A System for Natural Language Searching of Facebook's XML-formatted User Database

Eric Y. Hwang and Michael C. Hughes
F. W. Olin College of Engineering
2 May 2007

1. Introduction

Project Overview

With the ability of computers to store vast quantities of information, accessing that information in an efficient manner is important. For low-level access to database information, the Server Query Language, or SQL, provides a powerful method of access. However, SQL takes time to learn and master, so most information access systems targeted towards people with no specialized knowledge use either keyword searches or form-based searches. Although not as powerful as specialized query languages like SQL, those interfaces are much more intuitive to use. An alternative method that would possess both the power of query languages and ease of use would be a natural language interface. Natural language interfaces allow users to pose questions to databases in a natural language such as English and have their queries answered according to the information in the database.

In this paper, we describe our simple natural language interface to the social networking site Facebook. We choose to use an external knowledge base as the basis of our system to allow for more attention to the problems of natural language processing and query evaluation without getting bogged down in data collection, entry, and maintenance. Additionally, using a sophisticated external knowledgebase such as Facebook allows us to gain easy access to a diverse, information-rich data source, which makes our queries all the more interesting and relevant. We begin by outlining the general structure of our program. We then describe in detail the natural language processing and query processing components.

General Program Structure

Our program is divided into two distinct parts: a natural language processing component and a query parsing / database access component. The two parts interface with each other through a simple shared query language that we designed (described in detail in Section 3). The two-part paradigm allows our natural language processing component to interface with databases other than Facebook (with the appropriate modifications to input files) as well as allowing other query input methods to query Facebook using our query language.

The natural language component parses natural language input into our query language, and then the query parsing component takes the query, applies it to the database, and returns its results. We also have a simple GUI that formats the input and output of our two main program components.

2. Natural Language Processing

The natural language component is responsible for translating natural language queries into our query language. Currently, the natural language component can handle verb phrases describing conditions that must apply to a user, strung together by conjunctions. For example, it will take the following sentence fragment:

> are from Texas or California, go to MIT, Harvard, or Duke, and like watching House and Casino Royale

and translate it into this query:

> [hasCity(Texas)~hasState(Texas)]~[hasCity(California)~hasState(California)]&
> [hasAffiliationName(MIT)~hasAffiliationName(Harvard)~hasAffiliationName(Duke)]&
> [[likesMovie(House)~likesTV(House)]&[likesMovie(Casino Royale)~likesTV(Casino Royale)]]

The parser first translates the natural language input into an internal abstract representation of the conditions specified by the input. It then uses its internal representation to create the appropriate query. The intermediate internal representation allows the natural language component to generate queries formatted in different ways, increasing its flexibility. Currently, the parser can translate the following patterns into its intermediate representation:

> One verb phrase: <verb phrase>
> Many verb phrases: <verb phrase> (and/or) <verb phrase>
> Many verb phrases: <verb phrase>, <verb phrase>, …, (and/or) <verb phrase>

where <verb phrase> is one of the following:

> Single-object verb phrase: <verb> <object>
> Multiple-object verb phrase: <verb> <object> (and/or) <object>
> Multiple-object verb phrase: <verb> <object>, <object>, …, (and/or) <object>

<verb>s are words or sequences of words that are defined in an input file. <object> is a sequence of one or more words that does not contain any of the verbs in the verb input file nor the words "and" or "or". The system currently implemented handles any case of the above input patterns correctly. If a user attempts to use verbs that aren't defined in the input file, the behavior of the parser is not guaranteed to adhere to any particular specification. This is because it treats words that are not in its verb input file as parts of objects.

Verb phrases are prevented from nesting within one another to avoid structural ambiguity. The following shows two different ways of grouping one input phrase that contains a nested verb phrase:

- (are from Utah, like music, read Ayn Rand, or watch The Matrix), participate in orchestra, and are male
- are from Utah, (like music, read Ayn Rand, or watch The Matrix), participate in orchestra, and are male

Since even a human would not be able to tell which grouping to use in this case, and since a normal user would not need to construct such a convoluted query, we made the decision to limit the patterns that our parser would work with. The pattern that we use is flexible enough to allow our parser to process complex queries without having to deal with language ambiguity.

The parser uses different parsing algorithms depending on the complexity of the input. It first tries to categorize the complexity of the input sentence:

> No conjunction: One single-object verb phrase
> One conjunction:

One verb: One multiple-object verb phrase
Multiple verbs: Many single-object verb phrases
Multiple conjunctions: Many multiple-object verb phrases

If the input contains no conjunctions or only one conjunction, the parser directly parses the input into its internal representation of conditions. If there are multiple conjunctions, the parser finds the individual verb phrases, parses them separately, and then puts the resulting internal conditions together.

The parser uses two input files. One maps field names like "music" to query language calls like "likesMusic". The other maps each verb phrase, such as "live in", to the fields that the verb applies to, like "city,state".

The field to query language input file maps each field name, which can be arbitrary, to a single query language call. This allows us to easily adapt to changes in the query language by only changing a single data file. The verb input file does not have to be changed at all. Unfortunately, some of the query language formatting, such as the use of parentheses for method calls, square brackets for grouping, and the various symbols used to indicate "and" or "or", are imbedded inside the source code. Although changing them in the source code is not too difficult, it would be better if the information query language could be contained entirely within data file. It would not be too difficult to implement; we just did not have the time to do so.

The verb input file maps verb phrases to one or more field names. Multiple field names are allowed for one verb phrase because a verb phrase such as "lives in" could mean "hasCity" or "hasState". The easiest way to deal with multiple meanings is to just check them all. If multiple field names are given, the parser checks to see if any of them are true ("or" conjunction). "Lives in Nevada" will therefore find people who live in the state Nevada as well as those who live in a city named Nevada. The only restriction on verb phrases is that they cannot contain the words "and", "or", or commas, and that is only because those would interfere with the parser. Verb phrases can verb phrases specified elsewhere in the file. For instance, the verb phrases "are" and "are interested in" can appear in the same file. The parser tries longer phrases first, so if it does try a short phrase such as "are", it is guaranteed that any other phrase that contains that one, such as "are interested in", have already been tried.

The current parser functions correctly for all input following the well-defined pattern described above. However, its behavior is not guaranteed for input deviating from the defined patterns. A weakness in the current parser is that an object phrase cannot contain any of the verb phrases defined in the input file nor the words "and" or "or". Unfortunately, the parser cannot currently detect correctly formatted input, so it may crash or deliver unexpected results when incorrect input is given.

3. Query Evaluation

After fully transforming the user's natural language input into a more machine-friendly query, the program's execution flow moves into the query processing and evaluation phase. Before delving into the specifics of this phase, it is useful to first understand the process of accessing and structuring the knowledge database which we attempt to answer queries about.

Data Access

Our dependence on an externally-defined knowledgebase has a number of disadvantages, not the least of which is accessing the data in the first place. The process of connecting to a remote server each time a new query is evaluated can be problematic, since the user may be accidentally logged out or reach a query limit. Additionally, repeated calls would require nuanced knowledge of how to process the resulting information each time it is retrieved, since answers to different queries are not guaranteed to be formatted the same way and must be handled accordingly. Because of this, we decided that the program should access all possibly relevant data upon startup and then store that data in temporary internal structures which can be quickly accessed whenever a new query is entered.

Data Structure

Crucial for the success of this approach is the way in which the data is structured internally. In the interest of making our data structuring process both scalable and portable, we choose to adopt a flexible solution based upon the concept of a node tree. This choice came rather easily due to the fact that Facebook's data comes formatted with XML, allowing us to easily implement a tree-like structure. Using the idea that all data fields within a knowledgebase can be linked via semantic relations between "subject" and "object" nodes, we model the structure expected from the initial XML returned by Facebook as a series of these relationships written explicitly within a data file. For example, we can consider a node called a *user* to be a subject which possesses a relation called *hasName* which links it to the object node called *name*. The object node called *name* then links to a literal string value which represents a particular user's name. Figure 1 shows the basic structure of a user profile node model in the context of Facebook.

An important result of this structuring is that the program source code itself contains no information about specific node names or values; instead, it loads these all from the model's data file and the knowledgebase itself. Thus, we gain the advantage of portability, since our approach can be adapted to any knowledgebase which provides data in a format easily parsed into subject-object relations. Any database which allows XML based access, such as Facebook, can (in theory) be easily attached to our query processing program after the appropriate model files have been created.

One important caveat with our implementation is that often the external data source refers to multiple data fields by the same name. For example, with Facebook, a *user* has an object node called *name*, and each *affiliation* the user is related to also has a *name (*see Figure 1). This could cause problems in attempting to evaluate queries, since the evaluation process may not know which field to check. However, we realize that while we have no control over how the external source names its data nodes, we can define the relation name. By choosing unique relation names, we can ensure that fields will not become confused in the query process.

FIGURE 1

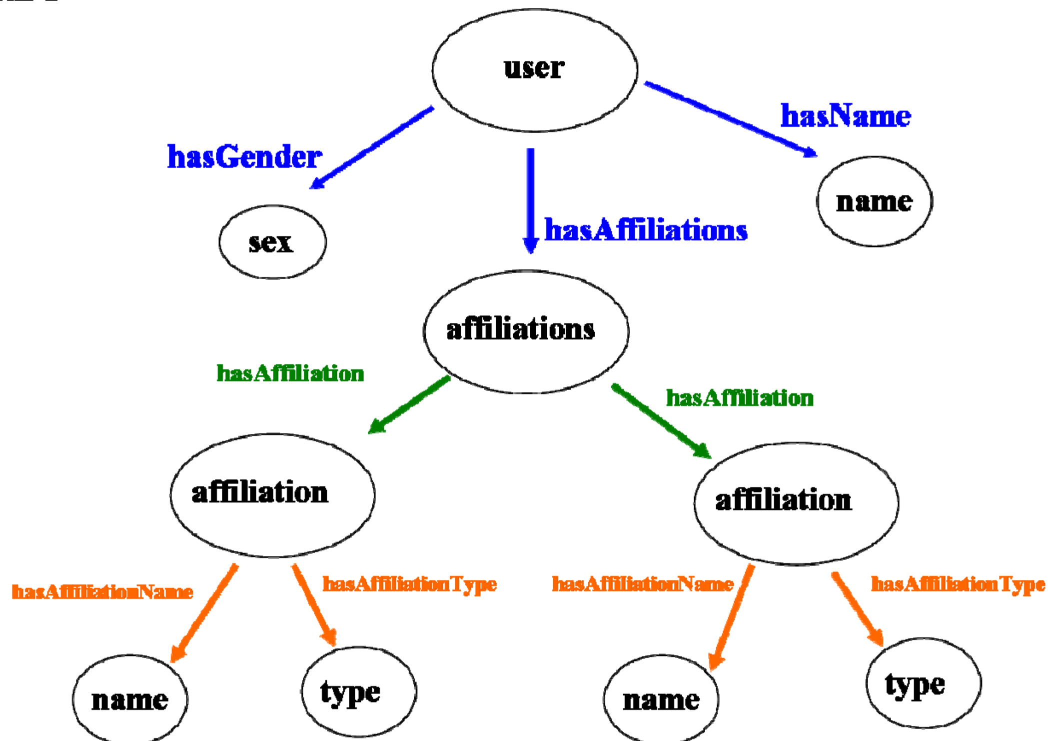

Figure 1 – Semantic Relation Structuring Diagram of a Facebook User

Query Language

With all relevant data acquired and appropriately structured, we turn our attention back to the problem of matching a given query with the corresponding data members within the internal structure. In order to efficiently process queries about all available data fields, we need to design a common query language around our semantic data structuring. At its fundamental level, a query in our language consists of a basic statement like

hasState(Massachusetts)

which indicates that the program should find all known users whose *hasState* relation is linked to the literal string value "*Massachusetts*".

More complicated statements in our query language can be seen in Table 1, including

(A) NOT condition, denoted by "!";
(B) AND conjunction, denoted with "&";
(C) OR conjunction, denoted by "~";
and (D) complex combinations of all the above

The query language will fail when given queries with a structure ambiguity such as found in (E). In cases where it is impossible to tell whether the user intended for an AND conjunction or an OR conjunction to be superior, no correct answer will exist and the program will most likely fail in its attempt to retrieve any answer or, even worse, return a false answer to the query. Perhaps enforcing a general heuristic like "when in conflict, AND supercedes OR" might resolve this issue, though we have not attempted any such fix.

In addition to the above behaviors, our query language supports nested queries, which are queries dependant on the result of other queries, as seen in (F). Processing this type of query from a natural language perspective proved to be challenging, and we have been unable to do so. Despite this difficulty, we have included the capability to process nested queries in the query language for possible future application.

TABLE 1:

A	!hasState(Ohio) * NOT YET IMPLEMENTED IN NL PROCESSING
B	hasGender(female)&hasAffiliation(Harvard)
C	hasInterest(poker)~hasInterest(billiards)
D	[hasInterest(baseball)~hasInterest(football)]&[!hasInterest(Yankees)&!hasInterest(Packers)]
E	hasGender(male)&hasInterest(AI)~likesMusic(rap) * STRUCTURAL AMBIGUITY
F	hasAffiliation(=hasName(Ryan Brady)) * NOT YET IMPLEMENTED IN NL PROCESSING

Table 1 – Sample Statements in our Query Language

Query Evaluation and Response

Using the given query language, it becomes rather straightforward to process the given queries against the data contained in our internal representations of the given knowledgebase. However, because we have left the formatting of each node's literal string value to an external data provider, the values of fields may not always take the form we would like and may in different cases require different methods of evaluation. To further complicate the issue, any given data field may be entirely blank.

There is not much we can do about a blank data field. However, we can use advance knowledge of what type of values we expect from a given node and attach to that node's model what we might call an *Evaluator* object, which can assist in making the right evaluation decisions about specific class of node. The type of evaluation ("contains", "equals", "case sensitive equals", etc.) can easily be set for an *Evaluator* of a given node. Additionally, we can give the *Evaluator* a list of equivalent string groups (defined in a separate data file), such that it can correctly equate "William" with "Bill" or the abbreviation "NY" with "New York" or even "Empire State." We can further facilitate desired evaluation behavior by applying a specified delimiter string to a given field's *Evaluator*. This allows node value strings to be split around specific regular expressions like whitespace or commas. With this functionality, a Facebook query for people with the name "Erik" will correctly find a profile with the name "Erik Kennedy" but will ignore the profiles of people named "Erika Boeing" and "Lauren Erikson." As another example, with the birthday field properly delimited, we can correctly search for people born on "May 2" without receiving results whose birthday field value equals "May 22" or "May 29."

In responding to a query, we provide answers at the highest level possible. That is, when we find Facebook users who all have a given interests, we retrieve the entire internal representation of matching users, not just their names. This provides maximal information to a user of our program and also allows for the query retrieving portion of our program to be more easily extended into a case in which we might wish to answer questions about previous queries. While we have no such capability now, the usefulness of being able to process natural language questions in series like

Which of my friends live in Wyoming?
Which of those like rodeos?

certainly encourages the preliminary practice of retaining all information about relevant results, as it may be a feature worth implementing in the future.

4. Conclusion

Results and Limitations

Within its defined capabilities, our program performs quite reliably. It can answer natural language queries about a range of data fields on Facebook. It can also use advance knowledge of how a given field is expected to be formatted to assist in more accurate, human-like evaluation. Though its portability has not been tested, we are confidant that if another XML-based knowledgebase were chosen and the appropriate alterations to the model definition files and vocabulary files were applied, the same program could answer the same class of queries within that database as well. Of course, the type of queries we can process from a natural language perspective is restricted to those which consist of compound statements of verb phrases, each of which can be a single-object or multiple-object phrase. Unfortunately, we are unable to handle object-phrases which contain the reserved key words defined as verbs or as conjunctions. Additionally, the domain is further limited by the need to avoid ambiguity within nested verb phrases or irresolvable conjunction superiority questions. Yet within these limitations, the program can still answer a large amount of interesting and useful questions about the

knowledgebase, and the more data fields we allow it to query, the more powerful the program becomes.

Comparison to Other Querying Methods

As a useful metric for gauging our success, we briefly compared our program's functionality to that of Facebook's own search features. Our program allows for more powerful queries than those allowed by the form-based advanced search provided on the Facebook website. Facebook's advanced search does not allow for any conjunction based searches within the same field (eg "Find people interested in swimming AND skiing" or "Find people who go to Olin or MIT"), does not allow the "OR" conjunction between different data fields (eg "Find people who go to Wellesley OR like guitar"), and also does not provide quite as sophisticated matching techniques (eg. a search for "Erik" finds people named "Erik" and "Erika"). Our program supports all of these features and also allows for convenient single-line input as compared to wide array of fields one must fill out when using Facebook's form-based search. We can thus consider our work at least on par if not superior to the current search capabilities for our chosen database.

Further Directions

Future improvements could certainly be made on many components of the program. Attempting to process more complicated natural language questions, such as negative queries like "Find people who do not like anchovies" or nested queries like "Find people who live in the state where Nick Bowers lives," would be an incredibly useful capability but would require substantial work within the natural language processing unit. Adding higher level reasoning capabilities to the query processing and evaluation component would also be very valuable. As an example, we could ask questions about a user's age when the knowledgebase contains only the birthday field. If we could somehow specify how to transform the birthday field into an age field within an external data file, we could generalize the process so that our program could answer queries about new data fields created from ones found in the knowledgebase. These features would significantly enhance the power and range of our program and make natural language-based querying of an external database all the more efficient and effective.