Natural Language Interface to Relational Databases

by

Ankit Bhankharia (atb5880)

Advisor

Dr. Carlos Rivero

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology

Fall 2016
Acknowledgments

A big thank you to Prof Dr. Carlos Rivero for guiding me through each phase of my project, right from selecting my capstone topic till the completion of my project. He constantly gave me advice and feedback to improve the application and helped me achieve each milestone on time. I would also like to thank Prof Joe Geigel for helping me with the project milestone presentations, poster preparation and report writing. Finally I would like to thank all my friends and all the faculty members that I worked with without whom I wouldn’t have been able to complete my degree successfully.
Abstract

Natural Language Processing (NLP) is the interaction between the computers and human languages and is being widely used in the field of computer science. It is being applied to various domains and one such domain being querying relational databases. Users can type in the query in human language and machine will convert that into a SQL query and return the results. But it has not been really successful mainly because of the complexity of queries and most of the times there is some ambiguity in understanding the language which results into an incorrect query generation. In this paper we propose a solution where the application can interact with the user before generating a query. Interaction helps in resolving all the ambiguities and increases the accuracy of the result. Once all the ambiguities are resolved, query in plain English will be converted into a SQL query using NLP. Using this application even naïve users, with minimal or no knowledge about the database schema, will be able to retrieve results from the database.
# Contents

Acknowledgments ........................................................................................................ 2  
Abstract ...................................................................................................................... 3  
1. Introduction ........................................................................................................... 5  
2. System Overview ................................................................................................... 6  
3. Implementation ...................................................................................................... 7  
   3.1 Connecting to the database .............................................................................. 7  
   3.2 Creating schema Nodes ................................................................................... 7  
   3.3 Dependency Parse Tree .................................................................................. 10  
   3.4 Candidate Mapping ......................................................................................... 13  
   3.5 Candidate Ranking and User Interaction ........................................................... 14  
   3.6 Parse Tree Structure Adjustor ......................................................................... 16  
   3.7 SQL Generation ............................................................................................... 17  
4. Results ................................................................................................................... 19  
5. Conclusion .............................................................................................................. 20  
6. Future Work .......................................................................................................... 20  
7. References .............................................................................................................. 21
1. Introduction

SQL is a widely used query language in order to interact with the database and retrieve information out of it. SQL queries tend to be really complex and it requires expert training in order to write correct queries. Training users on SQL or hiring trained users can be very expensive. Even for trained users it could be a challenging task to write queries because they need to first understand the entire database schema and understand the relationship between each table in the database.

It would be great if users could query the database in natural language instead of writing the queries and get back desired results. This is considered to be the ultimate goal of querying relational databases. There have been previous attempts in building a system called natural language interface to databases (NLIDB) which converts the plain English sentence into an SQL query. Such system became popular because it allowed naïve users to query the database in simple English. But despite of that, it has not been adopted widely mainly because of the low accuracy of the system.

Natural language processing is a very complex issue in itself. Because of the use of idioms, slangs, and technical terms, it is very difficult to extract the exact meaning out of the sentence. There is always some ambiguity in understanding what user wants. Even when humans interact with each other, there is some communication gap, then how can NLIDB be so accurate. If the system misinterprets the sentence then the SQL query generated for that will obviously be wrong. Also for naïve users there is no way to find out if the results returned by the system are correct or not. Even if the user comes to know that result is inaccurate, user will have to rephrase the sentence and try again and hope to get the right answer this time.

In real world human interaction, when there is some miscommunication, person needs to ask questions to the other person and get the doubts clarified. Similar logic can be applied to NLIDB to resolve the ambiguities. If the system does not understand a part of the sentence or have some doubts, it can be clarified by asking questions to the user.

So in this paper we suggest a new approach to NLIDB which is called Natural Language Interface to relational Databases. In this approach system will try to interact with the user by asking questions in case of ambiguity. We will use a data structure called query tree which will act as an intermediary between the English sentence and the SQL query. This system has 3 main components:

1) Transforming the natural language into a query tree.
2) Interactively tell the user how the questions is being interpreted to avoid miscommunication.
3) Translate the query tree into a SQL statement and return the result.
2. System Overview

Fig 2.1 has been referred from original paper[1].

The system has 8 major components:

i. User Interface – User interface is for the user to interact with the system. User can enter the DB details, enter the query in English and select candidate options given by the system to avoid miscommunication.

ii. Dependency Parser – Converts the English sentence into Dependency parse tree which is then used to create SQL query.

iii. Parse Tree Node Mapper – It maps each node in the Dependency parse tree with the nodes in schema graph.

iv. Interactive Communicator – It is the intermediary level for communication between the user and the application.

v. Schema Graph – It stores all the information related to the Database.

vi. Parse Tree Adjustor – After all the ambiguities are removed and correct mapping is done between the schema graph and the dependency parse tree, parse tree is adjusted accordingly.

vii. Query Tree Translator – Generates the SQL query from the updated parse tree using pre-defined rules.

viii. RDBMS – It stores the actual database. In our case it is MySQL.
3. Implementation

3.1 Connecting to the Database

User interface is provided to the user for interacting with the application. When user runs the application, a window pops up which asks for the database details like server name, port number, database name, user ID and password. Once all the details are verified connection is made to the database using JDBC driver. Connection is kept open till the application is closed by the user.

![Figure 3.1: UI to connect to the database](image)

3.2 Creating schema Nodes

Once the connection is made to the database, we need to retrieve the database schema and store it in some form. We will need all the information about the database like all the tables present, attributes in each database, primary key for each table and primary key/foreign key relationship between the tables. All this information will be useful while generating the SQL query and also during candidate mapping.

Initial approach was to store this information in the form of graph. Graph has two kinds of nodes: Relation nodes and Attribute nodes. As the names suggest, relation nodes will store the table name and attribute nodes will store all the attributes in the table. One relation node will be connected to all of its attribute nodes. There are 2 kinds of edges: Projection edge and
foreign key/primary key edge. Projection edges connect the relation node to its attribute nodes whereas foreign key/primary key edge will connect attributes from different tables that has FK/PK relationship.

Consider the following database:

![Database Schema Diagram]

The database in fig 3.2.1 has 3 tables: Patient, Doctor and Visit

Relation Patient has 6 attributes where ssn is the primary key which uniquely identifies each tuple for patient

Relation Doctor has 5 attributes where ssn is the primary key.

And relation Visit has 7 attributes where attributes scheduled and patient together forms the primary key.

Given database also has 3 foreign key/primary key relationships.

Patient.primaryDoctor is a foreign key mapped to Doctor.ssn

Visit.patient is a foreign key mapped to Patient.ssn

Visit.doctor is a foreign key mapped to Doctor.ssn
So for the above database schema, following graph will be generated, fig 3.2.2.

Patient, Doctor and Visit nodes are called relation nodes whereas all the other nodes are called attribute nodes. All the nodes that has black border are the primary keys. All the blue edges are projection edges that connects relation to its attributes whereas all the black edges are FK-PK edges that connects attribute nodes with FK/PK relationship.

The problem with above approach is that we need to store each subgraph (i.e. graph for each relation) separately because most of the time there is no connection between 2 relations i.e. (no foreign key/primary key relationship) and hence there is no edge from one relation to another. Also in order to find FK/PK relationship between 2 relations, we need to go through all
the attribute nodes and check if it connected to the attribute node in another relation which makes the search slower.

So instead of creating a graph like data structure, we created nodes for the relations only, which is a java class and we call it schema nodes. Each node will have 3 fields: List of attributes, List of primary keys and a hash map of FK/PK mapping. A key for FK/PK map will be currentTable.attribute and the value will be foreignKeyTable.foreignKeyAttribute. Schema nodes for above mentioned database is shown in fig 3.2.3. This way we will have to store 3 nodes separately but when we want to retrieve FK/PK relationship between tables, we could directly access the hash map of the relation node and get the relationship. We don’t have to go through all the attributes to find the relationship. This made the search really faster plus it is really easy to implement schema nodes compared to schema graph.

Node Patient
Attributes – {ssn, firstName, middleName, lastName, dob, primaryDoctor}
Primary key – {ssn}
Foreign Key – {primaryDoctor -> Doctor.ssn}

Node Doctor
Attributes – ssn, firstName, lastName, dob, salary
Primary key – {ssn}
Foreign Key – {null}

Node Visit
Attributes – {scheduled, patient, doctor, weight, height, room, otherNotes}
Primary key – {scheduled, doctor}
Foreign Key – {patient -> Patient.ssn, doctor -> Doctor.ssn}

Fig 3.2.3 Schema Nodes

3.3 Dependency Parse Tree

Once the connection to the database is made and database information is stored in schema nodes, user will be provided with a new window to enter the query in Simple English. After entering the query user will click on Submit button. The English sentence will then be processed by the dependency parser that converts the sentence into a dependency parse tree. There are many open source dependency parser libraries available and initial plan was to use ANTLR parser but after doing some research on other parsers and comparing different parsers, we come to the conclusion that Stanford parser was the easiest to implement and highly accurate. Also the research paper we used for building this application had used Stanford parser and hence we decided to go with that.
In order to use Stanford parser we first need to download following jar files:

i) stanford-parser.jar
ii) stanford-parser-3.5.1-javadoc.jar
iii) stanford-parser-3.5.1-models.jar
iv) stanford-parser-3.5.1-sources.jar

Then we have to add the jar files to the build path of the project.

Then we will import following libraries in our project:

v) edu.stanford.nlp.ling.CoreLabel;
vi) edu.stanford.nlp.process.TokenizerFactory;
vii) edu.stanford.nlp.parser.lexparser.LexicalizedParser;
viii) edu.stanford.nlp.process.CoreLabelTokenFactory;
ix) edu.stanford.nlp.process.PTBTokenizer;
x) edu.stanford.nlp.process.Tokenizer;
x) edu.stanford.nlp.trees.Tree;

Fig 3.3.1 User Query

Stanford NLP parser reads a sentence in particular language and creates a parse tree from it and generates rules for the same. It supports many languages like English, Chinese, French, Spanish, German etc. It also supports different programming languages like Java, Python, PHP, Ruby, C# etc. In order to use Stanford parser we first need to download all the necessary libraries and then write a small piece of code that takes in the sentence as an input and generates dependency query tree as an output.

Consider a sentence ‘return names for all doctors’.
Fig 3.3.2 shows the parse tree generated for the above sentence

Parse

(Root  
  (S  
    (VP (VB return)  
      (NP  
        (NP (NN name))  
        (PP (IN for)  
          (NP  
            (NP (DT all) (NNS patients))  
            (SBAR  
              (WHNP (WPS whose)  
                (NP  
                  (NP (NNS doctors))  
                  (NP (NN salary)))  
                (S  
                  (VP (VBZ is)  
                    (ADJP (JJR greater))  
                    (SBAR (IN than)  
                      (S  
                        (NP  
                          (NP (CD 2000))  
                          (CC and)  
                          (NP (NN name)))  
                        (VP (VBZ is)  
                          (NP (NNP Ankit))))))))))))))

Fig 3.3.2 Dependency Parse Tree

All the words from the sentence will be the leaf nodes in the tree and their immediate parent node will be the tagging nodes. Tagging nodes are used to identify what is the type of the leaf node.

<table>
<thead>
<tr>
<th>Tagged Node</th>
<th>Type</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>VB</td>
<td>Select Node</td>
<td>Return, Find, Display</td>
</tr>
<tr>
<td>JJR, RB</td>
<td>Operator Node</td>
<td>Less than, equals, not equal</td>
</tr>
<tr>
<td>JJ</td>
<td>Function Node</td>
<td>Average, Maximum, minimum</td>
</tr>
<tr>
<td>NNNST</td>
<td>Table Node</td>
<td>Name of the table</td>
</tr>
<tr>
<td>NNSA</td>
<td>Attribute Node</td>
<td>Name of the attribute</td>
</tr>
<tr>
<td>NNP</td>
<td>Value Node</td>
<td>Value of the attribute</td>
</tr>
<tr>
<td>DT</td>
<td>Quantifier Node</td>
<td>All, Any, Each</td>
</tr>
<tr>
<td>LN</td>
<td>Logic Node</td>
<td>And, Or, Not</td>
</tr>
</tbody>
</table>

Fig 3.3.3 Types of Tagging Nodes
Tagging will be used while interacting with the user to avoid any ambiguity and also during SQL query generation. Stanford parser generates many other tagging but the tags useful for us are the ones mentioned in Fig 3.3.3. Stanford parser assigns single tag for attributes and tables which is NNS. But in order to differentiate between the two we have assigned separate nodes for both of them, NNST for tables and NNSA for attributes.

3.4 Candidate Mapping

Once the dependency parse tree is generated, we have to map the leaf nodes in the parse tree to the attributes or tables in the schema nodes. We are creating this application for the naïve users who don’t need to have much idea about the database schema. So user may not know the exact name of the attribute or the name of the table. But in order to write SQL queries, reference to the attributes and table should exactly match with the database schema. This is where previous NLIDB applications used to fail.

In our system, for every NNS type of node i.e. Attribute node or Table node, we do the mapping with the schema nodes. We compare both the labels and in case of exact match we do the mapping without telling the user. But in case of slightest of mismatch we provide users with all the similar options from the schema and ask them to select one from the list.

For comparing the similarity between words, we have used java Wordnet library which has several similarity functions for e.g. WUP similarity, Path length, Leacock & Chodorow, Resnik, Jiang & Conrath, Lin, Nguyen and Al-Mubaid etc. Each of these functions compares words and provides scores to it. In order to use Wordnet library with the java code, we first need to download the following two jar files:

i) jawjaw-1.0.2.jar
ii) ws4j-1.0.1.jar

Then we need to add the jar files to the build path of our project.

Then we need to import following libraries in your code:

i) edu.cmu.lti.lexical_db.ILexicalDatabase
ii) edu.cmu.lti.lexical_db.NictWordNet
iii) edu.cmu.lti.ws4j.util.WS4JConfiguration

Then we will need to import library specific to the similarity function we want to use. For e.g. in order to use WUP similarity function we need to import edu.cmu.lti.ws4j.impl.WuPalmer

We tried all the similarity functions but WUP similarity function was giving the best results. So we decided to use WUP similarity function for this project.
Wu & Palmer (WUP) Similarity – It compares the similarity between words using the depths of two concepts in Unified Medical Language System (UMLS) and provides the score for each comparison. The formula for score is given as
Score = 2*depth (lcs) / (depth (word1) + depth (word2))
Where lcs is least common subsummer

WUP Similarity function basically compares the meaning of the words and checks for its similarity. Based on the similarity, score is generated for each comparison and it ranges from 0 to 1. 1 being exactly similar and 0 being not similar at all.

But the problem with using WUP similarity function is that it only checks for similarity in meaning of two words and does not compare the characters.

Consider user entering the query ‘Return name for all physician’ for the database schema shown in fig 3.2.1. WUP similarity function will return some score for physician and doctor because they have similar meaning, so we can map these two words. But what about name? There is no attribute or table in the database that is similar to ‘name’ in terms of synonyms. So WUP similarity function will return the score of 0. So we cannot completely rely on this function.

So along with WUP similarity function we have used longest common subsequence algorithm that compares the characters in two words. It will return the score based on the length of the substring and the length of the words. Formula for score is given by
Score = length (common subsequence) / max (length (word1), length (word2))

So the final score returned will be max (score (WUP function), score (longest common subsequence)).

So now for the above query, name will have similarity with firstname, middlename and lastname attributes.

3.5 Candidate Ranking and User Interaction

While mapping NNS nodes from parse tree to the schema nodes, there might be some ambiguity. One NNS node might have similarities with multiple schema nodes. For e.g. in the query ‘return name for all patients’, name has similarity with firstname, middlename and lastname. We can’t be sure which attribute user is referring to. So we cannot directly map name node to one of the schema nodes, we need to clear the doubt with the user.

In order to reduce the number of candidates to show to the user we keep a threshold score which in our case is 0.3. Since we are using longest common substring, even the words that has only few characters similar, will have a score greater than 0. All the candidates with a score of less than 0.3 are discarded because it is considered as a noise. There can be a case
when too many candidates has a score of 0.3 and higher, so if we show all the candidates in random order to the user it would be difficult for the user to find out the correct option. So before displaying all the candidates to the user we rank them based on the score and show the candidates in descending order of their score. It is highly likely that the candidate with highest score is what user will select. This way most of the times, user will get the correct option in top 3 or top 5.

In cases where there is only one candidate option, we do not show it to the user and map that automatically. This will save some time and effort of the user. We show all the candidates for NNS nodes in a dropdown list. User has to select the correct option and click on OK. In the back end, system will map the NNS node with the user selected candidate. This process is repeated for all the NNS nodes in order to remove all ambiguities.

As shown in Fig 3.5, user entered a query ‘return name for all patients whose doctors salary is greater than 2000 and name is Ankit’. So name is the NNS node and has several candidates as shown in the dropdown. User will have to select one option and click on OK. To avoid confusion, we show table name and then attribute name in the dropdown because same attribute might be present in multiple tables. In the above query there are multiple NNS nodes: name, patients, doctors, salary and again name. So each node will be compared to the schema nodes in order of their appearance in the query. Now patient and salary has only 1 candidate generated i.e. node patients -> Table patient and node salary -> Table doctor/Attribute salary. So in this case user will not be asked to solve the doubt and the candidates will be automatically mapped to the NNS node.
One of the drawbacks of using Stanford parser for generating dependency parse tree is that both the attributes and the tables are marked as NNS nodes. So we decided to create two different tags for attribute nodes (NNSA) and table nodes (NNST). This tags are assigned to the nodes when user selects a candidate from the dropdown. If user selects the candidate which is an attribute we change the node tag from NNS to NNSA and similarly when user selects a candidate which is a table, we change the node tag from NNS to NNST. Tagging attribute nodes and table nodes separately is useful while generating SQL queries.

Another problem is that sometimes nodes are tagged as NNS nodes when in reality it was not meant to be an attribute or a relation. In such a case, score returned will be 0 or less than 0.3 and no candidate mapping will be produced using similarity function. In such a scenario we change the node tag from NNS to NNP which means it is a value of one of the attribute.

### 3.6 Parse Tree Structure Adjustor

Once all the doubts are resolved by the user and all the candidates are mapped to correct schema nodes, we need to restructure the tree with correct mapping. We replace the old tags with the new ones and replace the old NNS leaf nodes with the actual mapped candidates. Updated parse tree is shown in Fig 3.6.1.

---

**Fig 3.6.1 Updated Dependency parse tree**
After removing all the unwanted tags from the updated dependency parse tree, visualization of the tree actually looks like the tree shown in fig 3.6.2

Fig 3.6.2 Query tree visualization

3.7 SQL Generation

Once the final query tree is generated, last step is to form an SQL query from the tree. When there are no function nodes or quantifier nodes in the tree, query will not contain any sub-queries. Now the query can have three main clauses: SELECT clause, FROM clause and WHERE clause. Now we need to decide which part of the tree should be present in which clause. All the attribute nodes (NNSA) under VB node will come under SELECT clause. If there is no NNSA node under VB node, then we have to look for the NNST node it is referring to and find out the primary key of that relation. Primary key will become part of the SELECT clause in such scenario. Then we look for table nodes (NNST) in the entire parse tree. All the NNST nodes will come under FROM clause. If there are any value nodes present in the tree, then along with the corresponding NNSA node and operator node it should be added to the WHERE clause. At last if more than one table nodes are present in the tree we look for foreign key/primary key relationship between the those tables and add those attributes to the WHERE clause as well.

Consider the parse tree shown in fig 3.6.2. All the NNSA nodes under VB node should be part of SELECT clause. So SELECT clause will have patient.firstname. In the entire query tree two table nodes (NNST) are present which should be added to FROM clause. So FROM clause will
have doctor and patient. Both these tables have foreign key/primary key relationship which should be added to WHERE clause. So WHERE clause will have patient.primaryDoctor = doctor.ssn. Also value 2000 is mapped to doctor.salary and value Ankit is mapped to doctor.firstname. Hence these two should also be included in the WHERE clause.

Now before forming the actual query, we must map some of the nodes with fixed tags to SQL keywords. This is a pre-defined list and can be extended in the future. Current mapping is shown in fig 3.7.1

<table>
<thead>
<tr>
<th>Tagged Node</th>
<th>Type</th>
<th>Keywords</th>
<th>SQL Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>VB</td>
<td>Select Node</td>
<td>Return, Find, Display</td>
<td>SELECT</td>
</tr>
<tr>
<td>JJR, RB</td>
<td>Operator Node</td>
<td>Less than, equals, not equal</td>
<td>&lt;, =, !=</td>
</tr>
<tr>
<td>JJ</td>
<td>Function Node</td>
<td>Average, Maximum, minimum</td>
<td>Avg(), Max(), Min()</td>
</tr>
<tr>
<td>NNST</td>
<td>Table Node</td>
<td>Name of the table</td>
<td>Maps to table in the DB</td>
</tr>
<tr>
<td>NNSA</td>
<td>Attribute Node</td>
<td>Name of the attribute</td>
<td>Maps to attribute in the DB</td>
</tr>
<tr>
<td>NNP</td>
<td>Value Node</td>
<td>Value of the attribute</td>
<td>Mostly used for Where clause</td>
</tr>
<tr>
<td>DT</td>
<td>Quantifier</td>
<td>All, Any, Each</td>
<td>Mostly used to form sub queries</td>
</tr>
<tr>
<td>LN</td>
<td>Logic Node</td>
<td>And, Or, Not</td>
<td>And, Or, Not</td>
</tr>
</tbody>
</table>

Fig 3.7.1 SQL keyword Mapping

Different clauses for the above parse tree are shown in fig 3.7.2

Elements mapped to VB Node are mapped to SELECT Clause:
  patient.firstname

All the tables referred are mapped to FROM clause:
  patient, doctor

Each Value Node along with its Operator Node and Attribute Node are mapped to WHERE clause:
  doctor.salary > 2000 and doctor.firstname = Ankit

Finally all the Foreign Key/Primary Key relationships are added to WHERE clause:
  patient.primaryDoctor = doctor.ssn

Fig 3.7.2 Different clauses in SQL
So the query generated for the parse tree using above mentioned rules and keyword mappings is

SELECT patient.firstname
FROM patient, doctor
WHERE patient.primaryDoctor = doctor.ssn
AND doctor.salary > 2000
AND doctor.firstname = ‘ANKIT’

4. Results

Application was tested thoroughly on the database shown in fig 3.2.1 and the application returned correct results most of the times for the queries that would not require any joins or sub queries. This is because we haven’t added support for the joins and sub queries in our application. Some of the user queries and SQL query generated after resolving all the doubts are shown below:

i) User input – return all doctors
SQL query – SELECT doctor.ssn FROM doctor

ii) User input – return average salary for doctors whose name is Ankit
SQL query – SELECT AVG(doctor.salary) from doctor WHERE doctor.firstname = ‘Ankit’

iii) User input – return name for all patients and doctors
SQL query – SELECT patient.firstname, doctor.firstname FROM patient, doctor WHERE patient.primaryDoctor = doctor.ssn

iv) User input - return name for all patients whose doctors salary is greater than 2000 and name is Ankit
SQL query – SELECT patient.firstname FROM patient, doctor WHERE patient.primaryDoctor = doctor.ssn and doctor.salary > 2000 and doctor.firstname = ‘Ankit’

Apart from this database, this application was tested on 2 other databases and it generated correct SQL queries for most of the user inputs.
5. Conclusion

We have successfully created an interactive natural language interface to relational databases application that can convert the user input English query into an SQL query. Even naïve users with minimal or no knowledge about the database schema can retrieve the results from the database using this application.

6. Future Work

- SQL Query has too many keywords and we have mapped SQL keywords shown in Fig 3.7. Support for more keywords can be added to this application for e.g. Distinct, Order By, Like, Group By etc.
- SQL also has too many functions and we have added support for MIN(), MAX() and AVG(). More functions can be added e.g. Count(), First(), UCase(), Len() etc.
- Current application can handle queries that do not have joins and sub-queries. So application can be extended by adding support for such complex queries.
- Currently user is given a list of candidates to avoid ambiguity. This candidates are ranked based on the score generated from WUP Similarity function and longest common subsequence algorithm. But it ignores the context of the sentence while ranking candidates. For e.g. consider the query ‘find doctors ssn and name’ for database shown in Fig 3.2.1. In this query current application will give 2 options for ssn i.e. patient.ssn and doctor.ssn and 5 options for name and would be ranked as follows: patient.lastname, doctor.lastname, patient.firstname, doctor.firstname and patient.middlename. But if you look at the query, there is no reference for table patient in the query and hence all the candidates related to patient table should either be removed or ranked lower down the order.
- Since we are already generating the query, we should also consider explaining the query to the user for teaching purposes.
7. **References**

