Disclaimer

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious.
I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy.
I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references.
I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.
By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.
By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml)

Name(s): Kevin Meehan

Date: 21st August 2017

Natural Language Interface to Databases - A Modelling Approach to Assess Natural Query Structure

Kevin Meehan

***Abstract*—Databases store a vast amount of information used world-wide on a daily basis. Database query languages such as Structured Query Language (SQL) allow skilled users to easily access this data. For a naive user however, extracting this data can be quite difficult. Natural Language Interface to Database (NLIDB) enables a user to query the database in their natural language eg. English, Chinese, etc. The natural language query (NLQ) is converted into a database language query, processed and the generated result relayed to the user.**

Many variations exist in terms of the methods used to convert NLQ to SQL. Most systems which boast high levels of accuracy are limited in there specificity to a given database. This research project uses a pattern-matching technique to build a system capable of achieving high levels of accuracy across multiple unrelated relational databases. It looks at whether the presence of keywords related to the database and their order in the NLQ can be fitted to a model based on the general syntax of an SQL query in order to address this issue of portability.

I. Introduction

Advancements in data gathering techniques have resulted in vast quantities of data being generated and stored. As such, to optimise the benefits of this data there is an increased requirement for it to be accessed and manipulated. For the naive user, this may be complicated. NLIDB enables the user to obtain database queries in their own natural language, saving them from having to learn database query languages such as SQL, which can be both time-consuming and difficult.

Research work in NLIDB began in the 1960s [1]. Since then various solutions have been proposed, with varying methods, limitations and accuracy of results. The traditional NLIDB model accepts an NLQ, parses the query into its various grammatical components and uses these components to generate an SQL query equivalent to the NLQ [2]. Variances occur among different researchers as how best to use the various components in generating the SQL query.

Early researchers used different approaches such as pattern-matching systems, syntax-based systems, semantic grammar systems and intermediate representation language systems [3].

Pattern-matching systems use pre-defined fixed patterns and rules for mapping. They are relatively simple to implement but can be quite shallow in their approach, which can lead to big errors [1]. They are the least specific of the various approaches however, which bodes well for addressing the issue of portability.

Syntax-based systems parse the natural language query into its grammatical components and map these components to their semantic meaning to generate a database query. They are usually application-specific and it can be quite difficult to generate accurate mapping rules [1], although corrections in dictionary errors has resulted in an increase in performance [4].

Semantic-grammar systems map the parsed NLQ components to corresponding words from the database corpus. Semantic grammar systems are easier to implement than syntax-based systems, but require a prior knowledge to various elements of the specific domain [4]. These systems perform well, but because they contain hard-wired knowledge about the domain, they are not portable across multiple databases [1].

Intermediate representation language systems convert the natural query to logical query and then to structured language query. Similar to the previous approaches, this system can achieve high levels of accuracy but is limited in its specificity to a given database [3].

Recent systems claim to achieving higher accuracy of results through more advanced mapping techniques, such as the use of word co-occurrence matrices (to match frequently occurring phrases) [3], intellisense (which prompts query input) [5] and machine learning [6]. These methods will be further explained in the following section.

As is evident, portability across databases is a big limitation in the field of NLIDB. Many systems boast high levels of accuracy, but only specific to a single database. In order to address this issue, this paper proposes a system which hails predominantly from the pattern-matching approach, but which also incorporates semantic-grammar modules to garner meaning from keywords in queries.

The remainder of this paper is outlined as follows: Section II provides a literature review of past and current NLIDB systems along with their advantages and limitations, Section III outlines the architecture of the system proposed in this paper, Section IV outlines the experimentation and results of this system and Section V offers a brief conclusion and future work.

II. Literature Review

Research work in the field of NLIDB began in the 1960s, but has only really accelerated in the last 20 years.

SAVVY is an NLIDB system which makes use of pattern-matching techniques similar to those used in signal processing

[1].

LUNAR was one of the first NLIDB systems and applies a syntax-based approach. It was specific to a database of rock samples retrieved from the Apollo missions to the moon. The prototype worked well, achieving a 78% success rate in one of their tests, but similar to most syntax-based systems was limited in it's specificity [7].

The LADDER system made use of semantic grammar techniques to parse natural language queries into structured database queries [4]. The system was specific to the domain of Navy command and control. Although based on simple principles and subject to certain limitations, the system was considered "sufficiently robust to be useful in practical applications" [8].

CHAT-80 is one of the earliest systems which implemented an intermediate representation language approach [3]. The system is specific to the domain of world geography and is implemented entirely in Prolog, a logical based programming language [9]. Although the system is specific to one domain, it was claimed that the CHAT-80 system "probably has the best combination of efficiency and portability of any comparable system at the present time, due principally to the use of Prolog as the implementation language" (Warren & Pereira, 1982).

More recent implementations of NLIDB systems have tweaked various components of existing systems.

In 2015, Huang et al. proposed an advancement on the syntax-based system using probabilistic context free grammar, which outlines an algorithm to compute inside and outside probabilities used in constructing the parse tree, leading to a more accurate SQL query. They conclude that a probabilistic approach "may be promising" [7].

In 2013, Shah et al. proposed a solution for both syntactically correct and incorrect natural language queries based on an amalgamation of NLIDB and Keyword Based Interface to Database(KBIDB). They claim a 53% increase in accuracy when NLIDB is combined with KBIDB [10].

In 2015, Mohite and Bhojane proposed a system which used a word co-occurrence matrix technique to better generate a database query. They concluded that their system returned "correct answers of simple queries, queries with logical conditions and aggregate functions", however was unable to "support all forms of SQL queries" [3].

Additionally in 2015, N. Choudhary and Gore outlined a system which made use of intellisense to "generate suggestions based on previously typed words to help frame a complete and correct query". They claim that through appropriate use of their intellisense guidelines that a user will achieve 100% accuracy in their queries [5]. However, it may still be the case that a user doesn't understand the guidelines or simply neglects to adhere to them.

In the same year, M. Choudhary et al. proposed a system [11] which made use of a web-based bilingual interface for NLQs entered in Hindi or Punjabi. Their system makes use of three modules; a linguistic module, a domain recognizer module and a database module. They boast good results but like most other sytems is domain specific.

This lack of portability is a big issue when it comes to practicality of an NLIDB system. Although these systems are useful in their fields, and their methodologies certainly useful wider afield, ultimately they are bounded in their effectiveness. The pattern-matching system proposed in this paper aims to combine the components of systems where portability is not a major issue in order to develop a solid base upon which user interactivity can be improved without loss of accuracy.

In 2016, Bais et al. proposed a system using an intermediate representation language approach based on machine learning for a relational database. They claim accuracy of 94.79% and that their system is advantageous to others as it is "independent of the database language domain and model and it is able to automatically improve its knowledge base through experience" [6].

In order to develop a pattern-matching system where portability is not an issue, it is important that keywords can be identified from the NLQ. In 2008, Hoque et al. propose a methodology [12] for extracting significant phrases in the database NLQ . They make use of soft computing and heuristics and boast an 88.5% success rate across three different databases. These techniques are likewise implemented in this paper's system.

In the same year, Djahantighi et al. offer a simple but elegant system [13] developed in prolog for dealing with ambiguity in NLQs. Dealing with ambiguities is a key task when trying to optimize portability. One of the difficulties faced in this system is in dealing with multi word expressions. For example, the words *Cambridge* and *University*, although related, have two very different meanings on their own versus when they are combined.

Another issue key to obtaining portability is deriving meaning from queries. An NLQ for one database may have an entirely different meaning to an NLQ for a different database. Li and Shi [14] provide a detailed report on a WordNet-based interface, which first syntactically parses the NLQ, and then uses the WordNet tool in order to derive its meaning. The use of WordNet in identifying semantics has since been adopted by others such as Billal et al. [15], who made use of it in order to preprocess large datasets. WordNet has been incorporated into the system proposed in this paper for this purpose of identifying semantics.

Once the keywords and their semantics have been extracted from the NLQ, the final step is to map them to the pre-defined pattern. Kumar and Dua propose a system [16] capable of converting a Hindi NLQ to SQL using a pattern-matching technique, although the pattern used is not explicitly outlined. They do however provide a solid high-level view of their architecture and claim "promising results" on their experiments.

III. Proposed System

The system proposed in this paper was constructed by analysing a sample of NLQs and building algorithms to process their structure. NLQs were chosen to reflect diversity in the structure of queries. It aims to become a working platform for future use i.e. it aims to process queries of a simple structure, but be easy to append more complex structures. This is in order to achieve portability, which this paper prioritises

(for now) over accuracy. The following figure is a high-level view of this system:

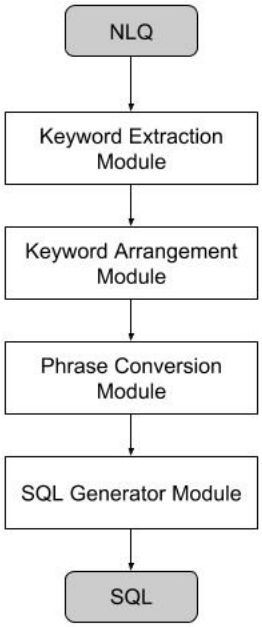

Fig. 1: An overview of the system

The NLQ is entered as input and from there it is processed by four modules to produce the resulting SQL query. In the context of this system, a *keyword* refers to any word in the NLQ which can be mapped to a *header*, *value*, or *function*, where:

- a *header* is a column header/name of a column in a table in the database
- a *value* is a column value/attribute of a column in a table in the database and
- a *function* is an SQL function that may be applied to headers and values. Note that this term covers both the SQL functions defined as *aggregators* (”COUNT”, ”SUM”, ”MAX”, ”MIN”, ”AVG”) and SQL functions not defined as aggregators (”SELECT”, ”WHERE”, ”ORDER BY (DESC)”, ”GROUP BY”, ”LIMIT n”).

It is assumed that the user declares which table he/she is analysing.

A. Keyword Extraction Module

In this module, the keywords from the NLQ and their corresponding meanings are identified first for headers, then values and then phrases. In most NLIDB systems the NLQ undergoes a cleaning process - which generally removes stopwords, punctuation and tokenizes the query on whitespace - before it enters any modules for syntactic or semantic analysis. This system defers this step (in its absoluteness), as to permanently remove characters/strings initially deemed insignificant may be of detrimental consequence later in the conversion process. Instead, the same NLQ undergoes different cleaning processes for different types of analysis in order to gain a better overall view of its context. This module can be represented as follows:

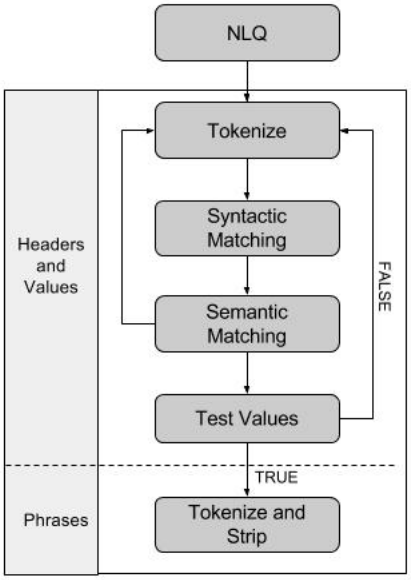

Fig. 2: How the Keyword Extraction Module operates on the NLQ

The NLQ enters the system and keywords are first extracted for headers. The NLQ and all headers are converted to lowercase. Using the Natural Language Toolkit (NLTK) [16], the NLQ is tokenized based on the headers in the table. So for example, in a table consisting of students, modules and corresponding grades and given the query ”How many students sat history of mathematics?” - where ”History of Mathematics” is a table header - then the query would be tokenized as [’how’, ’many’, ’students’, ’sat’, ’history of mathematics’]. This would result in a direct syntactic match with the header ”History of Mathematics”, enabling us to append it to a table of results and to remove it from the query for further processing.

As mentioned earlier, multi word expressions can cause difficulty in natural language processing. However, as in this example, NLTK proves very useful in overcoming this issue. It’s Multi-Word Expression Tokenizer allows us to tokenize strings (in this case the NLQ) based on a corpus of multi-word expressions (the list of headers).

The next step is to semantically analyze the query. For the same table, given the query ”Which pupil achieved the highest grade in biology?” a syntactic parse may only recognise the header ”biology”, although we would also like it to identify the word ”pupil” as being linked with the header ”student”. We therefore make use of NLTK’s similarity functions in order to derive a link. We can set a threshold and return [keyword, header] pairs whose similarity exceeds this threshold. We now have a link between ”pupil” and ”student” and have recognised

all headers required for our SQL query.

We adopt the same approach for identifying keywords related to values. Only non-numeric values are considered in this step, as numeric values could be the same for multiple headers. It may be the case however that we may have multiple values returned for the same keyword. Consider again the header "student", which contains a list of names. If our previous query was in fact "Which person achieved the highest grade in biology?", then the original threshold may be high enough to link "pupil" to "student", but not high enough to link "person" to "student". When we start searching for keywords related to values however, it may start linking the word "person" to multiple student's names, which we don't want. Therefore we test our values. If the same keyword is being listed too often, then we recalculate the similarity of that word alone with the list of headers and a lower threshold to ascertain whether or not that the keyword should map to a header and if so edit our results.

Finally, we analyse the query for words linked with functions. We currently have a query where keywords linked to headers and values have been identified. It is the other words in the NLQ which now hold our interest. Consider our first two sample queries. If we remove the keywords and split on whitespace we obtain:

- ['how', 'many', ' ', 'sat', ' ']
- ['which, ' ', 'achieved', 'the', 'highest', 'grade', 'in', ' ']

Using NLTK's list of stopwords we can break these down further:

- ['how', 'many', ' ', 'sat', ' ']
- ['which', ' ', 'achieved', ' ', 'highest', 'grade', ' ', ' ']

We can now concatenate phrases not separated by empty strings in order to obtain some context about their meaning. This list of concatenated phrases is passed into a later module to map to SQL functions.

B. Keyword Arrangement Module

This module takes the output from the first module and organises it into a suitable format. Tables of $[keyword : header]$ mappings, $[keyword : value]$ mappings and $[keyword : function]$ mappings are accepted as input. (Note that for the latter, it is yet unknown which functions are to be applied, only the keywords that will be used. These mappings are passed as $[keyword : "TBC"]$, where "TBC" will be generated in the next module.) The tables are combined and sorted based on the order of the keyword in the NLQ; this is important for mapping to a pattern.

C. Phrase Conversion Module

This module takes as input the list of phrases to be mapped to functions and identifies the correct function. Not all phrases will end up being mapped. The mapping is carried out using a dictionary of {function:[synonyms]}, where each function relates to an SQL function and the list of synonyms corresponds to a list of words or phrases that imply the use of that function. This dictionary was manually compiled based on a thesaurus and can be easily appended to.

This module also attempts to identify date values. As mentioned earlier, numeric values are not considered when computing the table for $[keyword : value]$. However, keywords relating to numbers are of importance. Therefore, if a keyword contains a number, the database table is analysed to see if it has any columns related to dates and if so, the values of that column are analysed to see if the keyword is present.

Algorithm 1 Keyword Extraction Module

```
Input Args: NLQ as q, database table as tbl, similarity threshold as sim
for type in [HEADERS, VALUES] from tbl do
  res := table of [KEYWORD, TYPE]
  tokenizer := tokenizer based on elements in type
  tk_query := query tokenized by tokenizer
  for token in tk_query do
    if token in type.elements then                    ▷ exact match
      res = res.append([token, type])
      tk_query = tk_query.remove(token)
    else                                              ▷ similar match
      for x in type.elements do
        if word_similarity(token, x) > sim then
          res = res.append([token, type])
          tk_query = tk_query.remove(token)
  if type == VALUES then
    test res for incorrect entries
    if test fails then
      recalculate HEADERS with lower sim
      remove entries from VALUES
return res for each type
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
res := table of [KEYWORD, PHRASE]
tokenizer := tokenizer based on HEADERS, VALUES and Stopwords
tk_query := query tokenized by tokenizer
for token in tk_query do
  if token not in HEADERS or VALUES or Stopwords then
    res = res.append(token)
return res
```

Fig. 3: The algorithm for identifying keywords in the natural language query

D. SQL Generator Module

This module converts the keywords and their mappings - based on their order - into a SQL query. The module deals only with SQL queries that *select* data from tables i.e. it does not create, drop or insert into tables and it does not update or delete records in tables. The module works off the base that all SQL queries of this nature (i.e. 'SELECT' queries) must

fit into the following pattern:

- SELECT (<aggregator>)<header(s)> FROM table
- (WHERE <header(s)> <logical operator> <value>)
- (GROUP BY <header(s)>)
- (ORDER BY (<aggregator>)<header(s)> (DESC))
- (LIMIT n)

Where:

- parentheses denote optional arguments
- an aggregator in this context is an element of the set of SQL functions {'COUNT', 'SUM', 'AVG', 'MAX', 'MIN'}
- an aggregator preceding a heading denotes that aggregator *applied to* that heading

The module is divided into three components:

1) Identify 'SELECT' statement
2) Identify (optional) conditional statement i.e. 'WHERE' statement
3) Identify (optional) combinatorial statements i.e. 'ORDER BY', 'GROUP BY' and 'LIMIT' statements

The following algorithm is used for component (1):

Algorithm 2 Extract 'SELECT' Statement

```
1: Input args: dataframe = {[Keyword]; [Mapping]; [Position]}, table_name
2: ▷ Keyword from NLQ, Mapping = header/value/function
   SQL := []
3: for row in dataframe do
4:   if row.Mapping = 'SELECT' then
5:     if row.next.Mapping is value then
6:       h := header such that row.next.Mapping in headers
7:       SQL = SQL.append("SELECT" h "WHERE" h "=" row.next.Mapping)
8:     else if row.next.Mapping is function then
9:       if row.next.next.Mapping is header then
10:        SQL = SQL.append("SELECT" row.next.Mapping row.next.next.Mapping)
11:    else
12:      SQL = SQL.append("SELECT")
13:      while row.next.Mapping is header do
14:        SQL = SQL.append(row.next.Mapping)
15:        next row
16: return SQL, position
```

This algorithm encompasses 'SELECT' statements of the form:

- SELECT <header> FROM table_name WHERE <header> = value
- SELECT <aggregator> <header> FROM table_name
- SELECT <header1>..<headerN> FROM table_name

The SQL outputted from this component, and its corresponding position in the dataframe, is inputted to the following component to determine the presence of conditional statements:

Algorithm 3 Extract 'WHERE' Statement

```
1: Input args: dataframe = {[Keyword]; [Mapping]; [Position]}, SQL, position
2: for row in dataframe.position do
3:   if row.Mapping in values then
4:     SQL = SQL.append("WHERE" header "=" value)
5:   else if row.Mapping = "DATE" then
6:     SQL = SQL.append("WHERE" date_header "=" value)
7:                                   ▷ This segment of code identifies headers related to datevalues. It computes both equality and ranges of datevalues.
8: return SQL, position
```

This system struggles to identify conditional statements which apply logical operators to numerical columns. This issue is discussed in detail in Section IV.

The next step is to identify combinatorial statements. First, it must be identified whether an 'ORDER BY' or 'GROUP BY' statement is needed. This is the trickiest stage of the pattern matching process. Information about the schema is required for complete accuracy.

Consider numerical columns. Depending on the context, either a high number or a low number could be considered good/bad, best/worst, top/lowest. For example, a golfer's round of 70 would better a round of 80, whereas a football player scoring 30 goals a season would better 20 goals a season. To compute an 'ORDER BY' statement therefore requires an understanding of the context of each header. This system has some hardwired knowledge of context, and although it works for some tables, it will not work for all. Therefore for complete accuracy it is essential that the domain context be generated.

Groupings (via 'GROUP BY' statements) require a knowledge of the relationship between each value in a table row. Depending on what the user is looking for, if a grouping is required then the program must know which columns can and cannot be grouped. For example, given a table consisting of students, lecturers, modules and grades, it is possible to calculate a student's highest overall grade by performing a grouping on students, but not on teachers or modules.

This cannot be hardwired into the system, so in order to complete this step, a separate module is required to ascertain the relationship between the database columns.

Finally, if a specified number of results are to be returned, then a 'LIMIT' statement must be used. This component is hardwired into the system.

It is in this step that most issues with the system arise. These issues are explored and solutions proposed in the following section.

IV. Experiment and Results

For testing the system [26], I processed 110 queries from 9 different open source database tables. Queries were chosen to reflect variety in query structure, complexity in queries and ambiguity in choice of words. The results of the experiment are outlined in Table I.

Table	Queries	Correct	Incorrect	Similar
universities [17]	20	3	17	6
movies [18]	20	2	18	9
census [19]	10	8	2	1
websites [20]	10	4	6	0
stations [21]	10	3	7	2
game sales [22]	10	2	8	5
magazine [23]	10	2	8	3
NASDAQ [24]	10	2	8	0
IMDB [25]	10	0	10	8
Total	**110**	**26**	**84**	**34**
Percentage	-	**23.64%**	**76.36%**	**30.91%**

TABLE I: Results of testing on various data tables

A correct result indicates an exact match of the NLQ to it's corresponding SQL; an incorrect result indicates anything else. A similar result corresponds to an incorrect result which is very close to a correct result. For example, it may be output which contains the correct answer but also contains extra information, or it may be output which contains an incorrect aggregating function but all other details are correct.

Although system accuracy is poor at 23.64%, portability is stable. The results prompt a standard deviation of 2.2, with only one table (census) falling outside one standard deviation of the mean.

Issues with the system are outlined in Table II. Issues (1) and (2) account for the majority of problems. Issue (2) is a result of the variety in which queries can be phrased. For example, the system will respond correctly to the query "How many students who sat mathematics achieved an A grade?", but it will not respond correctly to the query "Of the students who sat mathematics, how many achieved an A grade?" due to the order of the keywords in the query. This can however be achieved by appending possible layouts to the current algorithm in constructing the SQL.

Issue (1) is a combination of multiple issues. If 2 or more of issues (3)-(12) exist, then the NLQ fails on issue (1). Therefore if issues (3)-(12) can be remedied, then issue (1) will not arise.

Issue (3) arises when extra output is emitted. This can be fixed by cross-checking the final output against the query to assess how many pieces of information are required. Similarly with issue (9), keywords can be cross-checked for ambiguities at the end of module 1 and requests made of users to ascertain which is correct. An example of this issue is when the keyword "Balbriggan National School", meant to be interpreted as a value, is matched with the header "School Name", or when the keyword "max", meant to be interpreted as a function, is matched with the value "Mad Max: Fury Road".

Issues (4) and (12) arise when, as mentioned earlier, the context of a numerical column is unknown. The solution is to introduce a Context Manager Module (CMM) on loading the database to derive column context. The CMM would also provide a relationship between columns in order to address issue (8). When keywords are not explicitly stated or have multiple meanings, we get issues (7) and (11). For example, the universities table listed above provides a list of rankings for universities under different criteria, the main being "world rank". However, the query "which universities were in the top 10?" does not specify which criteria is desired, even if the user is implying the "world rank" criterion. A CMM would be able to pin-point key database components in order to make up for this lack of explicit information. As much as a human user might explicitly omit a keyword, they may also misuse one. For instance, the term "most common" generally corresponds with "mean" or, in SQL, the "AVG" function. However, take the query "what is the most common occupation for white males?", in which case the user desires the modal value and thus in SQL the "MAX" of a number of occupations. A CMM would be capable of interpreting this subtle difference.

The construction of a CMM could prove the most difficult of the solutions to this system's issues. It would be tricky to identify primary keys and to establish links between database columns, however it may be possible through mapping techniques. To derive context for numerical columns would prove more difficult and may require a dictionary - which may or may not be currently available - to provide an automated solution or, perhaps easier, a user prompt to order numerical

ID	Description	Count	Similar	Solution
(1)	Output is missing a necessary conditional statement	19		Multiple
(2)	NLQ structure not currently recognised by system	15		System additions
(3)	Output contains correct but extra undesired information	8	Yes	Cross-check
(4)	Output contains ORDER BY statement in reverse order	6	Yes	Context MGR
(5)	Output contains header/value similar to desired header/value	5	Yes	User request
(6)	A verb/adjective describing a column is read as an actual value	5		POS tagging
(7)	A desired header is excluded from output	6	Yes	Context MGR
(8)	Output requires a GROUP BY statement	3		Context MGR
(9)	A header/value/function keyword incorrectly assigned due to ambiguity	5		Cross-check
(10)	The incorrect aggregating function is used	3	Yes	Context MGR
(11)	Output should but does not contain a certain aggregating function	3		Context MGR
(12)	Output should but does not contain ORDER BY and/or LIMIT statement(s)	2	Yes	Context MGR

TABLE II: Reasons for Incorrect output

columns from ”high-low”, ”best-worst” etc. which may be more tedious but definitely more accurate.

Finally, issue (6) is one which arises when words relating to the database are confused with words in the database. For example, in the query ”Which British university ranked highest?” the adjective ”British” corresponds to any universities located in the United Kingdom. This system however returns the value ”British University of Colombia” as there is such a high similarity between the pair. In order to address this, part-of-speech (POS) tagging could be used. This is a method of extracting the word type (eg. noun, verb, adjective etc.) from a given word. It is used frequently in syntax-based systems and semantic grammar systems. For this system, it could be implemented after the system checks for direct keyword matches and before it checks for similar keyword matches in order to ascertain whether descriptive words are being used to highlight conditional statements e.g. ”WHERE country = United Kingdom”.

This POS module would be easy to incorporate into the current system and could be built using NLTK.

V. Conclusions

At present, this system provides low accuracy but consistent portability. Where queries are simple, desired headers/values/functions are explicitly stated and are ordered in compliance with the system then accuracy is high. Accuracy is low when the system receives small amounts of information and must work to derive similarities between the query and database. Issues exist, but a platform is in place which can be easily built upon. Four solutions are proposed in order to resolve all issues. Completion of cross-checking, user request, CMM and POS tagging components would, I believe, bring accuracy from 23.64% to at least 54.55% and probably higher, as failures arising from issue (1) would also decrease. The foundation for a universal NLQ to SQL converter is in place and through future work accuracy will improve.

VI. Future Works

Future improvements on this system would include the components already mentioned. Ultimately, an interface would be developed around this pattern-matching model to sit on top of a relational database management system.

More complex query structures would be appended to the system for more detailed searches. Ideally, these structures would be appended automatically by the system through interaction with the user.

A time-series component would also be explored, in order to attempt to fit a model based on a specific users historical tendencies when forming queries. This would speed up the conversion process as not all steps would need to be completed.

VII. References

[1] I. Androutsopoulos, G.D. Ritchie and P. Thanisch, ”Natural Language Interfaces to Databases - An Introduction” in Journal of Language Engineering, pp.2–15, 1995.

[2] F. Siasar Djahantighi, M. Norouzifard, S.H. Davarpanah and M.H. Shenassa, ”Using Natural Language Processing in Order to Create SQL Queries” in Proceedings of the International Conference on Computer and Communication Engineering, 2008.

[3] A. Mohite and V. Bhojane, ”Natural Language Interface to Database Using Modified Co-occurrence Matrix Technique” in International Conference on Pervasive Computing (ICPC), 2015.

[4] B. Sujatha and S.V. Raju, ”A generic model for Natural Language Interface to Database” in IEEE 6th International Conference on Advanced Computing, 2016.

[5] N. Choudhary and Prof. S. Gore, ”Impact of intellisense on the accuracy of Natural Language Interface to Database” on IEEE, 2015.

[6] H. Bais, M. Machkour and L. Koutti, ”Querying Database using a universal Natural Language Interface Based on Machine Learning”, 2016.

[7] B-B. Huang, G. Zhang and P. C-Y. Sheu, ”A Natural Language Database Interface Based on a Probabilistic Context Free Grammar” in IEEE Internationa Workshop on Semantic Computing and Systems, 2008.

[8] G.G. Hendrix, E.D. Sacerdoti, D. Sagalowicz and J. Slocum, ”Developing a Natural Language Interface to Complex Data”, ACM Transactions on Database Systems, pp.106–141,1978.

[9] D.H.D. Warren, F.C.N. Pereira, ”An Efficient Easily Adaptable System for Interpreting Natural Language Queries”, Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA, 94025, pp.110-118, 1982.

[10] A. Shah, Dr. J. Pareek, H. Patel and N. Panchal, ”NLKBIDB - Natural Language and Keyword Based Interface to Database” in International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp.1569–1576, 2013.

[11] M. Choudhary, M. Dua and Z. S. Virk, ”A Web-Based Bilingual Natural Language Interface to Database” in Third International Conference on Image Information Processing, 2015.

[12] M.M. Hoque, M.S. Mahub and S.M.A. Al-Mamun, ”Isolating Significant Phrases in Common Natural Language Queries to Databases” in Proceedings of 11th International Conference on Computer and Information Technology, 2008.

[13] F.S. Djahantighi, M. Norouzifard, S.H. Davarpanah and M.H. Shenassa, ”Using Natural Language Processing in Order to Create SQL Queries” in Proceedings of the International Conference on Computer and Communication Engineering, 2008.

[14] Hu Li and Yong Shi, ”A WordNet-Based Natural Language Interface to Relational Databases”, Department of Information Technology, Southern Polytechnic State University, 2010.

[15] Belainine Billal, Alexsandro Fonseca and Fatiha Sadat, ”Efficient Natural Language Pre-processing for Analyzing Large Data Sets” in IEEE International Conference on Big Data, 2016.

[16] NLTK Project, ”Natural Language Toolkit” available at http://www.nltk.org/, 2017.
[17] Center for World University Rankings, ”cwurData.csv” available at https://www.kaggle.com/mylesoneill/world-university-rankings, 2016.
[18] Kaggle contributor chuansun76, ”movie_metadata.csv” available at https://www.kaggle.com/deepmatrix/imdb-5000-movie-dataset, 2016.
[19] R. Kohavi and B. Becker, ”adult.csv” available at https://www.kaggle.com/uciml/adult-census-income, 2016.
[20] Kaggle contributor bpali26, ”Web_Scrapped_websites.csv” available at https://www.kaggle.com/bpali26/popular-websites-across-the-globe, 2017.
[21] Fingal County Council, ”Garda_Stations.csv”, available at https://data.gov.ie/dataset/garda-stations/resource/944f3670-5e33-4206-bea5-de2e8d98402a, 2016.
[22] Kaggle contributor GregorySmith, ”vgsales.csv” available at https://www.kaggle.com/gregorut/videogamesales, 2016.
[23] Time Magazine, ”archive.csv” available at https://www.kaggle.com/timemagazine/magazine-covers, 2017.
[24] finintelligence.com, ”fundamentals_dataset.csv” available at https://www.kaggle.com/finintelligence/nasdaq-financial-fundamentals, 2017.
[25] PromptCloud, ”IMDB-Movie-Data.csv” available at https://www.kaggle.com/PromptCloudHQ/imdb-data, 2017.
[26] Practicum repository, available at https://gitlab.computing.dcu.ie/meehank6/practicum, 2017.