NaLIR: An Interactive Natural Language Interface for Querying Relational Databases*

Fei Li
Univ. of Michigan, Ann Arbor
lifei@umich.edu

H. V. Jagadish
Univ. of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

In this demo, we present NaLIR, a generic interactive natural language interface for querying relational databases. NaLIR can accept a logically complex English language sentence as query input. This query is first translated into a SQL query, which may include aggregation, nesting, and various types of joins, among other things, and then evaluated against an RDBMS. In this demonstration, we show that NaLIR, while far from being able to pass the Turing test, is perfectly usable in practice, and able to handle even quite complex queries in a variety of application domains. In addition, we also demonstrate how carefully designed interactive communication can avoid misinterpretation with minimum user burden.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Natural Language*; H.1.2 [**Information Systems**]: User/Machine Systems—*Human factors*

Keywords

Relational Database; Usability; Natural Language Interface;

1. INTRODUCTION

Traditionally, research work in querying data from relational databases often follows one of two paths: the structured query approach and the keyword-based approach. Both approaches have their advantages and disadvantages. The structured query approach, while expressive and powerful, is not easy for naive users. The keyword-based approach is very friendly to use, but cannot express complex query intent accurately. In contrast, natural language has both advantages to a large extent: even naive users are able to express complex query intent in natural language. Thus supporting natural language queries is often regarded as the ultimate goal for a database query interface.

However, progress has been slow, even as general Natural Language Processing systems have improved over the years. We believe this is primarily due to the difficulty of translating user-specified query structure to the actual schema structure in the database. By addressing this challenge, we believe we have removed the greatest barrier in natural language querying of databases.

In this demo, we describe NaLIR, a generic interactive natural language interface for querying relational databases. In NaLIR, an arbitrary English language sentence, which can be quite complex in logic, is taken as query input. This query is first translated into a SQL query, which may contain aggregation, nesting, and various types of joins, among other things. Then, an RDBMS is used to evaluate the translated SQL query and return the results to the user. For example, the user can write "return the author in database area, whose papers have the most total citations", without worrying about the structure of the elements in the database.

We believe that an ideal natural language interface should work like a database programmer (DBA): when the user tells the DBA what she wants to query in natural language, the DBA will first try to fully understand the natural language query from both a linguistic and a database point of view. Then, the DBA conveys his understanding, first for some ambiguous words/phrases and then for the structure of the sentence, back to the user to avoid misunderstanding. After the user agrees on or corrects the DBA's understanding, the DBA will compose the SQL query statement, evaluate it and finally return the results back to the user. Our system is designed in this way. In the first step, we use an off-the-shelf natural language parser to obtain the linguistic understanding (represented by a parse tree) of the input sentence. In the second step, we transform the linguistic understanding to a database's understanding by mapping proximity of the patterns in the parse tree to proximity of corresponding database concept. In order to make sure that the sentence is correctly understood, we explain the database's understanding (the linguistic understanding is included in the database's understanding) back to the user in natural language. Finally, the understanding is translated to a SQL query statement and evaluated by an RDBMS.

In our system, we focus on the second step: given a parse tree, how to correctly understand it from the database point of view. This step is challenging for many reasons. First, some words/phrases may fail in mapping to database elements due to the vocabulary restriction of the system. Also,

*Supported in part by NSF grants IIS 1250880 and IIS 1017296

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2594519.

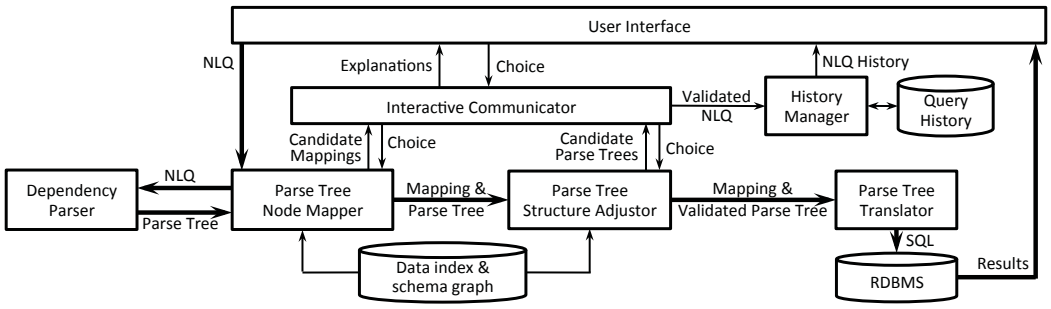

Figure 1: System Architecture.

some ambiguous words/phrases may map to multiple database elements, of which only some of them are correct. In the no match case, the only thing we can do is to point out which word/phrase is not recognized and ask the user to rephrase. In the multiple match case, we return all the possible choices to the user after ranking. Second, the parse tree generated from an off-the-shelf parser may not be correct. The natural language sentences describing complex database queries often have complex structures with modifier attachments, aggregations, comparisons, quantifiers, and conjunctions, among many others. As a result, the performance of an off-the-shelf parser is often unsatisfactory for sentences describing complex database queries. Third, the structure of the sentence may be out of the semantic coverage of our system. Fourth, due to the ambiguity of natural language, the fact that the sentence structure is in the semantic coverage does not necessarily mean that it is correctly understood.

For the second and third difficulties, we attempt to rephrase the user's input automatically by adjusting the structure of the parse tree to make it fall in our semantic coverage. Often, we may have multiple candidate adjustments. When that is the case, we show multiple options for the user to choose from. To make our adjustments understandable to the user, we translate each adjusted parse tree back in natural language. The fourth difficulty is dealt with in the same way. Even if the user's input is in our semantic coverage, if ambiguities are detected, we generate possible interpretations for users to choose from.

The paper is organized as follows: in Section 2, the architecture of NaLIR is introduced. We discuss related works in Section 3 and set up the demonstration in Section 4.

2. SYSTEM ARCHITECTURE

Figure 1 depicts the architecture of NaLIR. The entire system mainly consists of two parts: the query translation part and interactive communicator. The query translation part, which includes *parse tree node mapper*, *structure adjustor* and *translator*, is responsible for making full use of the existing information provided by the user and generate the correct SQL query statement. The *interactive communicator* is responsible for obtaining more information from the user and making sure that the system correctly understands her query intent.

2.1 Query Translation

The first obstacle in translating a natural language query into a SQL query is to understand the natural language query. In NaLIR, we use an off-the-shelf natural language parser to generate a parse tree from the natural language query[1]. The parse tree of query "return all the authors who have more papers than H. V. Jagadish in VLDB after 2005" is shown in Figure 2 (a).

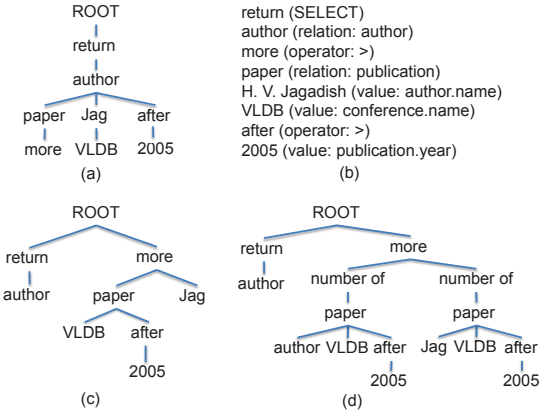

Figure 2: (a) Simplified parse tree from off-the-shelf parser. (b) Mapping results for the nodes in the parse tree. (c) Parse tree after reformulation. (d) Parse tree after inserting implicit nodes.

Parse Tree Node Mapper. The parse tree node mapper identifies the nodes in the parse tree that can be mapped to SQL components and tokenizes them into different tokens. Due to the limited vocabulary of the system, some nodes cannot be recognized by the system. Also, some nodes may have multiple matches (e.g. the node VLDB matches VLDB under conference.name, PVLDB and VLDB Journal under journal.name). These warnings are reported to the interactive communicator and will be explained to the user. The mapping results of the nodes in the sample query is shown in Figure 2 (b).

Parse Tree Structure Adjustor. After the node mapping (possibly with interactive communication with the user), we assume that each node is understood by our system. The next step is to correctly understand the tree structure from the database's perspective. However, this is not easy since the parse tree might be incorrect, out of the semantic coverage of our system or ambiguous from the database's perspective. In those cases, we adjust the structure of the parse tree and generate possible interpretations for it. In partic-

[1]In the current implementation, we use MySQL as the RDBMS, and Stanford NLP Parser [5] as the dependency natural language parser.

ular, we adjust the parse tree in two steps. In the first step, we reformulate the nodes in the parse tree to make it understandable by our system. A parse tree after reformulation is considered as an approximate interpretation for the query. If there are multiple approximate interpretations for the query, we will rank them and return top ones back for the user to choose from. Figure 2 (c) shows a parse tree reformulated from the parse tree in Figure 2 (a). In the second step, implicit nodes are inserted to the parse tree under the supervision of the user. After inserting implicit nodes, we get the exact interpretation for the query, which will be translated to a SQL statement with little ambiguity if at all. The parse tree in Figure 2 (c) is transformed into the parse tree in Figure 2 (d) after inserting implicit nodes.

Parse Tree Translator. Given the correct parse tree validated by the user, the translator utilizes its structure to generate appropriate structure in the SQL expression. In the case when all the join paths are explicit in the parse tree and the target SQL query does not have aggregation or subquery, the translation is quite straightforward. For other cases, we use three concepts, *related group*, *core node*, and *block* to complement the implicit join paths, clarify the scope of aggregation functions, and address nesting queries, respectively. Here, we take block as an example. In SQL, a block corresponds to a subquery in nesting, while in the parse tree, a block is a subtree rooted at an aggregation function node or a quantifier node. The parse tree shown in Figure 2 (d) has three blocks, rooting at ROOT and the two "number of" nodes, respectively. The left block will be translated to a subquery which returns the number of papers by each author in VLDB after 2005, while the right block will be translated to a subquery which returns the number of papers by H. V. Jagadish in VLDB after 2005. Finally, the upper block will be translated to the main query, shown as "SELECT author.name FROM author, left, right WHERE author.aid = left.aid AND left.count > right.count" for short. Based on these three concepts, our system, as it stands, supports comparisons, quantifications, aggregations, sorting and various kinds of joins.

2.2 Interactive Communicator

We design the interactive communicator for two goals. First, it explains to the user how her query is processed. That helps the user in realizing the possible misunderstandings between the user and the system. Second, it shows multiple understandings for the user to choose from, which facilitates the user in eliminating misunderstandings.

As pointed out in [8], people are unwilling to trade reliable and predictable user interfaces for intelligent but unreliable ones. So instead of leaving the users in the dark, our system explains how we deal with the query in every possible step from how we interpret an ambiguous word to the interpretation of the whole sentence. Especially, we translate the understanding of the whole query back to the user in English. For example, the parse tree shown in Figure 2 (d) will be translated back as "return all the authors, where the number of papers of the author in VLDB after 2005 is more than the number of papers of H. V. Jagadish in VLDB after 2005". In such a way, the user can easily examine whether our system correctly understands her query intent.

When misunderstandings exist, the system should help the user to eliminate these misunderstandings. Many previous systems put this burden to the user and ask the user to rephrase her input. From our real world experience, rephrasing is often an annoying task since the original input is often the most comfortable way for a user to express specific query intent. The situation is exacerbated by the unsatisfactory accuracy of the off-the-shelf parser for logically complex sentences. We believe that when a user fails in her first input, it is much easier for her to recognize her query intent from several interpretations than rephrase her intent. In our system, we generate multiple possible interpretations for a natural language query and translate all of them in natural language for the user to choose from. To facilitate the user in finding her query intent from possibly many interpretations, we show the interpretations hierarchically, in which each cluster is an approximate interpretation.

In addition to the whole sentence interpretation, our system also explain its understanding for each ambiguous node in the parse tree, making it possible for the user to find where the system misunderstands her. For example, if a node in the parse tree matches multiple elements in the database, the system will rank them and show the top ones for the user to choose from. The only case our system asks a user to rephrase is when a word/phrase is beyond the vocabulary coverage of our system. For example, a user queries the age of an author while there is no age information in the database. In this case, we have no option but to ask the user to rephrase. Indeed, this kind of rephrase is often at word level, which could be a much easier task for the user than rephrase at sentence level.

3. RELATED WORK

Keyword search interfaces are widely used by non-experts to specify ad-hoc queries over databases [12]. Recently, there is a stream of such research on keyword search [10, 9, 11, 3, 2], in which, given a set of keywords, instead of finding the data relevant to these keywords, they try to interpret the query intent behind the keywords in the view of a formal query language. In particular, some of them extend keywords by supporting aggregation functions [10], Boolean operators [9], query language fragments [3], and so forth. These works can be considered as a first step toward addressing the challenge of natural language querying. Our work builds upon this stream of research. However, our system supports a richer query mechanism that allows us to convey much more complex semantic meaning than flat structured (or a little more than flat structured) keywords.

Natural language interfaces to databases (NLIDB) have been studied for several decades [1]. Early NLIDBs depend on hand crafted semantic grammars tailored to each individual database, which are hard to transport to other databases. A few recent works present generic NLIDBs. PRECISE [8] defines a subset of NL queries as semantically tractable NL and precisely translate these queries into corresponding SQL queries. While PRECISE sacrifices recall for precision, our system focuses on both recall (using parse tree structure adjustor) and precision (using interactive communicator). NaLIX [7] achieves recall and precision by asking the user to rephrase their query until the query falls in its semantic coverage. Our system takes a step further, rephrasing the query automatically for the user. Also, the fact that a query falls in the semantic coverage does not necessarily mean that it is interpreted correctly. Our system provides different interpretations for a query (if ambiguity exists), so the user can choose.

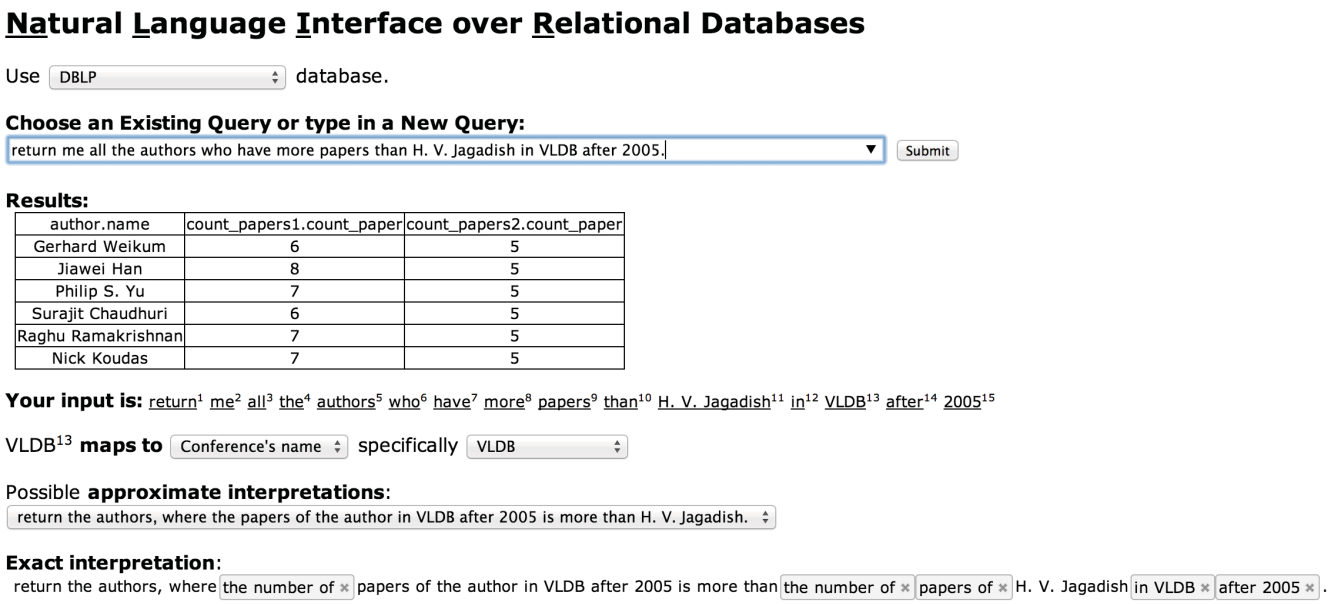

Figure 3: Query Interface of NaLIR

An important task in our interactive communicator is to convey each interpretation of the query intent back to the user. Previous systems explain SQL queries to naive users using natural language [6] and visualization tools [4, 2]. In our system, an interpretation is represented by a tokenized parse tree, which is an intermediate between SQL and a natural language query. So we translate each interpretation into natural language by reformulating the sentence given by the user according to each tokenized parse tree.

4. DEMONSTRATION

In our demonstration, we present the Javascript based interface of NaLIR, which communicates with the main Java based server. We intend to show the use of NaLIR against a number of real application scenarios including Microsoft Academic Search (http://academic.research.microsoft.com/), Yahoo! Movies (movies.yahoo.com), and DBLP collection (dblp.uni-trier.de).

The demonstration will consist of two phases. In the first phase, the user will run the queries in the query log, which contains a set of successfully processed NL queries. The chosen NL query can also serve as a query template for the user to make small modifications. In the second phase, the user will be free to run their own queries. We will demonstrate that quite complex query intents, which are typically expressed by complex SQL statements containing aggregations, comparisons, various types of joins and nestings, can be handled by our system.

Figure 3 shows a screenshot of NaLIR. When a new query is submitted from the client, the server processes it and returns the results. If NaLIR is uncertain in understanding some words/phrases, it adopts the best mapping as default and lists the others for the user to choose form. Also, when NaliR is uncertain in understanding the query intent behind the whole sentence, it lists multiple interpretations. To facilitate the user in recognizing her query intent, NaLIR shows interpretations hierarchically, in which each cluster/interpretation is an approximate/accurate natural language description. Each time when a user makes a choice, NaLIR immediately updates its interpretations, evaluates the best interpretation and updates the results.

5. REFERENCES

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[2] S. Bergamaschi, F. Guerra, M. Interlandi, R. T. Lado, and Y. Velegrakis. Quest: A keyword search system for relational data based on semantic and machine learning techniques. *PVLDB*, 6(12):1222–1225, 2013.

[3] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *PVLDB*, 5(10):932–943, 2012.

[4] J. Danaparamita and W. Gatterbauer. Queryviz: helping users understand sql queries and their patterns. In *EDBT*, pages 558–561, 2011.

[5] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.

[6] A. Kokkalis, P. Vagenas, A. Zervakis, A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Logos: a system for translating queries into narratives. In *SIGMOD Conference*, pages 673–676, 2012.

[7] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. In *SIGMOD Conference*, pages 900–902, 2005.

[8] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[9] A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.

[10] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD Conference*, pages 889–902, 2008.

[11] D. Xin, Y. He, and V. Ganti. Keyword++: A framework to improve keyword search over entity databases. *PVLDB*, 3(1):711–722, 2010.

[12] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.