Pattern Based Approach for Natural Language Interface to Database

Niket Choudhary¹, Sonal Gore²

¹,² Department of Computer Engineering, Pimpri-Chinchwad College of Engineering, Savitribai Phule Pune University, India

Abstract—Natural Language Interface to Database (NLIDB) is an interesting and widely applicable research field. As the name suggests an NLIDB allows a naive user to ask query to database in natural language. This paper presents an NLIDB namely PBNLIDB (Pattern based Natural Language Interface to Database) in which patterns for various SQL functionalities like handling simple query, aggregate function, relational operator, short-circuit logical operator and join among tables are defined. It takes an English language query as input, recognizes pattern in the query, selects the before mentioned functionalities of SQL based on the pattern, prepares an SQL statement, fires it on database and displays the result.

Keywords—Natural language interface, database, SQL, pattern based.

I. INTRODUCTION

Databases are an essential component for any enterprise. They contain huge volume of valuable data. To handle this large volume of data a powerful system called Database Management System (DBMS) is used. One of the most important features provided by a DBMS is SQL. It is used to store, retrieve and process structured data. But use of SQL restricts a naive user to retrieve his desired data. To overcome this problem, many researchers are continuously working on the concept of Natural Language Interface to Database (NLIDB). An NLIDB takes input a query in natural language, translates it into SQL and fires it on the database [1]. The NLIDB is a branch of more comprehensive subject namely Natural Language Processing (NLP). NLP is concerned with creating an easy and user friendly environment to interact with computer without requiring some programming language skills. Through NLP one can interact with computer in his natural language.

Many interesting theories and approaches have been developed so far about how to develop an NLIDB with improved accuracy [2], able to handle more natural language expressions [3,4] and able to guess real requirement of user who has not properly asked the query [5]. In this paper a novel approach for NLIDB is proposed. Section 2 first describes some of the earlier NLIDB systems. In section 3, the proposed NLIDB namely PBNLIDB is explained. Finally section 4 concludes the paper with some opportunities of future work in the project.

II. RELATED WORK

In recent times, there have been rising demands by non-expert computer users to query relational databases in natural language. Actually, research in NLIDB was started in the decade of 1960 [1]. The Lunar Science Natural Language Information System (LSNLIS) [6] was the first system based on the concept of NLIDB. It was actually a question-answering system. It was developed for the geologists who were studying about rocks on moon. The information was obtained by the Apollo missions. It was a waste of time and cost to teach the geologists the programming skill to process and retrieve data. Similarly LADDER was an NLIDB developed for US Naval ships [7]. Then in late seventies RENDEZVOUS System [8] appeared. This system first presented the use of paraphrasing and clarification dialog with the user in case the system was not able to parse the input. In the eighties CHAT-80 [9] was one of the most referenced NLP systems. It was implemented in Prolog and the database was consisted of world facts like oceans, major seas, major rivers and major cities of 150 countries. It also consisted of a small English vocabulary package required to process the query. ASK, developed in 1983, was the system that was able to work upon multiple databases simultaneously. NALIX [10], developed in 2005 is a natural language interface to XML. PRECISE [2] presented an interesting idea of making interaction with user more human like. In this if user asks one question and then asks another similar question by only changing the values then he do not need to ask the complete question. He can only ask partial question and remaining words will taken from previous question. For example if user first asks a query “who is the author of Algorithms?” then while asking second question he do not need to ask complete query, he can only ask “Database?”. It will automatically be taken as “who is the author of Database?”. Generic Interactive NLIDB (GINLIDB) [11] came with a component namely database adaptor which allowed the system to interact with multiple DBMS tools. It is used to set the environment according to the DBMS in use. One very recent NLIDB namely AskMe [12] presented a feature query-authoring service which helps a user to frame the query properly so that it can be validated and confirmed that it will not generate error when fired on database. It does not wait for the query to be fired on database and get error if any. It identifies error while framing the query.
It is difficult to say which existing interface is the best but through the study; the problem that we found is that they do not support most of the features of SQL. This made us to work in this direction. We developed the interface with the intention that it makes use of most of the features of SQL.

III. PROPOSED SYSTEM
The proposed system is mostly based on the ideas implemented in [3]. The only difference is that their idea of templates is replaced with more sophisticated Expert System which is responsible for converting English language query into SQL query. Architecture of proposed NLIDB namely PBNLIDB is shown in figure 1. Initially hyponyms, hypernyms and synonyms for all table and column names are found out using WordNet. Database Administrator can further add or remove the newly added similar words. This pre-processing is done only once. The purpose of pre-processing is to develop a knowledge base so that if user is using income instead of salary and in the database column name is salary then income will be mapped to salary. User will not be given a message that the query is not relevant. At run time initially an English language query is accepted. This query is syntactically parsed and tagged using Link Parser. Then it is tested semantically- whether the query asked is relevant to the database or not. Finally the most important component Expert System is used to translate the asked query into SQL query. Figure 2 shows internal modules of the Expert System. In the Expert System patterns for various functions of SQL like handling simple query, aggregate function, relational operators and short circuit logical operators are defined. These features are discussed in the following sections taking the example of an employee database. The E-R diagram of employee database is shown in figure 3. Once pattern of these functions is identified, one or more of these functions is selected, an appropriate SQL statement is prepared, fired on database and result is displayed to the user. To begin with the discussion of functions included in PBNLIDB, first refer to table 1 for various acronyms used for defining patterns.

![Fig. 1 Architecture of PBNLIDB](image-url)
An SQL template containing all the acronyms mentioned in table 1 is as shown below:
select <AF><DC> from <TB> where <PC> <RO> <VL>

A. Simple Query
This includes all those queries in which only TB or DC with their TB are mentioned. Following are few examples of simple queries based on employee database.
Example 1: List details of all employees.
Here “details” tells to select all column names and employee is identified as TB.
So the SQL query generated is
SELECT * FROM EMPLOYEE

Example 2: Show the names and salaries of all employees.
In this example name and salary are identified as DC and employee as TB.
So SQL query generated is
SELECT NAME, SALARY FROM EMPLOYEE

Thus valid patterns for simple query are- {TB}, {DC, TB};
And invalid pattern is {DC}.
Invalid pattern {DC} indicates that asking only column names without their table name is not allowed and user is asked to further clarify the query by selecting a table name from many produced as options by query expansion module.

B. Query requiring aggregate function
Various aggregate functions supported by our interface are count, sum, avg, min, max and distinct. Actually more than these can be seen in a DBMS tool but in our interface, only the aforementioned aggregate functions are included. Few examples mentioned below are the queries that require aggregate function:
Example 1: What is the number of employees working in the industry?
Here AF is count, DC is name and TB is employee. Name is identified as DC because it serves as default attribute [3] for the table.
In the above examples one aggregate function, column base. “total” in the query is mapped to SUM using the knowledge base. Similar to previous example here SQL query generated is SELECT SUM(SALARY) FROM EMPLOYEE.

Example 2: What is the total salary of all employees?

Example 1: Who is the employee whose salary is more than 20000 and profile is manager?

In this example DC is name (default attribute), TB is employee, PC are salary and profile, RO are > and <, VL is 20000 and manager and SC is “and”. So the SQL query generated is SELECT NAME FROM EMPLOYEE WHERE SALARY > ‘20000’ AND PROFILE = ‘MANAGER’.

Example 2: Who is the employee whose salary is more than 20000 and less than 50000?

Here DC is name, TB is employee, PC is salary, VL is 20000 and 50000, RO are > and < and SC is “and”. Thus SQL generated is SELECT NAME FROM EMPLOYEE WHERE SALARY > ‘20000’ AND SALARY < ‘50000’.

Example 3: Who is the employee whose salary is more than 20000 and less than 50000 and profile is either manager or team leader?

In this case DC is name, TB is employee, PC are salary and profile, RO are >, <, =, VL are 20000, 50000, manager and team leader. The SQL query generated is SELECT NAME FROM EMPLOYEE WHERE SALARY > ‘20000’ AND SALARY < ‘50000’ AND PROFILE = ‘MANAGER’ OR ‘TEAM LEADER’.

Observing the above examples it is clear that at least one TB, more than one PC with their RO and VL or one PC with more than one VL has to be present separated by “and” or “or”.

Thus valid patterns for short-circuit logical operator are- {PC, RO, VL, SC}, {DC, TB, PC, RO, VL, SC}. For invalid pattern mainly count of various entities in the set are considered. For example the number of PC and VL must be equal if there are distinct PCs are used. For one PC multiple VL are possible. The count of SC is one less than the number of VL. In a query there can be many PC with different count of their VL. To handle such situation their position in the query is considered. To identify VL of PC1 all VL present between PC1 and PC2 are taken. Similarly for VL of PC2 all VL between PC2 and PC3 are taken and so on. For final PC, PC3, VL coming after PC3 and before end of the query are taken as VL for PC3. In this case there are no special patterns are defined, instead the count and position of various PC, RO and VL are considered and valid and invalid patterns of relational operators are used with SC as one more data of the set.

E. Query requiring join

Join is used to fetch records from multiple tables. Following are the examples considered for identifying the valid and invalid patterns of Join operation.

Example 1: Which employee is doing the project of ABC Bank?

Here there are two different TB are used- one is employee and another is Project. In this case the relationship WorksOn is also taken by default. The SQL query generated is SELECT EMPLOYEE.NAME FROM EMPLOYEE, WORKSON, PROJECT WHERE EMPLOYEE.EMPID = WORKSON.EMPID AND WORKSON.PROJID = PROJECT.PROJID.
Example 2: Which employee is doing the project of ABC Bank in Mumbai branch and salary is 2000? Here extra conditions are put other than default as in the previous example. Extra conditions include PC as project.branch and employee.salary, RO as = and =, VL as Mumbai and 2000.

Thus SQL query generated is

```
SELECT EMPLOYEE.NAME FROM
EMPLOYEE, WORKSON, PROJECT
WHERE
EMPLOYEE.EMPID = WORKSON.EMPID AND
WORKSON.PROJID = PROJECT.PROJID AND
PROJECT.BRANCH = 'MUMBAI' AND
EMPLOYEE.SALARY = '20000'
```

Observe the above examples we get that a query in this group contains more than one TB plus it may contain extra predicates. Without any extra predicate default SQL is predefined but presence of extra predicate makes it follow the rules of relational operator or short circuit logical operator or both.

Thus briefly explaining, a set of valid and invalid patterns of various functionalities of SQL are defined in the Expert System. Query Expansion module of Expert System is used to provide hints to the user in case the query asked contains an invalid pattern. Hints are provided based on valid pattern of the functionality for which invalid pattern is identified. The sets formed in both valid and invalid group of all functionalities are unique. Thus there is no conflict in identifying a functionality based on the pattern identified from the English language query.

### IV. Conclusion And Future Work

In summary, the proposed system contains most of the modules of the natural language interface for CINDI virtual library. The only replacement made is that of template based translation with pattern based translation from English language query to SQL. The intent of using patterns is to support more complexities in translating function and making the interface more aware of the functions available in SQL. Currently, the proposed system supports handling simple query, aggregate function, relational operator, short-circuit logical operator and join. For each of these functionalities, a set of valid and invalid patterns are defined. In case any invalid pattern is identified, query expansion module provides options to the user containing valid patterns for that functionality. Unique Patterns are defined for simple query, aggregate functions and relational operators. Short circuit logical operator and join work in collaboration with the patterns of relational operator. There is no need to define lengthy and duplicate patterns for short circuit logical operator and join. Our next target will be to make the interface aware of other features of SQL like clauses (group by and order by), keywords used in predicate like BETWEEN, NOT IN, IN etc., nested query, varieties of join, union, intersection, difference and much more. More attention will be made towards developing interface in such a manner that a new feature could use the patterns or functioning of its previous features.

### REFERENCES


