Abstracting Natural Language Queries into SQL

Thomas Hagen
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
hagen715@morris.umn.edu

ABSTRACT

Databases are the structural backbone of storing digital knowledge in the modern era. For such a prevalent and important source of information, there is a surprisingly small pool of individuals who posses the skills to access them. In this paper, we cover the core process of making these databases accessible through natural language that anyone would be familiar with. From there, we will then explore two additional modern approaches to abstracting natural language queries into SQL, in the form of implementing keyword-based queries and implementing auto-suggestion queries.

Keywords

Natural Language Interface to Databases (NLIDBs), Natural Language Processing, SQL, Databases

1. INTRODUCTION

Since the original conception of databases and the creation of SQL in the 1960s-70s, there has been interest in the field of natural language processing on how to make these data stores more approachable. Over the years many attempts have been made. Some have been used as industry applications such as PRECISE, an interface for air travel information, and many more were never more than research projects. Despite this, both databases and SQL are still the structural backbone of storing and retrieving digital knowledge in the modern era. Given the popularity and usefulness of databases, it seems as though simple interaction with one would be a more prevalent skill, yet individuals with the know-how are limited in number and often have technical training on how to do so. Nowadays people interact with databases through applications such as Siri, Alexa, or Google, generalized everyday interfaces where no technical skills are required. Research on natural language interfaces to databases (NLIDBs) has also seen more application in specialized fields, such as medicine and biology [5].

In this paper, we cover the core process of making these databases accessible to a broader range of users through natural language. Starting with an open query approach, we will cover the principle pieces of translation by exploring a natural language interface called, "Natural Language Interface to Relation databases" (NaLIR) [3]. With NaLIR, we'll walk through deconstructing the initial English query, representing the query in a programmatic fashion, translating this representation into a structure to be mapped to SQL, and finally converting the intermediary representation into an executable SQL call. From there we move on to a brief overview of a keyword-based implementation as an additive feature to a system such as NaLIR. Keywords will lead us into our final interface through *Discover* and its auto-suggestion approach to natural language interfaces. *Discover* uses a mixture of a keyword-based approach and unrestricted approach to allow users to pose dynamic questions, while ensuring the question can be answered. After our overview of these approaches, we'll wrap up with some concluding remarks.

UMM CSci Senior Seminar Conference, December 2015 Morris, MN.

2. BACKGROUND

2.1 An Introduction to SQL

Structured Query Language (SQL) is a special-purpose programming language designed for managing data held in a relation database management system (RDBMS) [8]. A relational database can be thought of as a collection of spreadsheet-like tables containing rows, each row with a unique key, and columns. For example, you might have a table of books where each row is a book and each column is a book's attribute such as title, author, number of pages, etc. Each book can also hold unique keys to other books, like a reference. In our book table, lets say we hold the author's name, but we also hold the unique key that points us to a row in a separate table of authors. You can think about the unique key like a reference in a paper such as, "See the table of authors, line 7 for information about this book's author." This key points to the row that represents our author, and from there we can see all of the books they have published, including the one we were originally at. By sharing keys, we end up with a web of relationships between rows and tables where the more tables we add, the more complex relationships we form. In order to retrieve any useful information out of all of it we must use some formalized series of commands that can make use of the inherent structure of the database.

SQL gives users a way to add, modify, initialize, and query databases. For the purposes of this paper we will be focusing on the limited scope of querying. Queries (question asking) are denoted by the keyword "SELECT" followed by compo-

```
SELECT Authors.AuthName, Books.BookName,
       Books.BuyDate, Authors.NumOfBooks
 FROM Books
 INNER JOIN Authors
 ON Books.AuthName=Authors.AuthName;
```

Figure 1: Example Inner Join

nent specifications that specify what tables to select from and how to restrict the selection. A sample query might be as follows [8].

```
SELECT *
 FROM  Books
 WHERE price > 100.00
 ORDER BY title;
```

The star after select says that we want every column from our *Books* table. By using the WHERE keyword, we can say that out of all of those books, only give us books whose *price* is over 100. WHERE is used to restrict the information returned to just a subset that meets a condition. Finally, we ask to return the selected items alphabetically ordered by title using ORDER BY, where *title* is a column in the table *Books*.

When two tables each have a column with the same name, it is possible to ask for data from both tables limited by some restriction on the shared column. This operation is called a JOIN in SQL. Lets say there are two tables, one of authors and one of random books on a shelf. Both tables contain the column "AuthName" that is a unique identifier for the authors. If we wanted to retrieve every book's name, author's name, and date when the books were bought, we could use a JOIN as in Figure 1.

Notice above that we used the keyword INNER JOIN. This is because there are four different types of joins and each specifies a different condition. Inner join, also known as simple join, returns all rows from multiple tables where the join condition (the line after the ON keyword) is met [8]. Inner join looks at two columns, Books.AuthName and Authors.AuthName, and takes the rows that have AuthName's that are in both lists. So now we've got a list of authors and books, and most importantly, each row now has a unique AuthName. Now from this list, inner join returns the columns asked for, in our case that is all of the books names and buy dates (Books.Bookname, Books.BuyDates), and it also returns all of the authors names and their number of books (Authors.Authname, Auth.NumOfBooks). Figure 2 shows two tables and the resultant table of an inner join on them.

The other three main types of joins are Left Join, which returns all the rows from the first table (the table after FROM) as well as the matching rows. Right Join does the same as left join except it returns all of the rows from the second table instead of the first. Full Join returns all rows whether they match or not. By using these four types of joins, SQL can mimic certain types of questions that may be easy to phrase in English but hard to think about programatically.

Books Table:

BookName	AuthName	BuyDate	Value
1984	George Orwell	10.10.2009	$10
Green Eggs a..	Dr. Seuss	4.15.2001	$35
How To Kick	Gill Jeffers	6.17.1994	$5

Authors Table:

AuthName	Age	NumOfBooks
George Orwell	107	31
Phill Greggs	41	7
Dr. Seuss	141	74

Inner Join:

AuthName	Bookname	BuyDate	NumOfBooks
George Orwell	1984	10.10.2..	31
Dr. Seuss	Green Eg..	4.15.2..	74

Figure 2: Inner Join on Tables

2.2 Types of Natural Language Analysis

All natural languages are based around inherent rules that dictate sentence structure, word meaning, and subject/object relationships. Transforming a natural language query (NLQ) into something machine-understandable means distilling out ambiguity as well as retaining initial intent. In order for a machine to understand any presented NLQ, we must deconstruct it into its fundamental pieces in such a way that it can be systematically processed. The commonly accepted representation is as a tree structure where each leaf holds a word, and the internal nodes hold representational parts of a sentence that relate words and word groupings to one another. This data structure as it is applied to a sentence is known as a linguistic parse tree. To build this tree, we have to collect both information about the individual words (morphological and lexical analysis) and information about the sentence structure (syntactic analysis).

2.2.1 Morphological Analysis

Morphological analysis is used to represent the meaning and grammatical features of individual words [5]. Each individual word's meaning is modified by its prefix and suffix so morphological analysis splits words into their prefixes, roots, and suffixes. With the root, words can be identified easily and mapped against existing bodies of knowledge for their relations to other words as happens in lexical analysis.

2.2.2 Lexical Analysis

Lexical analysis involves understanding the properties of a word free of its context. In this step of the process, the natural language query is broken down into individual words. Each word is then associated with related information and "tags" such as its part of speech (noun, verb, adjective), synonyms, antonyms, homonyms, and other relevant information. In order to appropriately "tag" each word with its underlying properties, a source of pre-existing knowledge must be used. This source is referred to as a lexicon. There are many different lexicons depending on subject matter, language, and context. Take for example WordNet [4], a large lexical database. WordNet is considered a "Universal Lexicon" because it is not limited to a specific subset of words. A "Domain Lexicon" in contrast, is a specialized set of word relationships about an in-depth field, such as a "Pathological Viruses" domain lexicon containing accurate medical terminology relating to pathogens.

2.2.3 Syntactic Analysis

Syntactic analysis uses the structure of the sentence and the order of the words to determine the intent or specific

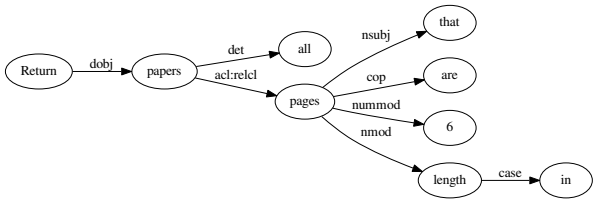

Figure 3: Dependency Tree for NLQ

meanings of the individual words in context to the query as a whole. In order to determine the intended syntactic structure of the sentence, a syntactic parser is used. Feature-based Context-Free-Grammars (FBCFGs) generate sets of features based on the properties of each word. With a wider range of understanding, past just a word's part of speech, FBCFGs use this information to derive internal word relationships. Neural networks look at large quantities of sentences to try and form inherent relationship rules. In this process, they are trained on a large number of sentences that already have the parse specified in order to learn associated patterns with a given query. Once a neural network is trained, it can then be applied to sentences for which the syntactic parse is unknown. For the purposes of this paper we will not be looking at them in depth, but more information can be found in the citations [2,3].

3. NALIR: AN UNRESTRICTED APPROACH TO NLQS

There are a multitude of different approaches to convert a NLQ to SQL with many of them being distinctly different from each other, but all of them are composed of three key components: a linguistic component, an intermediary representational form, and a database component to handle database representation and information retrieval [5]. In order to explore these three pieces, we will first focus on an approach by researchers at the University of Michigan called NaLIR [3]. This user interface takes in an unrestricted natural language query, translates it into SQL, and returns the query results, allowing users to interactively respond and adjust the query as necessary. By allowing an unrestricted vocabulary, NaLIR is not fixed to a particular set of materials as it implements a universal lexicon. This means it may be applied a database without prior knowledge of the subject of the material being queried. In the remainder of this section, we will outline the key steps of NaLIR's process.

3.1 Sentence Decomposition

Once the user has initially entered their query into the interface, the first step of translating natural language is sentence decomposition. The core of NaLIR's method of sentence decomposition lies in how we represent sub-relationships between individual word pairs. As an example, in the sentence "I kicked the ball.", the words "I" and "kicked" share a nominal-subject relationship between the subject "I" and the verb "kick". This process of decomposition is carried out in NaLIR by the Stanford Syntactic Parser. The Stanford Parser represents these natural language relationships using a fixed collection of 44 hierarchical relationships that one word may have to another. For example, parsing the sentence, "Return all papers that are 6 pages in length" gives you the resulting relationship pairs [2]:

```
root(ROOT, Return)
determiner(papers, all)
direct_object(Return, papers)
nominal_subject(pages, that)
copula(pages, are)
numeric_modifier(pages, 6)
acl:relcl(papers, pages)
case(length, in)
nominal_modifier(pages, length)
```

Each line is composed of three key pieces of information formatted as "A(B, C)" in which A is how C relates to B. Looking back at our last line, we can now read it as 'length' nominal modifier of 'pages'. This collection of relationships make up the fundamental structure of a dependency tree, the next form of intermediate representation. As the name implies, a dependency tree is a representational tree diagram formatted to show how the words in a sentence depend on one another. The nodes of the tree represent words, while the edges are derived from the relationships generated by the parser, where the child node is related to the parent node by the named edge. Our simple example query is diagrammed out into a dependency tree in Figure 3 above.

Node Type	Corresponding SQL
Select Node	SQL Keyword: SELECT
Operator Node	an operator, eg. =, >=, !=
Function Node	an aggregation function eg. AVG
Name Node	a table name or column name
Value Node	a value under a column
Quantifier Node	ALL, ANY, EACH
Logic Node	AND, OR, NOT

Table 1: NaLIR Node Classification

3.2 Classifying Dependency Tree Nodes

Once a dependency tree has been generated, the next step of the process is to attempt to map nodes from the dependency tree to reasonable query pieces. NaLIR's approach divides query pieces into 7 different classifications [3] as listed in Table 1. NaLIR attempts to put each node of the dependency tree into one of these classifications; if a node does not fit into a classification, it is typed as non-applicable and is deemed non-essential to forming the final query. As an example, let's take our query, "Return all papers that are 6 pages in length" and match it against a database that has authors, conferences, papers, and other information about technical papers. Our query could be deconstructed into the following mapping strategy in Figure 4 on the following page.

The mapping strategy shows the words from the original query that have been found to relate to either an element of the database being queried, or some piece of SQL language. In the lines above, 'return' was classified as a Select Node corresponding to the keyword SELECT, 'papers' was classified as a Name Node corresponding a table named papers, 'all' was classified as a Quantifier Node with the quantifier ALL, and so on. Both 'that' and 'in' were classified as not relevant to the query and marked with N/A. The process to

```
return (SELECT NODE: SELECT)
papers (NAME NODE: papers)
all    (QUANTIFIER NODE: ALL)
pages  (NAME NODE: pages)
that   (N/A)
are    (OPERATOR NODE: =)
6      (VALUE NODE: papers.length)
length (NAME NODE: length)
in     (N/A)
```

Figure 4: Dependency Tree Node Classifications

generate these classifications is comprised of multiple levels of analysis applied to the nodes of the dependency tree. Previously applied lexical analysis utilizing WordNet provides semantically similar words and synonyms for any given node. With this broadened range of meaning, similarity functions [7,8] can be applied to compare nodes to database elements. Using the same information, we can also apply a similar spelling check to compare words of the query to database values and nodes. This is useful for finding shared naming conventions and potentially identifying Name and Value nodes. Given this restricted set of items to compare to, it is reasonable to assume a pseudo-limited vocabulary without actually implementing constraints on queries. Operations such as querying a database about a subject not pertinent to its contents will yield little information, as the words of your query are unlikely to relate to values of the database. Both spelling and meaning similarity are mathematically calculated, and then the max of the two is taken and compared against a predefined threshold to evaluate whether a node maps to a value or SQL component.

3.3 Query Tree Generation

Query trees are intermediates between natural language sentences and SQL statements. A query tree is comprised of SQL component pieces as defined earlier. Once each node of the original linguistic parse tree has classified, we now have a query tree that may or may not be translatable into a valid SQL query. In order to validate the query tree, a set of rules define the possible structures of valid query trees. If the current query tree could not be generated from these rules then the tree must be reformulated. This is done through repeated subtree movement operations to generate new trees.

A subtree move is moving one subtree up or down the tree, so that the root of the subtree becomes the original root's parent, and the original root becomes a child of the subtree. First, the algorithm performs one subtree move and generates all resulting trees from that move. Then with the set of new trees, it evaluates the validity of the tree compared to the parent tree. The first validation is against our set of rules to calculate the tree, also known as a grammar. Trees with higher numbers of nodes that cannot be generated by the set of rules are deemed less desirable. The second validation of a given tree can be numerically quantified as follows. Let T be a parse tree, in which each node nt maps to a database element v_i. Let $valid(nt_i, nt_j)$ be the set of all the pairs where nt_i is an ancestor of nt_j and no value-mapped node exists between nt_i and nt_j. Given the relevance $w(p(v_i, v_j))$ between v_i and v_j, the score of T is defined as follows [3]:

$$\text{score}(T) = \prod_{\text{valid}(nt_i, nt_j)} w(p(v_i, v_j))$$

In English, to calculate the 'validity score' of a given query tree, we look at every pair of nodes in the tree with a direct child/parent relationship where both nodes map to some element of our database. Once we have every one of these pairs identified, we find their relationship weight to each other from a database graph and then multiply all of their weights. For these weighted values, NaLIR defines a pre-constructed directed database graph of the queried database. There are two types of nodes, defined as relation nodes and attribute nodes where relation nodes represent tables and attribute nodes represent columns. There are also two classifications of edges within the graph, projection edges and join edges. Projection edges are between a table name and its columns, whereas join edges, following the concept of Join from SQL, are between shared columns across tables. These edges are weighted between 0 and 1 where 0 is low correlation and 1 is high correlation [3]. If the generated query has a lot of pairs that are closely related to one another from the database's perspective, then our score will be high.

NaLIR's third way of judging query tree validity is by looking at how many generations away from the original tree this one is, thereby how closely this version of the query tree compares to the originally given syntactic parse tree. In the NaLIR implementation of these scoring systems, the number of valid nodes is the most significant factor of the final score, followed by node relationships, and then generational difference [3]. The trees with the highest scores are then presented to the user as a list of potential English queries to pick from.

3.4 SQL Generation and Execution

Once a user-selected final query tree has been chosen, the conversion to SQL can take place. The majority of the work is done at this point and from here forward, nodes are mapped to the corresponding elements and SQL functions. Function nodes (AVG, SUM) or quantifier nodes (ALL, EACH) denote subqueries called blocks. Any reasonably complex query will contain one or more blocks within it. A block is formally defined as a subtree rooted at the select node, a name node that is marked "all" or "any", or a function node.

The block rooted at the select node is the main block, which will be translated to the main query. Other blocks will be translated to subqueries. When the root node of a block $b1$ is the parent of the root node of another block $b2$, we say that $b1$ is the direct outer block of $b2$, and $b2$ is a direct inner block of $b1$. The main block is the direct outer block of all the blocks that do not have other outer blocks. After identifying blocks, we can outline the general building process of the final SQL query. Starting with the select node, we add nodes that map to columns in the database after the SELECT keyword. Nodes that map to values and any related operation nodes are added under the WHERE keyword. Finally, child/parent nodes that represent columns are translated into JOIN conditions and added. A query similar to, "Return any published books by the author with the greatest number of published conference papers" would produce resultant SQL code containing a block such as Figure 5 .

```
SELECT PublishedBooks.BookName
  FROM ConferencePapers, PublishedBooks,
    (SELECT MAX(paperNumber)            --BLOCK
      FROM ConferencePapers.Authors)    --BLOCK
    AS authGreatestPapers        <-- BLOCK NAME
  WHERE authGreatestPapers.Name = Papers.Author
  AND Papers.Author = PublishedBooks.Author
```

Figure 5: Example of a Block

4. KEYWORD BASED QUERIES

Another approach that has been taken to the problem of processing natural language is to add keyword recognition to previously implemented models as a method to answer more direct and simple questions. Using a keyword based implementation does not replace a full natural language interface, but instead provides another way to understand queries. In general, keyword implementations are built on the idea of matching given keywords against a representational body of words from the database. As an example, databases will often contain metadata, or information about the data within them, that can be retrieved and processed into a usable set of information. If the default mapping of a query cannot be created using the rules of the underlying system, the keyword based agent will attempt to apply a set of rules as to generate a limited SQL response in place of returning nothing. In one given approach developed by Shah, Axita, et al. [6], the set of conditions and actions for keyword queries are limited to just three responses defined as follows. If the keyword exists as a table tag, add it after FROM. If the keyword exists in the name or description attribute tags, add it after SELECT. If the keyword does not match a given tag in the knowledge base, attempt to use the keyword as a value placed after WHERE condition. Consider a farm owned database where an example query that could be converted to a keyword based query is the simple phrase, "agriculture data". Using a general mapping approach, there is no word to be mapped to SELECT, FROM, or any functional nodes. But, applying the rules above, this query can be translated into:

```
SELECT *
 FROM Crop_Data
```

We have two keywords, "agriculture" and "data" that the rules may apply to. Assuming the words "agriculture" and "data" did not directly match any table, column, or value names, we can apply similarity techniques as defined earlier to check lexical and spelling similarities between table, column, and value metadata against our keywords. Through this process, we might find high similarities between the table name "Crop_Data" and the keywords due to "agriculture" and "crop" synonym relationships, as well as exact spelling of "Data" between the keyword "Data" and the second half of the table name "Crop_Data". Once a table name is identified, it is added after the FROM in our call. Since all keywords have been evaluated but there is no specified material to return, * is defaulted after SELECT to return all columns. Keyword queries are often inaccurate due to their limited rule set and often times are only used as a backup.

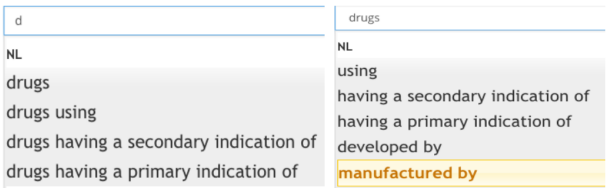

Figure 6: Auto-suggestions being generated

5. AUTO-SUGGEST QUERIES

Designed to bridge the gap between keyword based search and fully unrestricted database query languages, auto-suggestion systems such as Thomson Reuters *Discover* offers word-by-word corrections and suggestions while the user is forming their query [7]. In contrast to the NaLIR from earlier, *Discover* is a domain dependent system, meaning that it requires and utilities prior knowledge of the contents of the stored data in order to apply relational cases specific to the field.

5.1 Question Understanding

Discover's lexical parsing is done through a feature-based context-free-grammar (CFG). A context-free-grammar at its core is a system of rules that define how a sentence must be constructed. A rule may look like, "VP -> V NP", indicating that a verb phrase (VP) must consist of a verb (V) and noun phrase (NP). With these rules, a CFG can start by looking at an entire sentence and see if the parts of the sentence could be mapped out with these rules. Since *Discover* relies on foreknowledge of what type of information it is querying, it can use a domain-specific lexicon. Lets assume that *Discover* is querying a database containing technical information on drug patents. Words that could be commonly used in queries such as the word "headquartered" when referencing company locations can be given a special set of relationship rules to follow. The query, "drugs headquartered in the US" makes sense structurally, but thanks to prior knowledge, the CFG has a rule that the verb "headquartered" cannot have "drug" as a subject. Instead, it must have a company name or the word "company". If there was no prior knowledge of the database, such as in NaLIR, implementing these rules would be impossible since "headquartered" is never guaranteed to only be talking about companies.

With these implicit relationship rules, auto-suggestions can be generated based on what the rules say could potentially be the next part of the sentence, as seen in Figure 6. Starting with a query "d", *Discover* searches its lexicon for words/phrases that start with "d". From this list, a user must either select a presented suggestion or continue their query segment by appending more letters. By selecting the word "drugs", *Discover*, again using its lexicon, now references the rules for the root word "drug" to determine the set of valid phrases follow the word "drugs". This chain of auto-suggestions results in a list of known phrases that are highly likely to be syntactically correct for a query.

In a case where the user is familiar with constructing step-by-step queries, they may type out a full query and ignore the auto-suggestion. If this is the case, *Discover* first tokenizes the words of the query through morphological analysis, and then starting with the first token, it attempts to match the shortest run of tokens with a suggestion. Had

```
Natural Language Query: drugs developed by Merck

FOL: all x.(drug(x) ->
     (develop_org_drug(id0,x) & type(id0,Company)
     & label(id0,Merck)))

SQL: SELECT drug.*
       FROM drug
       WHERE drug.originator-company-name = 'Merck'
```

Figure 7: NLQ, FOL, and SQL

the user typed out, "drugs developed by Merck", the first word "drugs", tokenized to "drug", would match the suggestion "drugs". After "drugs" was processed, the next token is "developed", which does not match a suggestion, so *Discover* adds the next token "by". The two tokens "developed by" together match the suggestion "developed by" and are added after "drugs". After fully processing all tokens, the final set of suggested segments would be, "drugs, developed by, Merck" [7].

5.2 FOL Parsing and Translation

Discover uses first order logic (FOL) [1] as an intermediary representation between natural language and SQL. FOL performs a similar task as the node-mapping of NaLIR and allows the query to be broken down into parts mapped to correct SQL attributes. The translation of our query into FOL can be broken down into three steps. First, the natural language query is implicitly translated to FOL representation directly based on the set of rules that generated the natural language queries structures. Since the original query was constructed from a set of rules, the relationships between suggested segments are already defined and easily made into an FOL representation, similar to a dependency tree in structure. In a dependency tree the nodes are singular words and the edges are grammatical relationships based on implicit rules of English. In an FOL representation, the nodes are pre-defined suggestion segments which may contain more than one word, such as the segment "developed by". The edges, also known as predicate calculus, are a mix of universal English rules as well as rules specific to the database, such what suggestion segments can follow or precede the segment "headquartered".

Second, with the FOL representation, we can generate a syntax tree using another generic parser such as ANTLR, the one utilized by *Discover*. The FOL parser takes in the FOL representation of the query and the grammar. A grammar is the set of all rules used to generate the query and the lexicon used. With these pieces of information, the parser will attempt to determine a syntax tree.

Third, we do an in-order traversal of the syntax tree and push all logical conditions and logical connectors onto a stack. Once the tree is fully traversed, we pop the elements of the stack to build the query constraints. Finally, elements are mapped to their corresponding attributes in the database. The example NLQ, "drugs developed by Merck" would return the FOL and subsequent SQL Query as seen in Figure 7 [7].

6. EVALUATION AND CONCLUSION

In user tests with NaLIR, 90% of user asked questions were evaluated correctly. In cases where user feedback was required, the main point of failure was the translation from dependency to query tree. The natural ambiguity of questions lead to users not being able to accept a query tree that could reflect their initial query. The other steps of the process, dependency tree parsing and query-tree to SQL translation posed no problems for users.

Discover averaged around 80% to 90% recall depending on the test data it was applied to. With 60 grammar rules and approximately one million lexical entries, *Discover* claims comparable precision and recall to state-of-the-art question answering systems. Both of these systems are just two of many approaches taken in the task of converting natural language queries into SQL, and both are comparable to modern standards in their fields of application. The key steps of linguistic breakdown and analysis, intermediary representation, and SQL conversion as seen in the unrestricted approach "NaLIR", keyword implementation approaches, and the auto-suggest approach "Discover" show promising potential for future optimization.

7. ACKNOWLEDGMENTS

Thank you to Nic Mcphee for guidance and revision, KK Lamberty for structural layout, and Skatje Myers and classmates for feedback.

8. REFERENCES

[1] J. Barwise. An introduction to first-order logic. *Studies in Logic and the Foundations of Mathematics*, 90:5–46, 1977.

[2] M.-C. De Marneffe, B. MacCartney, C. D. Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454, 2006.

[3] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[4] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[5] M. N. Nihalani, S. Silakari, and M. Motwani. Natural language interface for database: a brief review. 2011.

[6] A. Shah, J. Pareek, H. Patel, and N. Panchal. Nlkbidb-natural language and keyword based interface to database. In *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*, pages 1569–1576. IEEE, 2013.

[7] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, et al. Tr discover: A natural language question answering system for interlinked datasets. In *The 14th International Semantic Web Conference*, 2015.

[8] Wikipedia. Sql — wikipedia, the free encyclopedia, 2016. [Online; accessed 9-October-2016].