NATURAL LANGUAGE INTERFACES TO DATABASES

Yohan Chandra

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2006

APPROVED:

Rada Mihalcea, Major Professor
Yan Huang, Committee Member
Robert P. Brazile, Committee Member
Armin R Mikler, Graduate Coordinator
Krishna Kavi, Chair of the Department of Computer Science and Engineering
Oscar Garcia, Dean of the College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse School of Graduate Studies
Natural language interfaces to databases (NLIDB) are systems that aim to bridge the gap between the languages used by humans and computers, and automatically translate natural language sentences to database queries. This thesis proposes a novel approach to NLIDB, using graph-based models. The system starts by collecting as much information as possible from existing databases and sentences, and transforms this information into a knowledge base for the system. Given a new question, the system will use this knowledge to analyze and translate the sentence into its corresponding database query statement. The graph-based NLIDB system uses English as the natural language, a relational database model, and SQL as the formal query language. In experiments performed with natural language questions ran against a large database containing information about U.S. geography, the system showed good performance compared to the state-of-the-art in the field.
CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER 1. INTRODUCTION 1

1.1. Definition and Motivation 1
1.1.1. Natural Language Interfaces to Databases 1
1.2. Scope 3
1.3. Linguistic Problems 4
1.3.1. Ambiguity 4
1.3.2. Nominal Compound Problem 5
1.3.3. Grammatical Correctness 6
1.3.4. Conjunction and Disjunction 6
1.4. Advantages and Disadvantages 7
1.4.1. Advantages of NLIDB 7
1.4.2. Disadvantages of NLIDB 8

CHAPTER 2. BACKGROUND 9

2.1. Overview 9
2.1.1. Pattern Matching Systems 10
2.1.2. Syntax Based Systems 11
2.1.3. Semantic Grammar Systems 12
2.1.4. Intermediate Representation Languages 14
2.2. Recent Development 16
6.2.1. Comparison Discussion

CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1. Conclusion

7.2. Future Work

7.2.1. System Improvement

7.2.2. Handling Nested Structures

BIBLIOGRAPHY
LIST OF TABLES

2.1 State Table .......................... 10
2.2 NALIX and the General Syntax Based Approaches .......... 17
2.3 Tokens in NALIX System .................................... 17
4.1 Reducing the Possible Fields of Two Value Tokens .......... 33
4.2 Pre-evaluation Result ........................................... 38
6.1 The Performance of my NLIDB System ...................... 49
6.2 The Performance of Various Systems .......................... 49
6.3 Questions that my NLIDB System Fails to Provide the Correct Answers 50
6.4 Questions that CHILL Fails to Provide the Correct Answers 53
6.5 Questions that KRISP Fails to Provide the Correct Answers 54
6.6 Questions that WASP Fails to Provide the Correct Answers 55
LIST OF FIGURES

2.1 Syntactic Based Parse Tree of "How long is the Mississippi?" 11
2.2 Semantic Grammar Parse Tree of "How long is the Mississippi?" 13
2.3 Intermediate Representation Language Architecture 14

3.1 System Architecture 24
3.2 Relations Extracted from a Database 26
3.3 Geobase Diagram 29

4.1 Plain Graph of City.population 34
4.2 Merged Graph(1) of City.population 35
4.3 Merged Graph(2) of City.population 35
4.4 Completed Graph Based 1 of City.population 36

5.1 Hints and their Relation with a Database and SQL Query 40
5.2 Choosing the Field based on the Shortest Distance 44
5.3 Relations in a SELECT Statement 46

7.1 Subsets in the Hybrid System 59
7.2 A Reduced Parse Tree in the Hybrid System 60
CHAPTER 1

INTRODUCTION

1.1. Definition and Motivation

Since a long time ago, information has played an important role in our lives; most people will try to get the information they need before making a decision. Recently, with the growth of technologies such as computers and laptops, personal digital assistants (PDAs), cellular phones, and the Internet, information can be accessed almost anywhere, at anytime, by anybody, including those who do not necessarily have computer backgrounds [13].

One of the major sources of information are databases. Databases contain a collection of related data, stored in a systematic way to model a part of the world. In order to extract information from a database, one needs to formulate a query in such way that the computer will understand and produce the desired output. However, not everybody is able to write such queries, especially those who lack a computer background.

The most common way for people to obtain information is by asking questions in their natural language [9]. But, without any help, computers cannot understand this language, as it merely represents a sequence of meaningless characters. Nowadays, people attempt to bridge this gap by providing forms; however, this solution is limited and not flexible enough to cover the diversity of questions that a user can ask. Another solution is giving the question to an expert, which is costly and has time constraints. Yet all of these factors only increase the appeal of natural language interfaces to databases (Popescu et al., 2003).

1.1.1. Natural Language Interfaces to Databases

Natural language interfaces to databases (NLIDB) are systems that translate a natural language sentence into a database query (Androutsopoulos et al., 1995). NLIDB can be considered as a classical problem in the field of natural language processing [12]. Although
the earliest research has started since the late sixties [1], NLIDB remains as an open research problem. Several NLIDB systems have also been made for commercial use; regardless, the use of NLIDB systems certainly is not wide-spread and it is not a standard option for interfacing to a database. This lack of acceptance is mainly due to the still large number of deficiencies in the NLIDB system in order to understand a natural language.

A complete NLIDB system will benefit us in many ways. We can allocate the need for an expert to the NLIDB system; thus, anyone may be able to gather the information he or she wants from a database. Additionally, it may change our perception about the information in a database. Traditionally, people are used to working with a form; their expectations depend heavily on the capabilities of the form. NLIDB makes the entire approach more flexible, therefore will maximize the use of a database.

There are many applications that can take advantages of NLIDB. In PDA and cell phone environments, the display screen is not as wide as a computer or a laptop. Filling a form that has many fields can be tedious: one may have to navigate through the screen, to scroll, to look up the scroll box values, etc. Instead, with NLIDB, the only work that needs to be done is to type the question similar to the SMS (Short Messaging System).

As another example, consider a travel agent who may have a lot of customers from various backgrounds; their needs may or may not be the same. A client may ask, ”What is the most popular vacation destination in the US?” This type of question, along with the other most frequent questions, can be answered by building a non-NLIDB system. However, many questions are not in the most frequent category, such as, ”Can you recommend me a place that is not too crowd but also not too quiet?” Of course, we can extend the system to cover more questions, but for a large database it may not be possible to build such a system. Another problem is as the system covers more questions, the complexity increases; therefore, at a certain level the system may not be user-friendly anymore. In addition, after the first question, a client may ask more questions like, ”What are the good places to visit there?” ”How many restaurants are there?” ”Is it full during the weekend?” ”Do we have to book
in advance?”, and many others. Currently a travel agent handles this situation by providing operators to help with the questions. Alternatively using a complete NLIDB system can give a lot of benefits.

1.2. Scope

In this thesis, I propose a new approach to build an NLIDB system using graph-based models. The system works by collecting as much information as possible from the particular database and training sentences. The information obtained will serve as the knowledge-base for the system. With this knowledge, given a new question, the system will analyze and translate the sentence into a database query. A system diagram and more detailed explanations about the methodology will be discussed in chapters 3–6.

My NLIDB system uses English as the natural language, a relational database model for the database, and SQL (Structured Query Language) for the query language. While there are many types of databases, from simple collections of data stored in text files, or in the form of facts and rules in a Prolog system, to more complex ones, such as object oriented model, the most widely used model is still the relational model. In a relational model, the most common query language used is the SQL; hence, it is chosen, although other query languages do exist.

To retrieve data using a SQL query, one should build a SELECT statement. The basic form of this statement is:

\[
\text{SELECT select_expressions FROM table_references WHERE where_conditions}
\]

Although there are more keywords instead of the basic one, they are rarely used and cannot be used alone without the basic form. These keywords such as: \texttt{HAVING, GROUP BY, ORDER BY, JOIN}, etc. A SELECT statement can also be nested with one or more sub-query in it; together with the set operation, like: \texttt{LEFT JOIN, RIGHT JOIN, INNER JOIN}, and \texttt{OUTER JOIN}, they can be very complex and difficult to understand. For example consider the following statements which mean to list the longest river that does not run through Texas.
This thesis focuses on the basic form of a SELECT statement without any sub-query structure. The problem of translating a sentence into a non sub-query SELECT statement certainly is not trivial. The reason are: first, constructing a nested SELECT statement is not an easy task, even for humans. Second, the correctness of the nested statement also depends on the correctness of the lower sub-query. And finally, many questions taking form in a natural language can be translated into a non sub-query SELECT statement. Therefore, it is important to build a system which works very well to address the non sub-query part so that it can lay a good foundation for the greater system. A more detailed evaluation on the system is provided in the chapter 6.

1.3. Linguistic Problems

"Understanding and communicating in natural language is one of the defining problems of AI" (Mooney, 2006). In order to fully understand a natural language, a great deal of knowledge is required such as the morphology, syntax, pragmatic, discourse, and semantic. Because of the difficulties in understanding a natural language, "Ideally an AI (Artificial Intelligence) system would be able to learn language like a human child" (Mooney, 2006).

This section will discuss specific problems related to the NLIDB domain. However, the problems described below are not exclusively attached to the NLIDB, as they may also appear in other fields of NLP, nor are they a complete list of all the problems in NLIDB. The problems listed are the problems that I had to deal with while building the NLIDB system.

1.3.1. Ambiguity

The most common problem in the area of NLP is ambiguity, Inputs are considered ambiguous if there are multiple alternative structures that can be built for them [7]. There are
many types of ambiguity such as part-of-speech ambiguity, word sense ambiguity, syntactic ambiguity, and many others.

While a human can often immediately understand the correct meanings of ambiguous terms in a sentence, a computer system must develop a method to handle these terms. Consider the example: "Through which states does the Mississippi traverse?" For Computer, the term "Mississippi" here is ambiguous since it can be the name of a state or a river. In contrast, a human will immediately know that "Mississippi" here refers to the river because a state cannot "traverse".

More severe cases of ambiguity do occur. In this case, even a human cannot understand the correct meaning because the context itself is ambiguous. Consider the example: "What is the population of New York?", the word "New York" here could be interpreted either as a city or as a state. Unless the questioner specifies the correct meaning, both interpretations are correct.

In this thesis, I am approaching the problem of ambiguity using heuristic methods based on graph models with prior knowledge obtained automatically from a database and training sentences.

1.3.2. Nominal Compound Problem

"A noun phrase can be viewed as revolving around the central noun" (Jurafsky et al., 2000). In English, a noun phrase can consist of both pre-nominal modifiers and post-nominal modifiers, and thus, the meaning of a noun phrase can sometime be hard to predict.

For example, the noun phrase: "The states bordering Texas" could easily be traced by following the words sequentially. However, consider the noun phrase "major river". The "major river" here may be a river which traverses at least several states, a river with certain length or width, or other interpretations.

Because of the difficulty in determining certain noun phrases, in this thesis, some noun phrases that require prior knowledge need to be defined manually during the configuration phase.
1.3.3. Grammatical Correctness

In our daily life, even though we often say something that is grammatically incorrect, other people may still understand what we are trying to say. Consider the example: "states bordering Iowa" In English, a sentence should contain at least a subject and a predicate; thus, the above sentence is not correct. However it still can be translated into a correct SQL query.

Other examples of incorrect grammar are sentences where the subject and verb do not agree, incorrect articles, capitalization errors, punctuation errors, etc. Instead of rejecting the sentence because of this type of errors, my system will accept and translate the sentence into the corresponding SQL query. Obviously, there are limitations to the errors, for example misspelling errors cannot be handled.

1.3.4. Conjunction and Disjunction

In the logic domain, the meaning of conjunction (denoted by AND) is obvious: the output will be true if both inputs are true, while disjunction (denoted by OR) means the output will be true if at least one of the inputs is true. This rule does not always apply in the natural language. Consider the example: "Name all the cities in Texas and Oklahoma." The term "and" here does not mean a conjunction, because a city can only have one state. Instead, it reflects a disjunction, where every city located in Texas or in Oklahoma should be listed.

A conjunction in English is a part of speech that connects phrases, words, or clauses; this part of speech can consist of a single word ("and", "or", "nor", "yet", "while", etc) or multiple words ("either ... or", "not only ... but also", "both ... and", etc). The meaning may vary from stating that both inputs are true, both inputs are false, and a contradiction where one of the input is true and the other is false.

In order to correctly interpret a conjunction or a disjunction, more extensive knowledge about the structure is required. Moreover, some conjunctions or disjunctions cannot be translated into a SQL query without using any sub-query structure. My NLIDB system is
not specifically designed to address the problem of conjunction or disjunction; regardless, some of them can be translated correctly.

1.4. Advantages and Disadvantages

This section discusses advantages and disadvantages of NLIDB, most of them cited from *Natural Language Interfaces to Databases - An Introduction* (Androutsopoulos et al., 1995)

1.4.1. Advantages of NLIDB

(i) No learning required

The main problem for most people that attempt to acquire information from a database is that they have to learn a computer language, which sometime can be difficult. On the other hand, they have been exposed to a natural language since an early age and have used it in daily communication; therefore, we can say that a natural language is already mastered by the user.

(ii) Simple, easy to use

Consider a database with a query language or a certain form designed to display the query. While an NLIDB system only requires a single input, a form-based may contain multiple inputs (fields, scroll boxes, combo boxes, radio buttons, etc) depending on the capability of the form. In the case of a query language, a question may need to be expressed using multiple statements which contain one or more sub-queries with some joint operations as the connector.

(iii) Fault tolerance

Most of NLIDB systems provide some tolerances to minor grammatical errors, while in a computer system; most of the time, the lexicon should be exactly the same as defined, the syntax should correctly follow certain rules, and any errors will cause the input automatically be rejected by the system. In the case of incomplete sentences, most of computer systems do not provide any support.
1.4.2. Disadvantages of NLIDB

(i) Linguistic coverage is not obvious

Currently all NLIDB systems can only handle some subsets of a natural language and it is not easy to define these subsets. Even some NLIDB systems, including mine, cannot answer certain questions belong to their own subsets. This is not the case in a formal language. The formal language coverage is obvious and any statements that follow the given rules are guaranteed to give the corresponding answer.

(ii) Linguistic vs. conceptual failures

In the case of NLIDB system failures, it is often the case that the system does not provide any explanation of what causes the system to fail. Some users may try to rephrase the question or just leave the question unanswered. Most of the time, it is up to the users to determine of what causes the errors.

(iii) False expectations

People can be misled by an NLIDB system’s ability to process a natural language: they may assume that the system is intelligent. Therefore rather than asking precise questions from a database, they may be tempted to ask questions that involve complex ideas, certain judgments, reasoning capabilities, etc, which an NLIDB system cannot be relied upon.
CHAPTER 2

BACKGROUND

2.1. Overview

According to Androutsopoulos et al. (1995), the earliest natural language interfaces to databases (NLIDB) research started in the late sixties. During that time, most of the research has concentrated on one database at a time as the implementation target, therefore they could not be modified to be implemented on other databases. One of the well-known NLIDB system in the sixties is Lunar [16], a system developed for a database which contained information about chemical analysis of moon rocks.

After the earliest research period, more NLIDB systems had appeared. By the mid-eighties NLIDB was a very popular research topic, indicated by the numerous systems were being implemented [1]. During this time, the research focus had changed to the issue of portability, and some systems were even brought to commercial use, though they did not get the expected gain. The main reason for the lack of acceptance is probably due to the difficulties to fully understand a natural language.

More recently the development of NLIDB systems have been greatly influenced by the evolution of natural language processing (NLP) methods [11]. Previously, a lot of NLP systems were developed by implementing a set of rules to address a particular problem. Because building accurate rules is difficult, in particular for those targeting a broad coverage, the main approaches have shifted to implement statistical or corpus-based techniques to automatically learn the knowledge required from a training corpus [11]. The same approach is also applied in NLIDB, and the result shows significant improvements compared to the systems used before.
In this section, typical architectures of NLIDB systems are described. In the following section, several recent particular NLIDB systems will be discussed.

2.1.1. Pattern Matching Systems

Consider this table below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Capital</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>Montgomery</td>
<td>3,894,000</td>
</tr>
<tr>
<td>Texas</td>
<td>Austin</td>
<td>14,229,000</td>
</tr>
<tr>
<td>Alaska</td>
<td>Juneau</td>
<td>401,000</td>
</tr>
</tbody>
</table>

Table 2.1. State Table

A simple pattern matching system may develop a rule like:

pattern: ... 'capital' ... [state]

answer: SELECT capital FROM state WHERE name=[state]

The above rule specifies that if there is a sentence containing a keyword ”capital” followed by a state name, then the related answer is SELECT capital FROM state WHERE name=[state]. The state name can be obtained by looking for a certain value in the table.

Certainly a pattern matching system is not necessarily to be that simple. Instead of just seeking an exact match with the keyword, the system could also consider additional conditions. For example: a single term in a pattern can consist of several features, such as the stemmed form of the keyword, the part of speech (POS), the synonym, the hypernym, the position in the sentence, and many others.

There are two main advantages for using a pattern matching approach. The first advantage is it is easy to implement compared to other systems. The second is it is easy to add or subtract features within the system. However as mentioned [1], a pattern matching system is too shallow and therefore would often lead to bad failures.
2.1.2. Syntax Based Systems

Syntax based systems are based on the idea of extending syntactic parsers with semantic labels [15]. A sentence is parsed using certain grammar rules resulting in a syntactic tree, some of the nodes in the tree are then mapped to their semantic meaning, and these semantic meanings are further combined to produce the corresponding database query.

For instance, figure 2.1 shows a syntactic tree for the sentence "How long is the Mississippi?", obtained using Charniak parser [3]:

![Syntactic Based Parse Tree of "How long is the Mississippi?"](image)

Using predicate logic as the query language, the above parse tree could then be mapped using these rules:

- \((\text{NP} (\text{DT the}) (\text{NNP Mississippi})) \rightarrow \text{const}(B,\text{riverid(mississippi)})\)
- \((\text{WHADVP} (\text{WRB how}) (\text{RB long})) \rightarrow \text{len}(A,B)\)
- \(\text{SBARQ} \rightarrow \text{answer}(A,\text{Goal})\)

Where:
- \(\text{riverid(mississippi)}\) — "Mississippi" as a river
- \(\text{const}(B,\text{riverid(mississippi)})\) — a variable B is attached to "riverid"
- \(\text{len}(A,B)\) — the length of B is A
- \(\text{answer}(A,\text{Goal})\) — a wrapper indicating variable A as the answer

Variable "Goal" in simple sense is something which can contain one or more logic statements. The complete query statement will then be:
answer(A,(len(B,A),const(B,riverid(mississippi))))

The main advantage of using syntax based approaches is that they provide detailed information about the structure of a sentence. A parse tree contains a lot of information about the sentence structure; starting from a single word and its part of speech, how words can be grouped together to form a phrase, how phrases can be grouped together to form more complex phrases, until a complete sentence is built. Having this information, we can map the semantic meanings to certain production rules (or nodes in a parse tree).

Unfortunately not all nodes should be mapped, some nodes have to be left just as they are without adding any semantic meanings. And it is not always clear which nodes should be mapped and which should not. Moreover the same node in different parse trees is not necessarily going to be translated in all the trees. The second problem is a sentence can have multiple correct parse trees, and if all are translated, they may lead to different query results. The last problem is that it is difficult for a syntax based approach to directly map a parse tree into some general database query language, such as SQL (Structured Query Language).

2.1.3. Semantic Grammar Systems

A semantic grammar system is very similar to the syntax based system, meaning that the query result is obtained by mapping the parse tree of a sentence to a database query. The basic idea of a semantic grammar system is to simplify the parse tree as much as possible, by removing unnecessary nodes or combining some nodes together. Based on this idea, the semantic grammar system can better reflect the semantic representation without having complex parse tree structures. Therefore, a production rule in a semantic grammar system does not necessarily correspond to the general syntactic concepts.

Instead of smaller structures, the semantic grammar approach also provides a special way for assigning a name to a certain node in the tree, thus resulting in less ambiguity compared to the syntax based approach. Both of the ambiguities that can occur in mapping a node to
its semantic label and the number of different parse trees which are possible for a particular sentence.

Figure 2.2 shows an example of a parse tree in a semantic grammar system:

![Semantic Grammar Parse Tree of "How long is the Mississippi?"]

**Figure 2.2.** Semantic Grammar Parse Tree of "How long is the Mississippi?"

A complete query statement: \( \text{answer}(A,(\text{len}(B,A),\text{const}(B,\text{riverid(mississippi))})) \)

can be obtained using the following rules:

\[
\begin{align*}
\text{(river\_name (the) (Mississippi))} & \quad \rightarrow \quad \text{const}(B,\text{riverid(mississippi)}) \\
\text{(length\_question (how) (long) (is))} & \quad \rightarrow \quad \text{len}(A,B) \\
\text{river\_question} & \quad \rightarrow \quad \text{answer}(A,\text{Goal})
\end{align*}
\]

The main drawback of semantic grammar approach is that it requires some prior-knowledge of the elements in the domain, therefore making it difficult to port to other domains. In addition, a parse tree in a semantic grammar system has specific structures and unique node labels, which could hardly be useful for other applications. Regardless, there are ongoing attempts to automatically build the grammar rules by obtaining the prior-knowledge based on user interaction [6] or by automatically extracting it from a corpus [8].
2.1.4. Intermediate Representation Languages

Due to the difficulties of directly translating a sentence into a general database query languages using a syntax based approach, the intermediate representation systems were proposed. The idea is to map a sentence into a logical query language first, and then further translate this logical query language into a general database query language, such as SQL. In the process there can be more than one intermediate meaning representation language [1]. Figure 2.3 shows a possible architecture of an intermediate representation language system.

Consider a sentence "What is the capital of the states which the Mississippi river flows through?" Based on the above architecture, the sentence will be parsed using a parser resulting into the related parse tree. This parse tree will be analyzed by a semantic interpreter implementing some mapping rules, and the result is the corresponding logical query statement. Using predicate logic as the logical query language, an intermediate representation system could develop a semantic interpreter that maps the above sentence into the following logical query:

\[
\text{SELECT capital FROM states WHERE state \in (\text{SELECT state FROM rivers WHERE river = "Mississippi")}
\]
answer(Capital,State):-
    liststate(State),
    flow(State,'Mississippi'),
    capital(Capital,State).

Further on, every logic predicate in the answer can be mapped to the corresponding SQL statement:

liststate(State):-
    SELECT name FROM state.

flow(State,'Mississippi'):-
    SELECT state.name FROM river_flow,state
    WHERE river_flow.state=state.name AND river_flow.name='Mississippi'.

capital(Capital,State):-
    SELECT Capital FROM state WHERE state.name=State.

The transformation from a logical query language to a database query language does not need to be made in one step. As an example, an NLIDB system developed at the University of Essex uses a multi-stage transformation process [5]. The first logic query is in the form of \( \lambda \)-calculus, which is then transformed to a first-order predicate logic, universal domain relational calculus, domain relational calculus, tuple relational calculus, and finally SQL.

In the intermediate representation language approach, the system can be divided into two parts. One part starts from a sentence up to the generation of a logical query. The other part starts from a logical query until the generation of a database query. In the part one, The use of logic query languages makes it possible to add reasoning capabilities to the system by embedding the reasoning part inside a logic statement. In addition, because the logic query
languages is independent from the database, it can be ported to different database query languages as well as to other domains, such as expert systems and operating systems [1].

2.2. Recent Development

This section provides a brief overview of three specific NLIDB systems developed recently in different universities.

2.2.1. NALIX

NALIX (Natural Language Interface for an XML Database) is an NLIDB system developed at the University of Michigan, Ann Arbor by Yunyao Li, Huahai Yang, and H. V. Jagadish (2006). The database used for this system is extensible markup language (XML) database with Schema-Free XQuery as the database query language.

Schema-Free XQuery is a query language designed mainly for retrieving information in XML. The idea is to use keyword search for databases. However, pure keyword search certainly cannot be applied. Therefore, some richer query mechanisms are added [10]. Given a collection of keywords, each keyword has several candidate XML elements to relate. All of these candidates are added to MQF (Meaningful Query Focus), which will automatically find all the relations between these elements. The main advantage of Schema-Free XQuery is that it is not necessary to map a query into the exact database schema, since it will automatically find all the relations given certain keywords.

NALIX can be classified as a syntax based system, since the transformation processes are done in three steps: generating a parse tree, validating the parse tree, and translating the parse tree to an XQuery expression. However, as implied in the paper [9][10], NALIX is different from the general syntax based approaches; in the way the system was built: NALIX implements a reversed-engineering technique by building the system from a query language toward the sentences.

A formal query language is rigorous, and the grammar is deterministic, thus every correct input will be guaranteed to get the desired results. Having these properties, NALIX starts from the Schema-Free XQuery language and develops some lexicons/tokens categorized into
starts by analyzing the XQuery Language
2. builds set of tokens from the XQuery Language
3. attach the tokens to the general POS
4. build the production rules
5. build the parser

1. start by analyzing input sentences
2. build the production rules’ labels
3. build the parser
4. attach the semantic meanings to the production rules

<table>
<thead>
<tr>
<th>Token</th>
<th>Query Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Token(CMT)</td>
<td>Return clause</td>
<td>Top verb or wh-phrase (WHNP)</td>
</tr>
<tr>
<td>Order by Token(OBT)</td>
<td>Order by clause</td>
<td>enum set of phrases</td>
</tr>
<tr>
<td>Function Token(FT)</td>
<td>Function</td>
<td>enum set of adjectives and noun-phrases(NP)</td>
</tr>
<tr>
<td>Operator Token(OT)</td>
<td>Operator</td>
<td>enum set of preposition phrases(PP)</td>
</tr>
<tr>
<td>Value Token(VT)</td>
<td>Value</td>
<td>Noun(NN), Noun Phrases(NP), Proper Noun (NNP), Number (CD)</td>
</tr>
<tr>
<td>Name Token(NT)</td>
<td>Basic variables</td>
<td>a non VT noun or Noun Phrase (NP)</td>
</tr>
<tr>
<td>Negation(NEG)</td>
<td>Function not</td>
<td>Adjective &quot;not&quot;</td>
</tr>
<tr>
<td>Quantifier Token(QT)</td>
<td>Quantifier</td>
<td>enum set of adjectives serving as determiners</td>
</tr>
</tbody>
</table>

The system was tested on Timber native XML Database which supports Schema-Free XQuery, and the result shows that for every correctly parsed query the average precision is 95.1% and the average recall is 97.6%. In the case of failures, the users were asked to rephrase the question, and the number of iterations needed to formulate a query acceptable by NALIX is less than 2.

The advantage of NALIX is that the system tries to maintain consistencies with the underlying schema, and therefore it inherits the strength of a formal language. Among these properties are rigorousness, determinism, and the ability to handle nested structures.
However NALIX only makes a little attempt to understand the natural language itself [10]. A user may get confused why two sentences having the same exact semantic meaning but different syntactic structures give different results, e.g. one sentence can be translated while the other is rejected. In the future, as it grows, the system has to move from the white area (where everything is made clear and not ambiguous) to the gray area (where it is forced to face the challenge of understanding a natural language). At that stage, the system will have to compromise between maintaining consistency or having more coverage of natural language subsets.

2.2.2. PRECISE

PRECISE is a system developed at the University of Washington by Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates (2004). The target database is in the form of a relational database using SQL as the query language. It introduces the idea of semantically tractable sentences which are sentences that can be translated to a unique semantic interpretation by analyzing some lexicons and semantic constraints.

According to the terminologies used in PRECISE, a database consists of three elements: relations, attributes and values. A relation is a table name or a relation between tables, while an attribute is a field name, and a value is the value of particular field. A lexicon is a tuple \((T,E,M)\) where \(T\) is a token, \(E\) is a set of possible database elements, and \(M\) defines the relation between a token and the set \((T \times E)\) [12].

A semantically tractable question that defines a valid mapping from \(T\) to \(E\) should fulfill the following conditions:

(i) Sentence Token - Database Element Match

There is one to one match between the tokens \(T\) and the database elements \(E\).

(ii) Attribute Token - Value Token correspondence

Each attribute token corresponds to a unique attached value token.
(iii) Implicit Attributes

Some of the database attributes may not match any of the attribute tokens in the sentence.

(iv) Relation Token - Attribute/Value Token Correspondence

Each relation token corresponds to either an attached attribute token or an attached value token.

By the above definition, the matcher analyzes given inputs by implementing a graph matching approach (Maxflow algorithm) resulting in possible mappings. As an example, consider this sentence: "What are the flights from Boston to Chicago?". The set of tokens obtained from the above sentence are: value tokens - What, Boston, and Chicago; attribute tokens - to, from; and relation token - flight. Using the Maxflow algorithm "Boston" and "Chicago" can be mapped to city.cityname field, "from" can be mapped to flight.fromairport or fare.fromairport, "to" can be mapped to flight.toairport or fare.toairport. Combined with the syntactic information from the parse tree, "from" will be mapped to flight.fromairport rather than flight.toairport. A similar analysis is also applied to "to".

PRECISE was evaluated on two database domains. The first one is the ATIS domain, which consists of spoken questions about air travel, their written forms, and their correct translations in SQL query language. In ATIS domain, 95.8% of the questions were semantically tractable. Using these questions gives PRECISE 94% precision. The second domain is the GEOQUERY domain. This domain contains information about U.S. Geography. 77.5% of the questions in GEOQUERY are semantically tractable. Using these questions gives PRECISE 100% accuracy.

The strength of PRECISE is based on the ability to match keywords in a sentence to the corresponding database structures. This process is done in two stages, first by narrowing the possibilities using Maxflow algorithm and second by analyzing the syntactic structure of a sentence. Therefore PRECISE is able to perform impressively in semantically tractable questions.
As other NLIDB systems, PRECISE has its own weaknesses. While it is able to achieve high accuracy in semantically tractable questions, the system compensates for the gain in accuracy at the cost of recall [13]. Another problem is as PRECISE adopts a heuristic based approach, the system suffers from the problem of handling nested structures.

2.2.3. WASP

Word Alignment-based Semantic Parsing (WASP) is a system developed at the University of Texas, Austin by Yuk Wah Wong (2005). While the system is designed to address the broader goal of constructing "a complete, formal, symbolic, meaningful representation of a natural language sentence" (Yuk Wah Wong, 2005), it can also be applied to the NLIDB domain. A predicate logic (Prolog) was used as the formal query language.

WASP learns to build a semantic parser given a corpus a set of natural language sentences annotated with their correct formal query languages [15]. It requires no prior-knowledge of the syntax, because the whole learning process is done using statistical machine translation techniques.

Statistical machine translation is an approach designed to translate a natural language into another natural language. Viewed from a different perspective, this other natural language can be a formal query language. Let an English sentence (e) be a possible translation of a formal query language (f), every pair of strings (e,f) is assigned a probability Pr(e|f), then the complete formula is:

\[
e^* = \text{argmax } \text{Pr}(e)\text{Pr}(f|e)
\]

The above formula involves two probability distributions, the language model (Pr(e)) and the alignment model (Pr(f|e)).

There are three alignment models in statistical machine translation: word-based alignment models, phrase-based alignment models, and syntax-based alignment models. WASP uses the word-based alignment models for retrieving the probability of Pr(f|e). Suppose f =
\[ f'_1 = (f_1, \ldots, f_J) \text{ and } e = e'_1 = (e_1, \ldots, e_J). \] A word alignment (a) is defined as \( a = (a_1, \ldots, a_J), \forall j=1,\ldots,J, 0 \leq a_j \leq I, \) Pr(\( f | e \)) can then be defined as below:

\[
\Pr(f | e) = \sum_a \Pr(f,a | e) \\
\Pr(f,a | e) = \Pr(J | e) \prod_{j=1}^J \Pr(a_j | a_{1:j-1}, f_{1:j-1}, J, e) \Pr(f_j | a_{1:j}, f_{1:j-1}, J, e)
\]

WASP was evaluated on the GEOQUERY domain, the same domain as PRECISE. GEOQUERY corpus consists of 880 questions in the training set and 250 questions in the test set, which are merged together into one larger data set. Each data set was divided to 10 equal-sized subsets, and standard 10-fold cross validation was used to estimate the system performance. WASP achieved 86.14% precision and 75.00% recall in the GEOQUERY domain. The system was also evaluated on a variety of other natural languages: English, Spanish, Japanese and Turkish. There were no significant differences observed between English and Spanish, but the Japanese corpus has the lowest precision and the Turkish corpus has the lowest recall.

The strength of WASP comes from the ability to build a semantic parser from annotated corpora. This approach is beneficial because it uses statistical machine translation with minimal supervision. Therefore, the system does not have to manually develop a grammar in different domains. Moreover, while most of NLIDB systems use English as their natural language, WASP has been tested on several languages.

In spite of the strength, WASP also has two weaknesses. The first is: the system is based solely on the analysis of a sentence and its possible query translation, and the database part is therefore left untouched. There is a lot of information that can be extracted from a database, such as the lexical notation, the structure, and the relations within. Not using this knowledge prevents WASP to achieve better performances. The second problem is that the system requires a large amount of annotated corpora before it can be used, and building such corpora requires a large amount of work.
CHAPTER 3

SYSTEM OVERVIEW

3.1. Basic Concepts

I view the problem of translating a natural language sentence to a formal query language as the problem of filling correct semantic meanings in a SELECT statement. Since the syntax is fixed, if we can design algorithms to fill every slot properly (select_expressions, table_references, and where_conditions), a valid SQL (Structured Query Language) statement can be built.

The main idea behind the system is to analyze and understand the SELECT statement in a SQL query language. The fundamental constituents of a SELECT statement are shown below:

```
SELECT select_expressions
FROM table_references
WHERE where_conditions
```

While:

- select_expression: a field in a database or a field wrapped by functions
- table_reference: a table in a database
- where_condition: a field equal to a value or a field equal to field

Regardless of the complete structure, these elements are the core of a SELECT statement and by themselves already cover a large subsets of a natural language. Relating this structure to a relational database model, there are direct relations of elements in a SELECT statement with terms in a database, such as: tables, fields and values. Moreover, every term in a SELECT statement should be as specified in the database.
The relations between a SELECT statement and a sentence are not as clear as the relations between a SELECT statement and a database. Consider a sentence ”How long are the rivers in Texas?”., which can be translated into:

```
SELECT length FROM river,river_flow
WHERE river.name=river_flow.name AND state='texas';
```

Several terms like ”river” and ”Texas” are explicitly specified in the sentence, so does ”long” which has the same meaning with length. But, a condition ”river.name=river_flow.name” cannot be found anywhere in the sentence. This condition is obtained from the database structure, and it is needed to connect the river and river_flow tables. The explicit terms found in a sentence will later be called ”hints”: terms in a sentence that also appear in a database, and therefore should be included to form the SELECT statement.

Let us focus on the select part of a SQL statement that is meant to choose rows in a table to be displayed as the result of a query. The select part accepts select expressions as inputs, and a select expression contains a field that may or may not be wrapped with a SQL function such as MAX, DISTINCT, COUNT, and others. Using information extraction technique by automatically constructing patterns derived from an annotated corpora (a file contains of sentences and their correct SQL translations), these select expressions can be obtained. More detailed explanations about the algorithm will be discussed in chapter 4.

If we can determine the correct translations of select expressions and where conditions, finding table references for the from part is trivial. The easiest way is by keeping track of every table occurrence in the select expressions and where conditions in an array. Only new occurrence is allowed to be added, and therefore, in the end, every element in the array is a table reference.

The where conditions can be divided into two categories according to their database meaning. The first part (field=value) refers to a specific value in a database and the second part (field=field) represents a relation in a database. A value can be traced from the sentence since it should be explicitly stated. And once we have the corresponding field
in a select_expression and the values in where_conditions, relations can also be obtained by finding a correct path from fields in the where_conditions to a field in the select_expression. More detailed explanations about the algorithm will be discussed in chapter 5.

3.2. System Architectures

Figure 3.1 shows the complete diagram of the system:

![Diagram of System Architecture](image)

**Figure 3.1. System Architecture**

The system can be divided into two parts: The knowledge extraction part (step 1 to 4) and the evaluation part (step 5 and 6). After the completion of the knowledge extraction part, both parts can work separately. This way, the system can work efficiently without having to regenerate the knowledge-base every time a new sentence is given. The knowledge extraction part needs to be re-executed only if a change occurs in the database or corpus.

3.2.1. Obtaining Knowledge from a Database

In the system diagram (figure 3.1), this step starts from "Database" until the generation of "Token" and "Relation" (step 1 and 2). Everything in a database can be considered useful, from tables which contain fields, and fields which contain specific values. In order
to maintain the same structure as the database, tokens are divided into three categories respectively: table tokens, field tokens, and value tokens. A token may overlap in the same or different categories according to its entries in a database. For example: a token ”Mississippi” belongs to the value tokens: river.name and state.name, and the token ”state” belongs to a table token and a field token.

A relation can be defined as two fields that connect at least two different tables. In designing a database, people already have a concept in mind of what the relations will be. In an ER (Entity Relational) diagram, a relation is denoted by a line from a certain field in a table to another field in a different table. Instead of the formal relations (relations that are described by the user), there are other relations that occur accidentally due to the world model. While this type of relation may not be applied to every value in both fields, at least one value can connect the two fields. These relations should also be stored, because there are possible questions which may be derived from the situation. For example: the term ”Missouri” which in general is used to map a relation between the state.name and city.state. It can also be used to map a relation between state.name and river.name, and one possible question for this situation is ”List rivers which have the same name as its state”.

All relations in a database can be obtained by comparing all the values. Each value that has an identical entry in another field represents a candidate relation. There are weights attached to every relation: a single match has the weight of 1, and multiple matches will add the value. And thus, a rare relation can be identified because it has less weight. A match to a key either a primary key or foreign key has higher weight of 1.2. Therefore, the overall relations will point more toward the key fields. The same case also happens in a database; in the sense that a master field is used more often than a transaction field. Finally, every relation in a table is normalized by the highest relation of that table. This step is done to prevent the gap due to relations that occur too frequent, which may cause some relations to be overused, while others are never used.
3.2.2. Obtaining Knowledge from Sentences

Since the goal of input sentences in a natural language interfaces to databases (NLIDB) system is to obtain information from a database, they have certain "meanings" to the particular domain. Therefore, they can be used to form a SELECT statement. However, not every word carries the same degree of meaning, some words are meaningful by themselves, while others have to be combined to be meaningful.

The first step to obtain knowledge from sentences is: gives a tag to every word in the input sentence with its POS (part of speech) using Brill’s tagger [2]. All function words are removed by eliminating words that belong to certain POS categories, e.g: DT - determiner, TO - to, punctuation marks, etc. In common these words are called stopwords - words which are so common that they are useless.

The second step is to compare every word in a sentence to the tokens obtained from the database. First, both words are stemmed using Porter Stemmer [14] and then compared. Every match is stored, because it indicates a direct connection between the sentence and the database.

Figure 3.2. Relations Extracted from a Database

Figure 3.2 illustrates the relations obtained automatically from a database. The thick lines denote relations between key fields in a database, and the dashed lines denote other relations that occur due to identical entries in a database.
Because my NLIDB system does not use any syntactic parser; hence, a noun phrase has to be manually searched considering only the bigram (sequence of two words) matches. Every time a noun (NN) is found, the preceding word is taken, and then a bigram model is built. All bigrams are compared to the tokens, and all matches are stored.

3.3. The Database

There are two commonly used databases in the NLIDB domain: ATIS (a database that contains information about air travel system) and GEOQUERY. The database used for my system is GEOQUERY; a database consists of basic information about the U.S. geography, including: the state, population, area, city, river, lake, mountain, highest and lowest points along with their elevations [17]. GEOQUERY was developed by the UT Austin group led by Prof. Mooney. As mentioned in the previous papers [13][15], GEOQUERY is a more challenging domain compared to ATIS.

The GEOQUERY can be divided into three parts: the database, the query language, and the corpus. The database called Geobase contains about 800 Prolog facts stating relational tables for basic information about the U.S. geography. The query language written in Prolog consists of basic predicates and meta-predicates necessary to formulate a query. The training corpus has 880 questions paired with their correct Prolog queries and the testing corpus has 250 questions. An example of a question from the training corpus is given below:

```
parse([can, you, tell, me, the, capital, of, texas, ?])
answer(A, (capital(A), loc(A, B), const(B, stateid(texas))))
```

The GEOQUERY is built using Prolog; therefore, it has to be translated to a relational database model before it can be used. My NLIDB system uses MYSQL as the database system; hence, it will be transformed into a MYSQL format. The Geobase does not specify enough information needed to built a relational database model, such as field names and data types for each field. These elements were developed manually while still maintaining the
integrity of the database. Most of the fields are assigned to a string data type (VARCHAR), and numbers are assigned to INT or DOUBLE respectively to their size.

Several tables in Geobase are not in the BCNF (Boyce-Codd Normal Form), therefore these tables have to be modified into a BCNF. For example, consider a fact:

river('cimarron',965,['new mexico','kansas','oklahoma'])

The above fact means a river named "cimarron" with length of 965 flows through these states: "new mexico", "kansas", and "oklahoma". In a BCNF database, this information cannot be kept in one table. Therefore, this table will be split into two tables. A master table (River) which has two fields: "name" and "length", with "name" as the key. A child table (River_flow) which has two fields "name" and "state".

Instead of the nested tables problems, there are also inconsistencies between value entries that refer to the same entity in the Geobase. Consider two Prolog facts:

river('delaware',451,['new york','pennsylvania','new jersey','delaware'])
highlow('pennsylvania','pa','mount davis',979,'delaware river',0)

The first table specifies information about a river named "delaware" and the second table mentions that the lowest point in Pennsylvania is the "delaware river". Both terms refer to the same entity delaware as a river, but each entry uses different names. Although the reason is to help the user to read easier, they are incorrect. Because in a database system, they mean separated entities. In order to fix this, a single name "delaware" is used. The complete structure of the Geobase in a relational model is shown in figure 3.3.
Figure 3.3. Geobase Diagram
The first element of the basic SELECT statement is: `SELECT select_expressions`, which is used to specify the target field of a query. In a sentence, this element refers to the subject/object that is asked by the user. However, it is difficult to develop certain rules to identify the `select_expression` in a sentence. First: while most of the time it is located in the beginning, it can also be placed in the ending of a sentence. Second: Although some subjects/objects are closely related to certain words (e.g: what, how many, mention, etc), given fields in a database, it is hard to attach these words to every field. One possible approach is to develop the patterns automatically from an annotated corpora.

4.1. Pattern Extraction Model

In order to obtain the `select_expression`, an information extraction method is used. The definition of information extraction is: a system that identifies specific pieces of information in an unstructured or semi-structured textual document. In my natural language interfaces to databases (NLIDB) system, the specific information refers to the `select_expression` and the unstructured document refers to the sentence.

Because the original corpus of GEOQUERY is developed in Prolog, it has to be modified to a SQL (Structured Query Language) form first. For example, an entry in the original corpus:

```
parse([give,me,the,cities,in,virginia,'.''])
answer(A,(city(A),loc(A,B),const(B,stateid(virginia)))))
```

is then modified to the following form:
select city.name from city where city.state='virginia';

The tagging part of the transformed corpus is obtained using the Brill’s tagger (Brill, 1995), and the SQL part is constructed manually. In total, there are 880 questions in the training set and 250 questions in the testing set. All had been translated and tested to work under Mysql environment. However, since the system only focuses on the non sub-query statement, not every question in the training set is used. The number of non sub-query sentences in the training set are 543 questions (61.70%) and the number of non sub-query sentences in the testing set are 175 questions (70%).

After stopwords removal, every question in the corpus is grouped according to its select expression (sentences in the training set that have the same select expression translation are belong to a group), below is an example of grouped sentences with their select expression (lake.name):

```
lake.name:
give VB lakes NNS california NN
name NN lakes NNS
name NN major JJ lakes NNS michigan NN
show NN major JJ lakes NNS
what WP lakes NNS states NNS bordering VBG texas NNS
what WP major JJ lakes NNS united VBN states NNS
```

Every sentence in a group can serve as a pattern: in terms, given a new question, it can be compared. If it matches, the corresponding select expression can be chosen. However, at this stage, the patterns are too specific; unless the new sentence appears exactly the same as the training set, the select expression cannot be produced. Hence, the task is to make the pattern as general as possible, that it can have good coverage, at the same time maintaining its accuracy.
4.1.1. Graph Based 1

Regardless of the existing techniques, I use a novel graph based model to approach the problem. A vertex in this graph refers to a word, and an edge represents a word can be placed in front of another word. A start state ($S_0$) is added at the beginning of the graph, and an end state ($S_f$) is added at the ending of the graph. A match with a pattern means there is at least a path connecting the $S_0$ to the $S_f$.

The same words in a group of patterns can be merged into a single vertex; with the exception that when the same word appears more than once in a sentence, it cannot be merged, because it may change the overall meaning of that particular sentence. Using this step, the graph becomes more flexible, since a vertex, which is also a word, can cross among different patterns. Therefore, there are more possibilities to construct a path from $S_0$ to $S_f$.

The next step is dealing with the value tokens. According to the database structure, a more general term that is close to a value is a field. Therefore, every value token appearance can be replaced with its corresponding fields. But, a value token may have several fields depending on its entries in the database. Consider the value token below:

value token: "Missouri"

fields : border.state, city.state, state.name, river.name, river_flow.name

Since at this point, the correct semantic meaning is not known, then all the possible fields are included. However, there is a way to reduce the number of possible fields. The possible fields of a value token can be compared to the possible fields of another value token which appears in the same state in different patterns, every match is stored (table 4.1). In the case of no matches found, both value tokens cannot be combined; therefore, they have to be split into two different vertexes.

Every vertex in a graph is assigned to several features; they are: the real word, its stem, and its POS (part of speech). Exceptions are made to the start state ($S_0$), end state ($S_f$), and value tokens. Instead of just considering a match with the real word (the word as it appears in the training set), a word in a new question can also be compared to these features.
Therefore, we can adjust the graph to be more general. In the future, a new feature can be added with a slight modification.

A wildcard(*) is introduced. A wildcard in a feature means it can match everything. All features in a vertex will be changed to the wildcard first (*/*/*); every time a change is made, the graph is re-evaluated to the training set. If the overall score is lower then the change is canceled. The scoring formula is as below:

\[
\text{score} = 0.9 \times \text{match} - 0.05 \times \text{over} - 0.05 \times \text{under}
\]

"match" means the number of correct patterns it identifies, "over" means the number of the graph catches other patterns, and "under" means the number of the graph misses the correct patterns. After "all wildcard" step (*/*/*), the wildcard is arbitrarily applied to the word, stem, and tag; in a sense, trying to get more general at each step.

At the end of the above stage, a graph pattern based 1 is completed. To provide detailed explanation, a sample case for a select expression (city.population) is given. In this example, there are four steps necessary to build a completed graph based 1. They are:

(i) Building a plain graph from the patterns
(ii) Merging vertexes
(iii) Assigning value tokens according to their fields
(iv) Generalizing the graph
Consider eight patterns for city.population (the actual number for patterns in city.population are 50):

city.population:
how/WRB many/JJ people/NNS live/VB washington/NN dc/NN
how/WRB many/JJ people/NNS live/VB austin/NN
number/NN citizens/NNS boulder/NN
number/NN people/NNS boulder/NN
what/WP population/NN erie/NN pennsylvania/NN
what/WP population/NN springfield/NN missouri/NN
how/WRB many/JJ inhabitants/NNS montgomery/NN
how/WRB many/JJ people/NNS new_york/NN

transforming patterns into a graph structure:

![Plain Graph of City.population]

**Figure 4.1.** Plain Graph of City.population
merging same words into a single vertex:

Figure 4.2. Merged Graph(1) of City.population

changing value tokens into their field:

Figure 4.3. Merged Graph(2) of City.population

35
generalizing patterns with the training set:

4.1.2. Graph Based 2

In making graph model 1, a question arises; what if a new sentence is longer than a pattern, in which there are other words between two vertexes, such that a vertex cannot go to the next vertex? Graph model 1 may fail to identify this.

Another wildcard (**) is introduced. This means that a vertex can match everything multiple times until the next vertex is reached. This next vertex must not be another **. In the generalization step, instead of changing the vertex into "all wildcard" first, the ** is applied. The other steps are exactly the same as graph model 1. An evaluation and comparison result will be provided in section 4.3.

4.2. Vectorial Model

Another possible approach is using the vectorial model. Thinking of a select group and its patterns as a document and a new question as a short document. Then by reading the training set we can collect multiple documents. Therefore, we can measure the closeness of
their vector (the training groups) to the query vector (the new question). The closest vector is regarded as the result.

Standard \textit{tf-idf} (term frequency - inverse document frequency) is used as the weighting factor. The complete formula is as below:

\[
\begin{align*}
\text{tf}_{ij} &= \frac{f_{ij}}{\text{max}(f_{ij})} \\
\text{idf}_i &= \log_2(N/\text{df}_i) \\
W_{ij} &= \text{tf}_{ij} \cdot \text{idf}_i = \frac{f_{ij}}{\text{max}(f_{ij})} \cdot \log_2(N/\text{df}_i)
\end{align*}
\]

where:

- \(f_{ij}\) = frequency of term \(i\) in document \(j\)
- \(\text{df}_i\) = document frequency of term \(i\)
- \(N\) = total number of documents

After the weight is obtained, the similarity between two vectors is measured using the cosine similarity formula:

\[
\text{Sim}(d_j, d_k) = \frac{\vec{d}_j \cdot \vec{d}_k}{\|\vec{d}_j\| \cdot \|\vec{d}_k\|} = \frac{\sum_{i=1}^{n} W_{ij}W_{ik}}{\sqrt{\sum_{i=1}^{n} W_{ij}^2} \sqrt{\sum_{i=1}^{n} W_{ik}^2}}
\]

The vectorial model regards a document as a bag of words, without considering the relations between words, such as the position of a word in a sentence, and its tag (POS). However, it is still an interesting method to try, since in many cases, it proves to be successful which really means that the relations between words are not as necessary as one may think.

4.3. Pre-evaluation

Since the testing set also contains the correct SQL translation for every question, the three approaches can be evaluated. In graph based 1 and 2, a new question is evaluated against the graphs. The graph that gives the longest successful sequence (able to reach \(S_f\)) is chosen, and in case of ties, the last graph is the answer. For the vectorial model, the
pattern that gives the best cosine similarity score is the result. Finally, the method that
gives the best performance in terms of precision and recall will be applied to the system.

As mentioned before, the testing set has 175 non sub-query questions. They are given as
inputs for every method, and the result is as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Attempt</th>
<th>Correct</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Based 1</td>
<td>175</td>
<td>174</td>
<td>99.43%</td>
<td>99.43%</td>
</tr>
<tr>
<td>Graph Based 2</td>
<td>175</td>
<td>172</td>
<td>98.29%</td>
<td>98.29%</td>
</tr>
<tr>
<td>Vectorial</td>
<td>175</td>
<td>155</td>
<td>88.57%</td>
<td>88.57%</td>
</tr>
</tbody>
</table>

Table 4.2. Pre-evaluation Result

The vectorial method gives a very low score compared to the others. This is due to the
closeness of certain groups. Treating patterns as bags of words cannot differentiate certain
sentences that are close to the similar groups, such as sentences that address the following
fields: state.name and city.state. Overall, since the patterns belong to the same domain
(U.S. geography), there are a lot of words overlapping in different patterns, that make the
weight distributes equally; therefore, the vectors are very similar to each other.

Graph based 2 has three errors. They are:

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Answer</th>
<th>Tried</th>
</tr>
</thead>
<tbody>
<tr>
<td>what/WP is/VBZ the/DT population/NN of/IN new_york/NN city/NN</td>
<td>city.population</td>
<td>state.population</td>
</tr>
<tr>
<td>how/WRB many/JJ cities/NNS are/VBP there/EX in/IN the/DT us/PRP count(city.name) sum(state.area)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>how/WRB many/JJ people/NNS live/VB in/IN new_york/NN</td>
<td>city.population</td>
<td>state.population</td>
</tr>
</tbody>
</table>

Actually Graph based 2 is able to recognize the correct answer for every mistaken question
given above; however, the correct graph score is the same as the incorrect graph, and because
it precedes the incorrect graph, the incorrect one is chosen (in the case of a tied score, the
last graph is chosen). Changing the sequence by choosing the first match is not a good
solution, since the problem still remains and other incorrect cases will occur. Other rules
should be applied in case of a tied score.

Graph based 1 has one error. That is:
The same as Graph based 2, the correct answer is also identified. However it has the same score as the incorrect one. At first, I thought that graph based 2 would have better performances than graph based 1; however, it is not true. In the case of a longer sentence, the words between two vertexes get pruned by the stopword removal. Usually, these words are at most three words longer and they are meaningless. Even if they are really long, they may imply different semantic meanings such that they cannot be categorized into the pattern. Based on the pre-evaluation results graph based 1 is implemented.
CHAPTER 5
WHERE EXTRACTION

5.1. Identifying Hints

A question in a natural language interfaces to databases (NLIDB) system is meant to ask information in a database. Therefore, there are words in the question that are related to the terms in the database. These words are called hints: words in a sentence that can be matched with the tokens extracted from a database (table token, field token, value token). As a term in a database may overlap between different tokens, a hint in a sentence may also overlap between different tokens.

Hints are useful in two ways. First, they map the relations between a sentence and a database. Second, they can be used to provide the correct semantic meaning of a sentence. Because of these characteristics, all hints found in a sentence should be used to form a SQL (Structured Query Language) query.

For example consider the illustration in figure 5.1:

```
Sentence
How long is the river in Texas?

SELECT statement
SELECT length
FROM river,river_flow
WHERE
  river.name=river_flow.name
  AND state='texas';
```

**Figure 5.1.** Hints and their Relation with a Database and SQL Query

A sentence “How long is the river in Texas?”, has three hints: ”long”, ”river”, and ”Texas”. The hint ”long” can be matched with the field token ”length”, which is related to the field
name "length" in "river" table (river.length). The hint "river" can be matched with the table token "river". And the hint "Texas" can be matched with the value token: "Texas", which is related to "Texas" as the entry in the river_flow.state field. With this information, the following SQL query can be generated:

SELECT length FROM river,river_flow WHERE river.name=river_flow.name AND state="texas";

5.1.1. User Defined Dictionary

As described in the first chapter, it is impossible to find the semantic meaning of a nominal compound phrase without any prior-knowledge of the domain. Therefore, in my NLIDB system, these phrases have to be defined manually in a text file named "user.cps" which has the following structure:

[Noun Phrases]
[Semantic SQL representation - field operator field/value]
[Newline]

For example, a noun phrase "major city" which means: city.population>'150000' and a noun phrase "major river" which means: river.length>'750' are written:

major city

  city.population > 150000

major river

  river.length > 750

In total there are six entries in the user defined dictionary for the Geobase domain.

There are two steps to find hints in a sentence based on the user dictionary. The first is every word in a sentence is compared to the entries in the user dictionary, any matches will be recorded. The second is to build bigram models of a sentence; two words next to each other are merged into a single bigram. For example, a noun phrase "major city in Ohio"
consists of the following bigrams: "major city", "city in", "in Ohio". The three bigrams are then compared to the user dictionary, any matches will be recorded; the matched bigrams will be kept and given a new POS of NN, while the unmatched bigrams are restored to the previous form.

Another possible approach to identify hints in a sentence is: using the information contained in the parse tree. Every noun or noun phrase appeared can be compared to the entries in the user dictionary. However, since my NLIDB system does not use any syntactic parser, this approach is not implemented.

5.1.2. Matching to the Database

Information in a database is extracted in the form of tokens. The tokens are divided into three types: table token, field token, and value token. In order to have more coverage, the stemmed form of a token is also stored.

A sentence is divided into words, and each word is stemmed. The next step is to compare each stemmed word with the stemmed tokens. The comparison method is the same with the method used on user dictionary based. First, a single stemmed word is compared. Second, the bigram models are constructed and then compared to the tokens.

5.2. Building WHERE using Shortest Path Approach

In order to build a WHERE statement, the where_conditions need to be filled. The where_conditions consist of two elements: "[field=value]" and "[field=field]". The first element refers to a specific value in a database, and the second element denotes the relations that connect the field from the first element to the field in the select_expression.

The problem of filling the semantic meaning in a WHERE statement can be viewed as the problem of finding the shortest path of different vertexes in a graph. The graph here is the database structure; with each field as a vertex and a relation as an edge. Every edge has a weight attached to it. While more weight in the extracted relations carries more meaning, more weight in the shortest path means the opposite. Therefore, every weight has to be transformed according to the following formula:
\[
W_e = \infty - W_r
\]

Where:

- \( W_e \) = Weight of an edge
- \( W_r \) = Weight of a relation

5.2.1. Defining Values

The first element of where_conditions is "[field=value]" which refers to a specific value in a database. In order to specify this value, the corresponding entry in the database should be explicitly stated in the sentence. Although there are possibilities by stating other names (nicknames) of the entity; they require prior-knowledge of the domain. Thus, cannot be automatically extracted from a database. For example: the state “Texas” has a nickname “The lone star state”. However, if “The lone star state” is mentioned in a sentence, it cannot be identified because the database does not contain any entry of “The lone star state”.

Every value explicitly specified in a sentence can be found by finding a hint which has the type of value token. The word as it appears in the sentence can be given as the value. For example: a sentence ”How big is Texas?”, has the value token ”Texas”. Since the field is not known yet, by assuming the field as ”fielda”, the where_condition for ”Texas” can be written as: fielda='Texas'. And therefore, the next task is to find the correct field for each value, as it reflects the user request.

Every value token has several possible fields, but the field that is already used in the select_expression cannot be chosen. Although if it is chosen, it will still form a valid SQL query. But, this type of question is a self-answered question that should not appear. For example:

"What is the state of Texas?"

```
SELECT state.name FROM state WHERE state.name='Texas'
```

Hence, the list of possible fields can be expressed as follows:
ListP = ListA - ListB

Where: ListP is the list of possible fields, ListA is the list of possible fields for a value token, and ListB is the field of the select_expression.

The distance from every field in the ListP to the field in the select_expression can be measured using shortest path method, the Djikstra algorithm is implemented in finding the shortest path [4]. The field that gives the shortest distance is chosen, which means: the best field to express the user request. Using this approach the problem of ambiguities can be minimized.

For example consider a sentence "How long is the shortest river in Texas?" The corresponding select_expression is \( \text{min(river.length)} \), and thus the field for the select_expression is river.length. The possible fields for the value token "Texas" are: border.border_state, border.state, city.state, highlow.state, river_flow.state, road.state, and state.name. Because river_flow.state has the shortest distance compared to the other fields, it is chosen to build the correct semantic meaning. Then, the corresponding where_condition can be written as: river_flow.state='Texas'. Figure 5.2 depicts the example.

**Figure 5.2.** Choosing the Field based on the Shortest Distance
Not every value can be translated by naively applying the shortest path method. The specific field of certain value is explicitly stated in the sentence by the user, hence it has to be selected regardless of its distance to the field in the select_expression. For example, consider a sentence ”What are the rivers in the states border Ohio?” the select_expression field for this sentence is river.name and the closest possible field of a value token ”Ohio” is river_flow.state. But the correct translation as requested by the user is \texttt{border.state='Ohio'}.

An approach to revise the correct possible translation of a value token is developed. First, all hints in a sentence are identified. The closest hint (measured by the position to the value token) that is not another value token is chosen. If it does not appear as the field of the value token and it does not appear in any of the relations. There is a possibility that the hint is the field for the value token as specified by the user.

The list of possible fields of the closest hint is generated. This list is then intersected with the ListP (further reducing the list of possible fields for a value token), resulting a new list (ListR). Finally, the shortest path algorithm is re-applied to the ListR.

5.2.2. Constructing Relations

A relation which is the same as \texttt{[field=field]} can be derived after the first element \texttt{[field=value]} is built. The shortest path from the field in the \texttt{[field=value]} to the field in the select_expressions contains all the relations. For example, consider the sentence ”which rivers run through states bordering Ohio?” Until this point, the elements of the corresponding SELECT statement obtained are:

\begin{verbatim}
select_expressions:  river.name
where_conditions:    border.state='Ohio'
\end{verbatim}

The shortest path connecting border.state to river.name is as shown in the figure 5.3.

There are cases when the constructed relations are not correctly expressing the user request. They due to the leftover hints; hints in a sentence that are never used to form the
SELECT statement. In a sense, rather than goes through the shortest path, the user wants to reach the destination by traversing through several vertexes.

The problem of the leftover hint can be viewed as the problem of finding the shortest path with constraints. All hints are scanned, any leftover hint is put into an array. The field from the select_expressions serves as the start state and the first element in the leftover array serves as the goal state. A shortest path connecting the start state to the goal state is built. Because the new path may contain other fields in the leftover array, the array is cleaned, and the leftover hints are re-scanned considering the new path. The start state is now the last field in the new path (the previous goal state) and the goal state is now the first element in the re-scanned array. Another shortest path is built and connected to the previous path. These steps are repeated until there is no leftover hint found. Finally the last field in the path is connected to the field in the value token. The algorithm can be written as follows:
Startnode = SELECT field
Repeat
  Scan leftover hints, stored in an array
  Pick the first element
  Build a path from the Startnode to the 1st element
  Startnode = 1st element
Until no hint left

For example, consider the sentence "How many people live in the capital of Texas?" The SQL translation before applying shortest path with constraints is:

SELECT population FROM city WHERE state='Texas'

Although it implies the shortest distance from city.state to city.population, the above translation is not correct. There is a hint in the sentence ("capital") that cannot be found anywhere in the SELECT statement; however, the user emphasizes this term to be specified. Applying shortest path with constraints yields:

SELECT city.population FROM city,state
WHERE city.name=state.capital and state='Texas'
6.1. Evaluation

My natural language interfaces to databases (NLIDB) system was evaluated on the GEOQUERY domain. The GEOQUERY domain can be divided into three parts: the database (GEOBASE), the training set, and the testing set. Since the GEOQUERY was built in Prolog, it was transformed to the Mysql format. Using the transformed Mysql database, all the tokens and relations were extracted and saved into two binary files. In order to obtain the patterns for the select expressions, a pattern extraction approach (graph based 1) was run on the 543 non sub-query questions in the training set.

After the knowledge-extraction part finished, my NLIDB system was ready to accept new questions. 175 non sub-query questions in the testing set were given as the inputs for the system. And the performance was measured in terms of precision and recall:

\[
\text{Precision} = \frac{A}{B}, \quad \text{Recall} = \frac{A}{C}
\]

Where:

- \(A\) : Number of correct translations
- \(B\) : Number of questions answered
- \(C\) : Number of questions in the testing set

Every output produced by the system was compared manually to the output of the correct SQL (Structured Query Language) query. A correct translation means the system generates the same output as the correct query. Table 6.1 shows the performance of my NLIDB system in the GEOQUERY domain.
6.1.1. Comparisons

My NLIDB system was compared to three other systems: CHILL (Constructive Heuristic Induction for Language Learning), KRISP (Kernel-Based Robust Interpretation for Semantic Parsing), and WASP (Word Alignment-based Semantic Parsing). CHILL is an NLIDB system based on inductive logic programming [17], KRISP is an NLIDB system based on string-kernel-based classifiers [8], and WASP is an NLIDB system based on statistical machine translation [15]. The reason for choosing these systems are merely because they are available on the web (http://www.cs.utexas.edu/users/ml/geo-demo.html); therefore, they can be tested.

The same 175 non sub-query questions were given as inputs for each system. The outputs were compared to the outputs of the correct SQL query and the performance of each system was measured (table 6.2). Experimental results show that my NLIDB system compares favorably to the other systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Question</th>
<th>Attempt</th>
<th>Correct</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>My NLIDB System</td>
<td>175</td>
<td>175</td>
<td>168</td>
<td>96.00%</td>
<td>96.00%</td>
</tr>
<tr>
<td>CHILL</td>
<td>175</td>
<td>171</td>
<td>167</td>
<td>97.66%</td>
<td>95.43%</td>
</tr>
<tr>
<td>KRISP</td>
<td>175</td>
<td>164</td>
<td>155</td>
<td>94.51%</td>
<td>88.57%</td>
</tr>
<tr>
<td>WASP</td>
<td>175</td>
<td>169</td>
<td>151</td>
<td>89.35%</td>
<td>86.29%</td>
</tr>
</tbody>
</table>

Table 6.2. The Performance of Various Systems
<table>
<thead>
<tr>
<th>No.</th>
<th>Sentences and their incorrect answers</th>
</tr>
</thead>
</table>
| 1.  | **Question:** How many people live in New York?  
**Answer:** `SELECT city.population FROM city WHERE city.state='New York';`  
**My system:** `SELECT state.population FROM state WHERE state.name='new york';` |
| 2.  | **Question:** What are the high points of states surrounding Mississippi?  
**Answer:** `SELECT highlow.highest_point FROM highlow, border  
WHERE highlow.state=border.border_state AND border.state='Mississippi';`  
**My system:** `SELECT highlow.highest_point FROM highlow,city WHERE highlow.state=city.state  
AND city.name='high point' AND highlow.lowest_point='mississippi';` |
| 3.  | **Question:** Which rivers flow through Alaska?  
**Answer:** `SELECT river.name FROM river,river_flow  
WHERE river.name=river_flow.name AND river_flow.state='Alaska';`  
**My system:** `SELECT river.name FROM river,border,river_flow WHERE river.name=river_flow.name  
AND river_flow.state=border.border_state AND border.state='alaska';` |
| 4.  | **Question:** How many rivers does Alaska have?  
**Answer:** `SELECT count(river.name) FROM river,river_flow  
WHERE river.name=river_flow.name AND river_flow.state='Alaska';`  
**My system:** `SELECT count(river.name) FROM river,border,river_flow WHERE river.name=river_flow.name  
AND river_flow.state=border.border_state AND border.state='alaska';` |
| 5.  | **Question:** What are the rivers in Alaska?  
**Answer:** `SELECT river.name FROM river,river_flow  
WHERE river.name=river_flow.name AND river_flow.state='Alaska';`  
**My system:** `SELECT river.name FROM river,border,river_flow WHERE river.name=river_flow.name  
AND river_flow.state=border.border_state AND border.state='alaska';` |
| 6.  | **Question:** What is the highest point in states bordering Georgia?  
**Answer:** `SELECT highlow.highest_point FROM highlow, border  
WHERE highlow.state=border.border_state AND border.state='Georgia';`  
**My system:** `SELECT highlow.highest_point FROM highlow, border  
WHERE highlow.state=border.border_state AND border.state='Georgia';` |
| 7.  | **Question:** What is the total population of the states that border Texas?  
**Answer:** `SELECT sum(state.population) FROM state,border  
WHERE state.name=border.border_state AND border.state='Texas';`  
**My system:** `SELECT sum(state.population) FROM state,border  
WHERE state.abbreviation=border.state.abbreviation AND border.state='Texas';` |

Table 6.3. Questions that my NLIDB System Fails to Provide the Correct Answers
6.2. Discussion

Table 6.3 shows seven questions from the testing set that my NLIDB system fails to provide the correct answers. Based on what cause the errors, they can be divided into four groups:

(i) Incorrect select expressions (question number 1)

In the question number 1, graph based 1 gave the field "state.population" as the select expression; Although the correct answer should be "city.population". Since the field in the select expression serves as the destination in building the shortest path, the overall SELECT statement will be built toward the field. Therefore, an error in the select expression is propagated to the whole system.

(ii) The problems of synonyms (question number 2)

In the question number 2, there is a word in the sentence ("surrounding") that should be included as a hint; "surrounding" has the same meaning as a table token "border". Because the system does not have any synonym capability, it cannot be identified. Therefore, a value token "Mississippi" was attached to the field "highlow.lowest_point" which is incorrect.

(iii) The problems of no-entry in the database (question number 3 - 5)

Based on the testing entry, the correct translation of the value token "Alaska" is \texttt{river\_flow.state='Alaska'}. The value token "Alaska" was identified and the list of possible fields was generated. However, there was no entry of "Alaska" in \texttt{river\_flow.state}, hence it was excluded from the list of possible fields. Because \texttt{river\_flow.state} was excluded, it could not be chosen as the field for the value token "Alaska", and the system produced incorrect answer.

An approach to solve the problem is by considering every entry in the corresponding field (\texttt{river\_flow.state}). If it has a relation with other fields in different tables (e.g: \texttt{state.name}), then for certain threshold, every value in the other field (\texttt{state.name}) can be considered as an entry in the particular field (\texttt{river\_flow.state}).
Another approach is by extracting the knowledge from the training sentences. However, another information extraction method should be developed and the corpus should contain the related sentences.

(iv) Incorrect relations (question number 6 and 7)

In the question number 6 and 7, my NLIDB system used border.state.abbreviation as the relation for the SELECT statement. However, the correct relation should be border.border.state. The problem is due to the database structure. Consider a Prolog fact for table "border" in the GEOBASE:

\[
\text{border(\text{'alabama'}, \text{'al'}, [\text{'tennessee'}, \text{'georgia'}, \text{'florida'}, \text{'mississippi'}])}.
\]

The fact was transformed to a table ("border") that consists of three fields: state, state.abbreviation, and border.state,. The array information was kept in the border.state. Since the array has four elements, the table also has four entries respectively to the elements of the array. Thus, state.abbreviation occurred multiple times which denotes a strong relation. Because it is a strong relation, hence it is chosen to form the incorrect SELECT statement.

In order to fix the problem, the above fact should be transformed into two tables: a master and child table. The "state" table can act as the master table since it contains the same information (state.name and state.abbreviation) and a new child table should be created. However, changing the database structure causes the correct translation for the training and testing set should also be changed.

6.2.1. Comparison Discussion

When giving an input to a system, there are three cases that might occur: the result is correct, the system cannot produce the result, and the result is incorrect. There is not much can be learned from the first two cases. Therefore, we will only discuss the case when an incorrect result produced by a system.

Table 6.4 shows the questions that CHILL cannot answer correctly. Consider the function next.to in the first case; next.to(A,B) means state A border state B. Since both inputs for
<table>
<thead>
<tr>
<th>No.</th>
<th>CHILL: sentences and their incorrect answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Question: What is the total population of the states that border Texas?</td>
</tr>
</tbody>
</table>
|     | CHILL: answer(.99,(sum(.100,(population(.101,.100),state(.101)),.99),
|     | next_to(.101,.101),const(.101,stateid(texas)))) |
| 2.  | Question: How many people live in New York? |
|     | CHILL: answer(.136,(population(.137,.136),const(.137,stateid(new york)))) |
| 3.  | Question: What is the largest state capital in population? |
|     | CHILL: answer(.104,largest(.104,(state(.104),capital(.105)))) |
| 4.  | Question: What is the population of Portland, Maine? |
|     | CHILL: answer(.107,(population(.108,.107),const(.108,cityid(portland,.255)))) |

Table 6.4. Questions that CHILL Fails to Provide the Correct Answers

this function are of the same value (101 ≡ Texas), it means Texas border Texas which is incorrect. Thus, the query statement is also incorrect. The second case is due to the context ambiguity, hence this query can be considered as a valid answer. In the third case, CHILL misinterpreted ”largest” by choosing state as the input and gave the largest state. The correct answer should give the largest capital instead of the largest state. In the forth case, CHILL ignored the keyword ”Maine”. Since ”Portland” is the name of two cities, both populations are reported which is not the same with the output from the correct query (the population of Portland in Maine).

While KRISP has nine incorrect answers (table 6.5), they can be divided into four groups. The first group errors (No. 1 - 4) are related to the function loc_2, since there is no explanation about loc_2 in the paper, the exact reason cannot be provided. However, the output is incorrect. The second group (No. 5 and 6) is dealing with two values appear in a sentence. When two values appear in a sentence, KRISP chooses the last value and the first one is ignored. This may cause the system to produce incorrect answers. The third group (No. 7 and 8) is caused by the term ”US” which can be translated to countryid(’usa’), but to find the area of ”USA”, every area in each state should be calculated. Instead of giving the total area, KRISP generates ”None” as the result. The fourth group (No. 9) is due to the context ambiguity.
<table>
<thead>
<tr>
<th>No.</th>
<th>KRISP: sentences and their incorrect answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Question: What is the capital of Maine?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(capital(loc_2(stateid('maine'))))</td>
</tr>
<tr>
<td>2.</td>
<td>Question: What is the capital of New Hampshire?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(capital(loc_2(stateid('new hampshire'))))</td>
</tr>
<tr>
<td>3.</td>
<td>Question: What is the capital of North Dakota?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(capital(loc_2(stateid('north dakota'))))</td>
</tr>
<tr>
<td>4.</td>
<td>Question: What is the capital of Vermont?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(capital(loc_2(stateid('vermont'))))</td>
</tr>
<tr>
<td>5.</td>
<td>Question: What is the population of Seattle, Washington?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(population_1(cityid('washington',)))</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(population_1(cityid('washington',)))</td>
</tr>
<tr>
<td>7.</td>
<td>Question: How many square kilometers in the US?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(area_1(countryid('usa'))))</td>
</tr>
<tr>
<td>8.</td>
<td>Question: What is the total area of the USA?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(area_1(countryid('usa'))))</td>
</tr>
<tr>
<td>9.</td>
<td>Question: How many people live in New York?</td>
</tr>
<tr>
<td></td>
<td>KRISP: answer(population_1(stateid('new york')))</td>
</tr>
</tbody>
</table>

Table 6.5. Questions that KRISP Fails to Provide the Correct Answers

Like KRISP, the incorrect answers in WASP can also be divided into several groups. Since the table will be too large to fit all the 18 incorrect answers, only one example for each group is listed. The first three groups are the same problem like KRISP, hence they will not be discussed anymore. In the fourth question, when dealing with two values appear in a sentence, WASP chooses the last value. But, this approach may produce different query result; both values should be used to form the query. The fifth problem is related to the inconsistent entries in the GEOBASE. An entity "Ohio" as a river appears differently in the river table ("Ohio") and highlow table ("Ohio river"). Because of these inconsistencies, WASP failed to generate the correct query. In the sixth question, Instead of listing all the major river in Illinois, WASP gave the number of the major river which is false. In the seventh question, the function capital should be wrapped by another function (state), which means the state of a capital. However, WASP did not wrap the capital, hence the answer is
the capital itself (Columbus) which is different from the user request. In the last question, WASP failed to identify Mississippi as a river and gave the population of every state in the US. Therefore, it is incorrect.

<table>
<thead>
<tr>
<th>No.</th>
<th>WASP: sentences and their incorrect answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Question: What is the capital of Maine?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(capital(loc_2(stateid('maine'))))</td>
</tr>
<tr>
<td>2.</td>
<td>Question: How many square kilometers in the US?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(area_1(countryid('usa')))</td>
</tr>
<tr>
<td>3.</td>
<td>Question: How many people live in new york?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(population_1(stateid('new york')))</td>
</tr>
<tr>
<td>4.</td>
<td>Question: What is the population of Seattle , Washington?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(population_1(stateid('washington')))</td>
</tr>
<tr>
<td>5.</td>
<td>Question: What states does the ohio river go through?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(state(loc_1(placeid('ohio river')))))</td>
</tr>
<tr>
<td>6.</td>
<td>Question: What major rivers run through illinois?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(count(major(traverse_2(stateid('illinois')))))</td>
</tr>
<tr>
<td>7.</td>
<td>Question: What state is columbus the capital of?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(capital(cityid('columbus',.))))</td>
</tr>
<tr>
<td>8.</td>
<td>Question: What are the populations of states through which the mississippi river runs?</td>
</tr>
<tr>
<td></td>
<td>WASP: answer(population_1(state(state(all)))))</td>
</tr>
</tbody>
</table>

Table 6.6. Questions that WASP Fails to Provide the Correct Answers
CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. Conclusion

In this thesis, I presented a novel natural language interfaces to databases (NLIDB) system using a graph based model. The system works by obtaining knowledge automatically from a database and training corpus. With this knowledge, the system will analyze and translate a new question into its corresponding SQL (Structured Query Language) query.

To obtain the knowledge from a database, we can store all the database structures, such as: table names, field names, and value entries. The relations in a database can be extracted by considering identical value entries appear in different fields. A weighting scheme is applied to distinguish between strong and weak relations. The database is then mapped to a graph structure with the fields as the vertexes and the relations as the edges.

The knowledge in a corpus can be obtained using an information extraction technique. Sentences are grouped according to their select_expressions, the patterns for each group are developed using a novel graph pattern-based model. Several features are attached to each node in the pattern. The graphs are then trained using the training corpus by modifying the features to make the graphs as general as possible while maintaining their accuracy.

Given a new question, the problem of translating the question into a SQL query can be viewed as the problem of filling the elements in the SELECT statement. The select_expressions are obtained using the graph pattern-based model. The where_conditions are obtained by constructing shortest path with constraints from the value tokens in a question to the fields in the select_expressions. The table_references are extracted from the select_expressions and where_conditions.
The system was evaluated on the GEOQUERY; a database commonly used in the NLIDB domain for evaluating purposes. A comparison was made to other NLIDB systems. The results showed that my NLIDB system was able to achieve high performance (96% Precision and Recall) and compared favorably to the state-of-the-art in NLIDB.

7.2. Future Work

This section will discuss suggestions for future works. The suggestions are divided into two subsection: suggestions for system improvement and suggestions to improve the system capability by handling nested structures.

7.2.1. System Improvement

Based on the evaluation results, there are suggestions to increase the system performance:

(i) Using a lemmatizer instead of a stemmer

In order to obtain the base form of a word, a stemmer is used. However, it may not be the best approach. Since a stemmer only considers the general base form of a word, it may give incorrect outputs for certain words such as irregular verbs. A lemmatizer is more precise system, hence it is a better option.

(ii) Analyzing value tokens based on syntactic parser

Because my NLIDB system does not use any syntactic parser, a bigram model is implemented. But, this is not a good approach, since it will fail to identify a value token which consists more than two words. Using a syntactic parser, every noun or noun phrase in the parse tree can be compared to the value tokens. Since a noun phrase may consist of several words, it will be able to identify all the value token entries.

(iii) Adding feature to handle synonym

Currently my NLIDB system does not have any feature to handle the synonym of a word. Using an online based dictionary such as Wordnet will enable the system to recognize the synonym of a word.

(iv) Designing a method to deal with numbers
Because the GEOQUERY does not contain many sentences dealing with numbers, my NLIDB system only provides a little support to numbers. However, number is an important term in the real world domain. Therefore, a greater support should be provided. A number has more operators than a string such as: greater than, less than, greater equal, etc. One approach to deal with numbers is: All numbers appear in a database are represented by one value token ("Number"), the possible fields for "Number" are stored according to the database entries. If a number appears in a sentence, the exact value is stored, and the operator must be traced from that sentence. The corresponding field for the number can be searched using the shortest path approach. Finally, a where condition can be built using the following syntax: [field] [op] [number].

(v) Designing a method to handle tied-score patterns

The select expressions are obtained using the score produced by the graph pattern-based model. In the case of a tied score, the last pattern is chosen. However, it may not be the correct answer. Since the other patterns have the same score, they also have the same possibilities to be the answer. Therefore, a new approach to distinguish the best pattern in the case of a tied score should be built. One way is to attach a weight to every vertex in the graph. Since a vertex represents a word, the tf-idf weighting scheme is suggested.

(vi) Adding degree of certainty

Not every query generated by my NLIDB system is the correct query. But, there is no way to measure the query. Adding degree of certainty will help the user to identify the query. At every stage in building the SELECT statement can be scored. The total score is then calculated and displayed along with the query.

7.2.2. Handling Nested Structures

My NLIDB system can be considered as a pattern based system. While the pattern based system suffers from the problem of handling nested structures, the syntax based system
has difficulties to directly map a parse tree into a general database query language (SQL). Therefore, a hybrid system combining both approaches is suggested.

The parse tree of a sentence is generated by the syntax based system. Then, the task is to find a subset of the tree that can be given as an input for the pattern based system (verb phrase or noun phrase). This subset is then merged into a single node. All other subsets are searched and merged. Finally, a formal language function can be attached to the node in the reduced parse tree. The algorithm is as follows:

1. Generate a parse tree
2. Identify subsets of the parse tree which can be given as inputs to the pattern based system
3. Merge every subset into a single node
4. Generate the query for every merged node
5. Translate the reduced tree using a formal language

Figure 7.1 and 7.2 depict an example for the suggested hybrid system. The node S in the reduced parse tree can be translated into a formal language function (Max) which has the following structure:
Instead of using the general syntactic based parser, other parsers may be easier to use such as: KRISP [8] and another parser based on the probabilistic categorial grammar [18].
BIBLIOGRAPHY


