPE Structure

The primary purpose of QPE is to extract meaningful data from presented SQL statements and return this data to the user. The output returned is either the JSON-formatted parse tree or the natural language paraphrases and is determined by the user at the time the request is made.

When the user requests the JSON-formatted parse tree, the output generated by the QPE is sourced directly from the *pglast* library, specifically from the *parse_sql_json* method[4]. Because a form of the parse tree may be of some use to more technical users, the option to return this data was retained in the application.

When the user requests the English paraphrased results, the output is similar to the example seen in Figure 4.3. The process by which the paraphrased sentences are generated can be observed in Algorithms 4.1 and 4.2.

Using the parsed collection of clause dictionaries returned from Algorithm 4.1, the user interface renders the graph as demonstrated in Figure 4.4 and is designed to visualize the relational algebra comprising the query. Because the graph uses this collection of clause dictionaries, on each request the graph dynamically updates to reflect the structure of the new query according to three levels (from bottom to top): the relations involved, the joining conditions, and the projected columns.

[4]Documentation for this method located at `https://pglast.readthedocs.io/en/v4/parser.html#pglast.parser.parse_sql_json`

Algorithm 4.1 Process of extracting data from the parse tree

```
1: function ExtractClauseData(PT)                                  ▷ Input: JSON Parse Tree
2:     For all clauses (SELECT, FROM, WHERE, etc.) generate a dictionary
3:     for each clause in PT do
4:         Filter out unnecessary values (e.g. location, persistence, etc.)
5:         Extract appropriate data points                          ▷ Will vary by clause
6:         Assign value to appropriate clause dictionary
7:     end for
8:     Combine all clause dictionaries into a containing object
9:     return The containing object of dictionaries (Collection of Clauses)
10: end function
```

Algorithm 4.2 Process of ascribing meaningful paraphrases to parsed data

```
1: function phraseGenerator(Collection of Clauses)
2:     Identify all relations (tables) present
3:     Identify all columns or values used
4:     for each clauseDict in Collection of Clauses do
5:         Identify relations referenced and conditions applied
6:         Use a preset phrase for the appropriate type and incorporate data
7:         Append new paraphrase to an output string
8:     end for
9:     return The completed output string
10: end function
```

Chapter 5

Discussion

5.1 Outcome

QPE successfully addresses the problem identified in chapter 3.2: it provides a repeatable and accurate translation of SQL statements and serves as a proof-of-concept for the feasibility of solving the SQL-to-NL problem. The application is robust in handling complex queries, lightweight in resource management, and capitalizes on existing libraries to expose the parse tree of a SQL statement and visualize its significance in an easy-to-understand manner. Additionally, the system is extensible and can be improved over time by contributions from a global community.

A noteworthy feature unique to QPE in the studies conducted on the conversion between SQL and natural language is that this system does not utilize machine learning or natural language processing to develop its results. The functionality of QPE is established through an algorithmic approach to parsing meaning from the parse tree of SQL statements. Because of this, no training is required for extracting meaning. Because the parse tree is utilized the results are more specified than an approach using regex filtering on the original SQL statement might produce.

5.2 Limitations

Certain limitations are present at the time of writing with QPE that should be identified. It is currently possible to reliably parse and extract meaning from SQL queries containing *SELECT*, *FROM*, *WHERE*, *GROUP BY*, *HAVING*, and *LIMIT* clauses, however no method exists presently for extracting meaning from data definition language (DDL) statements. Additionally, there exists a limitation on queries using *UNION* or similar qualifiers. Development is ongoing and the system serves as a proof of concept for solving the SQL-to-NL problem. One additional complexity introduced is the requirement of aliases for the extraction of English paraphrases to be computed correctly. Due to the fact that this application is not linked to any existing database and does not store queries entered by the user, it has no method by which to determine which columns specified in the *SELECT* clause relate to a given table in the *FROM* clause.

```
SELECT author, name, publisher
FROM authors
INNER JOIN books
ON books.id = author.id
```

The system is unable to reliably determine on its own which columns belong to each table. Therefore, an alias requirement exists in order to correctly represent the parse tree using both the English paraphrase generation as well as the graphical representation of the query. A possible solution to this particular issue would be to join this application with an existing database, allowing table schema to be verified during query analysis. However, this is outside the scope of this current research and has not been attempted presently. Despite these limitations, it is noteworthy that such a solution to the SQL-to-NL problem has not been previously identified in literature with an open-source solution akin to QPE.

Similar shortcomings exist in solutions designed for the opposite problem of NL-to-SQL. It is common that a system designed to translate a natural language command into SQL would require some details regarding the database schema, such as table and column names.

This is in part due to the ambiguity that exists when an end user requests to generate SQL without knowing the exact table or column names, as the system has no method by which to deduce this on its own. Therefore, in both SQL-to-NL and NL-to-SQL problems, certain requirements are necessary for setting up the solution to ensure an accurate output.

Testing the results of QPE proved to be a challenge. Due to the limited existing research into NL-to-SQL and the even more limited available software or code designed to solve this problem, comparing QPE to similar applications proved to be outside the scope of this research. Attempts were made to demonstrate the readability and accuracy of the natural language paraphrases generated by QPE by using this output as input into a system designed to convert natural language to SQL. The possible usefulness of such validation would demonstrate that the paraphrased output of QPE could then be used to recreate the original SQL accurately, thus indicating a retention of the query's purpose. Due in part to a similar issue with limited available code but also with certain available solutions found online requiring paid subscriptions, this validation was unable to be carried out. It is expected by virtue of the algebraic approach to generating the paraphrased values that such validation would prove QPE's ability to parse and retain meaningful data about the presented SQL statements.

Chapter 6

Conclusion

This research has demonstrated a new approach to the topic of deconstructing SQL statements into natural language paraphrases by taking advantage of readily available open-source libraries and contributing a novel application named QPE. A significant contribution of this paper is in providing a foundational, open-source system to parse meaning out of SQL statements for the educational benefit of any potential users.

Due to the nature of most solutions to the SQL-to-NL problem either involving closed-source or otherwise publicly unavailable code, QPE offers future extensibility to any interested individuals or organizations seeking to contribute to solving this problem.

6.1 Future Work

Additional work is planned for completing QPE and extending its current functionality to provide support for parsing of more sophisticated SQL queries. Future work in this topic may involve approaching the SQL-to-NL problem from different algorithmic perspectives, perhaps by utilizing the abstract syntax tree of a SQL statement in a different way, or by devising another reproducible and reliable method of extracting meaning from the SQL statements.

As discussed in chapter 2, related studies have identified the benefit of integrating another layer to database access which can assist with the translation of natural language to SQL statements with higher degrees of accuracy. A possible future topic of study could involve a similar investigation in developing a SQL-to-NL solution that can learn from successful queries about the selected database and generate a model of potentially similar domains or data that could be related for providing a more holistic understanding of the database's domain, or of suggestions for future architectural optimizations that could be considered. Such a solution could enhance existing data or SQL scripts to provide extensibility and validation. One additional application that could be explored in future studies is the potential for increased analytic insights into frequently queried topics or groupings in a database given the converted meanings of SQL statements. By having another tool for examining how their databases are accessed, database administrators could consider restructuring tables or views for better querying performance, or to help group related data points into more closely-related entities.

Of particular interest for potential future work on the SQL-to-NL problem involves the recent captivating advances in natural language processing through GPT-4 [21], particularly with ChatGPT[1]. The technology has demonstrated enormous capabilities in providing meaningful, fluent output that coherently conveys information. Despite the astonishing results produced by the seemingly exponential growth in ability of large language models, its limitations are important to consider, specifically regarding the hallucinations that generative

[1]`https://chat.openai.com`

artificial intelligence is prone to experiencing [14] [16]. One possibility for future directions of the research presented in this thesis includes collaboration with GPT models to provide a grounding in truth. The output from QPE is currently rudimentary but does have logical ties to the parsed SQL queries which lends to its credibility–this could be a benefit to large language models, which often struggle with maintaining a foundation in truth. Extrapolating data from the parse tree could be used in conjunction with a natural language processing approach to extracting meaning from queries to improve accurate output from such models.

Additional research into SQL-to-NL solutions and use cases will likely continue the discussion on the overall relationship between programming languages and natural languages. It is hoped that this research will lead to a more welcoming experience for non-technical users of software as their professional positions become ever more closely linked with technology.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Annamaa, A., Breslav, A., Kabanov, J., and Vene, V. An interactive tool for analyzing embedded sql queries. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems* (Berlin, Heidelberg, 2010), APLAS'10, Springer-Verlag, p. 131–138.

[3] Appel, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, July 2004.

[4] Bai, Z., Wu, B., Wang, Z., Wang, B., et al. Learning to generate structured queries from natural language with indirect supervision. *Computer Speech & Language 67* (2021), 101185.

[5] Brunner, U., and Stockinger, K. Valuenet: A natural language-to-sql system that learns from database information. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), pp. 2177–2182.

[6] Chamberlin, D. D., and Boyce, R. F. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control* (1974), pp. 249–264.

[7] Chang, S.-K., and Ke, J.-S. Translation of fuzzy queries for relational database system. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-1*, 3 (1979), 281–294.

[8] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Date, C. J. *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[10] Elgohary, A., Hosseini, S., and Awadallah, A. H. Speak to your parser: Interactive text-to-sql with natural language feedback. *CoRR abs/2005.02539* (2020).

[11] Elmasri, R., and Navathe, S. *Fundamentals of Database Systems*, 6th ed. Addison-Wesley Publishing Company, USA, 2010.

[12] Grune, D., and Jacobs, C. J. H. *Parsing Techniques (Monographs in Computer Science).* Springer-Verlag, Berlin, Heidelberg, 2006.

[13] Guo, T., and Gao, H. Bidirectional attention for sql generation. *arXiv preprint arXiv:1801.00076* (2017).

[14] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. Survey of hallucination in natural language generation. *ACM Comput. Surv. 55*, 12 (2023).

[15] Jones, J. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design* (2003).

[16] Koubaa, A. Gpt-4 vs. gpt-3.5: A concise showdown, 2023.

[17] Kumar, S., Kumar, A., Mitra, P., and Sundaram, G. System and methods for converting speech to sql. *arXiv preprint arXiv:1308.3106* (2013).

[18] Liu, M., Li, K., and Chen, T. Deepsqli: Deep semantic learning for testing sql injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2020), ISSTA 2020, Association for Computing Machinery, p. 286–297.

[19] Liu, Q., Chen, B., Guo, J., Ziyadi, M., Lin, Z., Chen, W., and Lou, J.-G. Tapex: Table pre-training via learning a neural sql executor. *arXiv preprint arXiv:2107.07653* (2021).

[20] Neves, J. L., and Bordawekar, R. Demonstrating ai-enabled sql queries over relational data using a cognitive database. *Knowl. Discov. Data Min* (2018).

[21] OpenAI. Gpt-4 technical report, 2023.

[22] Paira, S., and Chandra, S. Sql_nl -a parser that converts sql query to natural language. *ICTACT Journal on Soft Computing* (01 2019), 1999–2003.

[23] Qin, K., Li, C., Pavlu, V., and Aslam, J. Improving query graph generation for complex question answering over knowledge base. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing* (Online and Punta Cana, Dominican Republic, Nov. 2021), Association for Computational Linguistics, pp. 4201–4207.

[24] Timbadia, D. Conversion of sql query to natural language. *International Research Journal of Engineering and Technology (IRJET) 8*, 2 (2021).

[25] Wile, D. S. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering* (1997), pp. 472–480.

[26] Xu, K., Wu, L., Wang, Z., Feng, Y., and Sheinin, V. Graph2seq: Graph to sequence learning with attention-based neural networks. *CoRR abs/1804.00823* (2018).

[27] Xu, K., Wu, L., Wang, Z., Yu, M., Chen, L., and Sheinin, V. Sql-to-text generation with graph-to-sequence model. *ArXiv abs/1809.05255* (2018).

[28] Yaghmazadeh, N., Wang, Y., Dillig, I., and Dillig, T. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–26.

[29] Yellin, D. M., and Mueckstein, E.-M. M. The automatic inversion of attribute grammars. *IEEE Transactions on Software Engineering SE-12*, 5 (1986), 590–599.

[30] Yin, P., and Neubig, G. A syntactic neural model for general-purpose code generation. *CoRR abs/1704.01696* (2017).

[31] Zhang, S., and Sun, Y. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), pp. 224–234.

[32] Zhong, V., Xiong, C., and Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR abs/1709.00103* (2017).