Understanding Natural Language Queries over Relational Databases

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

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

ABSTRACT

Natural language has been the holy grail of query interface designers, but has generally been considered too hard to work with, except in limited specific circumstances. In this paper, we describe the architecture of an interactive natural language query interface for relational databases. Through a carefully limited interaction with the user, we are able to correctly interpret complex natural language queries, in a generic manner across a range of domains. By these means, a logically complex English language sentence is correctly translated into a SQL query, which may include aggregation, nesting, and various types of joins, among other things, and can be evaluated against an RDBMS. We have constructed a system, NaLIR (Natural Language Interface for Relational databases), embodying these ideas. Our experimental assessment, through user studies, demonstrates that NaLIR is good enough to be usable in practice: even naive users are able to specify quite complex ad-hoc queries.

1. INTRODUCTION

Querying data in relational databases is often challenging. SQL is the standard query language for relational databases. While expressive and powerful, SQL is too difficult for users without technical training. As the database user base is shifting towards non-experts, designing user-friendly query interfaces will be an important goal in the database community.

In the real world, people ask questions in natural language, such as English. Not surprisingly, a natural language interface is regarded by many as the ultimate goal for a database query interface, and many natural language interfaces to databases (NLIDBs) have been built towards this goal [1, 7, 9, 11, 13]. NLIDBs have many advantages over other widely accepted query interfaces (keyword-based search, form-based interface, and visual query builder). For example, a typical NLIDB would enable naive users to specify *complex*, *ad-hoc* query intent *without training*. In contrast, flat-structured keywords are often insufficient to convey complex query intent, form-based interfaces can be used only when queries are predictable and limited to the encoded logic, and visual query builders still requires extensive schema knowledge of the user.

Despite these advantages, NLIDBs have not been adopted widely. The fundamental problem is that understanding natural language is hard. Even in human-to-human interaction, there are miscommunications. Therefore, we cannot reasonably expect an NLIDB to be perfect. Therefore, users may be provided with a wrong answer due to the system incorrectly handling the query, and in many cases, it is impossible for users to verify the answer by themselves.

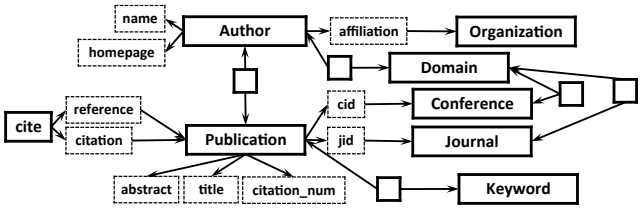
When humans communicate with one another in natural language, the query-response cycle is not as rigid as in a traditional database system. If a human is asked a query that she does not understand, she will seek clarification. She may do so by asking specific questions back, so that the question-asker understands the point of potential confusion. She may also do so by stating explicitly how she interpreted the query. Drawing inspiration from this natural human behavior, we design the query mechanism to facilitate collaboration between the system and the user in processing natural language queries. First, the system explains how it interprets a query, from each ambiguous word/phrase to the meaning of the whole sentence. These explanations enable the user to verify the answer and to be aware where the system misinterprets her query. Second, for each ambiguous part, we provide multiple likely interpretations for the user to choose from. Since it is often easier for users to recognize an expression rather than to compose it, we believe this query mechanism can achieve satisfactory reliability without burdening the user too much.

A question that then arises is how should a system represent and communicate its query interpretation to the user. SQL is too difficult for most non-technical humans. We need a representation that is both "human understandable" and "RDBMS understandable." In this paper, we present a data structure, called *Query Tree*, to meet this goal. As an intermediate between a linguistic parse tree and a SQL statement, a query tree is easier to explain to the user than a SQL statement. Also, given a query tree verified by the user, the system will almost always be able to translate it into a correct SQL statement.

Putting the above ideas together, we propose an NLIDB comprising three main components: a first component that transforms a natural language query to a query tree, a second component that verifies the transformation interactively with the user, and a third component that translates the query tree into a SQL statement. We have constructed such an NLIDB, and we call it a NaLIR (Natural Language Interface to Relational databases).

The remaining parts of the paper are organized as follows. We discuss the query mechanism in Section 2. The system architecture of our system is described in Section 3. Given a query, we show how to interpret each its words/phrases in

The original version of this article was published in PVLDB 2014.



Query 1: Return the average number of publications by Bob in each year.
Query 2: Return authors who have more papers than Bob in VLDB after 2000.
Query 3: Return the conference in each area whose papers have the most total citations.

Figure 1: A Simplified Schema for Microsoft Academic Search and Sample Queries.

Section 4 and infer the semantic meaning of the whole query (represented by a query tree) in Section 5. We discuss how our system translates a query tree to a SQL statement in Section 6. In Section 7, our system is evaluated experimentally. We discuss related work in Section 8 and draw conclusions in Section 9.

2. QUERY MECHANISM

Search engines are popular and effective in inferring the user intent from limited information. As we think about the architecture of an NLIDB, it is worthwhile to draw inspiration from search engines. First, given a query, a search engine returns a list of results, rather than a single result. This is central to providing acceptable recall. Second, users are able to verify whether a result is correct (useful) by reading the abstract/content. Third, these results are well ranked, to minimize user burden to verify potential answers. These strategies work very well in search engines. However, due to some fundamental differences between search engines and NLIDBs, as we will discuss next, this query mechanism cannot be directly applied to NLIDBs.

First, users are often able to verify the results from a search engine by just reading the results. However, the results returned by an NLIDB cannot usually explain themselves. For example, suppose a user submits Query 1 in Figure 1 to an NLIDB and gets an answer "5." How could she verify whether the system understands her query and returns to her the correct answer?

Second, unlike search engines, users tend to express *sophisticated query logics* to an NLIDB and expect *perfect results*. That requires the NLIDB to fix all the ambiguities correctly. However, when a natural language query has a complex structure, it may contain several ambiguities that cannot be fixed by the system with confidence. As a result, the candidate interpretations of the query are often permutations of the solutions for each single ambiguity. Such interpretations are hard to rank and their number often grows exponentially with the number of such ambiguities. Users will be frustrated in verifying them.

Given the above two observations, instead of explaining the query results, we explain the query interpretation process, especially how each ambiguity is fixed, to the user. In our system, we fix each "easy" ambiguity quietly. For each "hard" ambiguity, we provide multiple interpretations for the user to choose from. In such a way, even for a rather complex natural language query, verifications for 3-4 ambiguities is enough, in which each verification is just making choices from several options.

The ambiguities in processing a natural language query are not often independent of each other. The resolution of some ambiguities depends on the resolution of some other ambiguities. For example, the interpretation of the whole sentence depends on how each of its words/phrases is interpreted. So the disambiguation process and the verification process should be organized in a few steps. In our system, we organize them in three steps, as we will discuss in detail in the next section. In each step, for a "hard" ambiguity, we generate multiple interpretations for it and, at the same time, use the best interpretation as the default choice to process later steps. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps and updates the query results.

3. SYSTEM OVERVIEW

Figure 2 depicts the architecture of NaLIR. The entire system we have implemented consists of three main parts: the query interpretation part, interactive communicator and query tree translator. The query interpretation part, which includes *parse tree node mapper* (Section 4) and *structure adjustor* (Section 5), is responsible for interpreting the natural language query and representing the interpretation as a query tree. The *interactive communicator* is responsible for communicating with the user to ensure that the interpretation process is correct. The query tree, possibly verified by the user, will be translated into a SQL statement in the *query tree translator* (Section 6) and then evaluated against an RDBMS.

Dependency Parser. The first obstacle in translating a natural language query into a SQL query is to understand the natural language query linguistically. In our system, we use the Stanford Parser [2] to generate a linguistic parse tree from the natural language query. The linguistic parse trees in our system are dependency parse trees, in which each node is a word/phrase specified by the user while each edge is a linguistic dependency relationship between two words/phrases. The simplified linguistic parse tree of Query 2 in Figure 1 is shown in Figure 3 (a).

Parse Tree Node Mapper. The parse tree node mapper identifies the nodes in the linguistic parse tree that can be mapped to SQL components and tokenizes them into different tokens. In the mapping process, some nodes may fail in mapping to any SQL component. In this case, our system generates a warning to the user, telling her that these nodes do not directly contribute in interpreting her query. Also, some nodes may have multiple mappings, which causes ambiguities in interpreting these nodes. For each such node, the parse tree node mapper outputs the best mapping to the parse tree structure adjustor by default and reports all candidate mappings to the interactive communicator.

Parse Tree Structure Adjustor. After the node mapping (possibly with interactive communications 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 linguistic 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 linguistic parse tree and generate candidate interpretations (query trees) for it. In particular, we adjust the structure of the parse tree in two steps. In the first step, we reformulate the nodes in the parse tree to make it fall in the syntactic

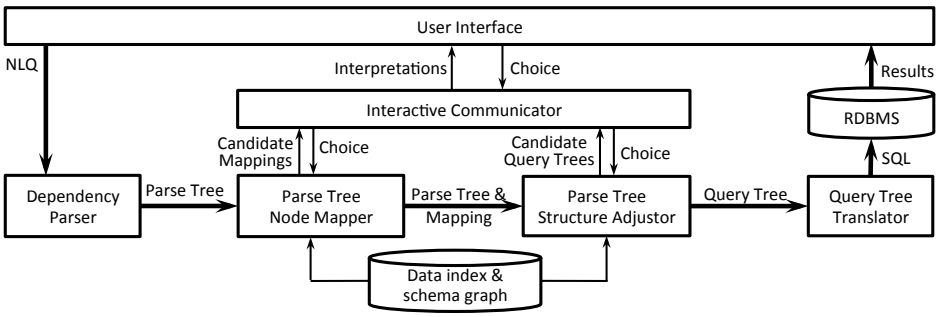


Figure 2: System Architecture.

coverage of our system (valid parse tree). If there are multiple candidate valid parse trees for the query, we choose the best one as default input for the second step and report top k of them to the interactive communicator. In the second step, the chosen (or default) valid parse tree is analyzed semantically and implicit nodes are inserted to make it more semantically reasonable. This process is also under the supervision of the user. After inserting implicit nodes, we obtain the exact interpretation, represented as a query tree, for the query.

Interactive Communicator. In case the system possibly misunderstands the user, the interactive communicator explains how her query is processed. In our system, interactive communications are organized in three steps, which verify the intermediate results in the parse tree node mapping, parse tree structure reformulation, and implicit node insertion, respectively. For each ambiguous part, we generate a multiple choice selection panel, in which each choice corresponds to a different interpretation. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps.

Example 1. *Consider the linguistic parse tree T in Figure 3(a). In the first step, the parse tree node mapper generates the best mapping for each node (represented as M and shown in Figure 3 (b)) and reports to the user that the node "VLDB" maps to "VLDB conference" and "VLDB Journal" in the database and that our system has chosen "VLDB conference" as the default mapping. According to M, in the second step, the parse tree adjustor reformulates the structure of T and generates the top k valid parse trees $\{T_i^M\}$, in which T_1^M (Figure 3 (c)) is the best. The interactive communicator explains each of the k valid parse trees in natural language for the user to choose from. For example, T_1^M is explained as "return the authors, where the papers of the author in VLDB after 2000 is more than Bob." In the third step, T_1^M is fully instantiated in the parse tree structure adjustor by inserting implicit nodes (shown in Figure 3 (d)). The result query tree T_{11}^M is explained to the user as "return the authors, where the number of papers of the author in VLDB after 2000 is more than the number of paper of Bob in VLDB after 2000.", in which the underline part can be canceled by the user. When the user changes the mapping strategy M to M', our system will immediately use M' to reprocess the second and third steps. Similarly, if the user choose T_i^M instead of T_1^M as the best valid parse tree, our system will fully instantiate T_i^M in the third step and update the interactions.*

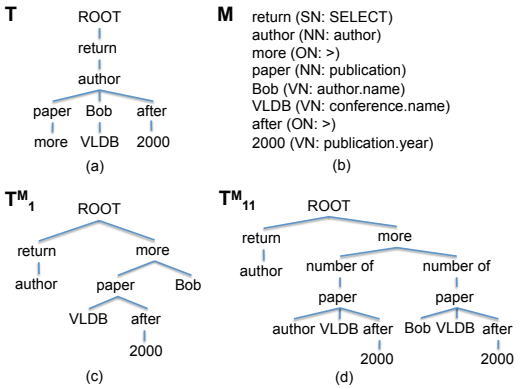


Figure 3: (a) Linguistic Parse Tree. (b) Node Mapping Strategy. (c) Valid Parse Tree. (d) Query Tree.

Query Tree Translator. Given the query tree verified by the user, the translator utilizes its structure to generate appropriate structure in the SQL expression and completes the foreign-key-primary-key (FK-PK) join paths. The result SQL statement may contain aggregate functions, multi-level subqueries, and various types of joins, among other things. Finally, our system evaluates the translated SQL statement against an RDBMS and returns the query results.

4. PARSE TREE NODE INTERPRETATION

To understand the linguistic parse tree from the database's perspective, we first need to identify the parse tree nodes that can be mapped to SQL components. Such nodes can be further divided into different types as shown in Figure 4, according to the type of SQL components they mapped to. The identification of select node, operator node, function node, quantifier node and logic node is independent of the database being queried. In NaLIR, enumerated sets of phrases are served as the real world "knowledge base" to identify these five types of nodes.

In contrast, name nodes and value nodes are entirely depend on the database being queried. For name nodes, we use the WUP similarity function [16] (based on Wordnet) to evaluate the similarity in meaning and adopt the square root of the Jaccard Coefficient to evaluate the similarity in spelling [17]. For value nodes, we only use the Jaccard Coefficienct to evaluate their similarity in spelling. When their similarity is above a predefined threshold, we say that the schema element/value is a candidate mapping for the node.

Node Type	Corresponding SQL Component
Select Node (SN)	SQL keyword: SELECT
Operator Node (ON)	an operator, e.g. =, <=, !=, contains
Function Node (FN)	an aggregation function, e.g., AVG
Name Node (NN)	a relation name or attribute name
Value Node (VN)	a value under an attribute
Quantifier Node (QN)	ALL, ANY, EACH
Logic Node (LN)	AND, OR, NOT

Figure 4: Different Types of Nodes.

1 Q -> (SClause)(ComplexCondition)*
2 SClause -> SELECT + GNP
3 ComplexCondition -> ON + (leftSubtree*rightSubtree)
4 leftSubtree -> GNP
5 rightSubtree -> GNP|VN|MIN|MAX
6 GNP -> (FN + GNP) | NP
7 NP -> NN + (NN)*(Condition)*
8 condition -> VN | (ON + VN)

\+ represents a parent-child relationship
* represents a sibling relationship

Figure 5: Grammar of Valid Parse Trees.

5. PARSE TREE STRUCTURE ADJUSTMENT

Given the correct mapping strategy, each node in the linguistic parse tree can be perfectly understood by our system. In this section, we infer the relationship between the nodes in the linguistic parse tree from the database's perspective and then understand the whole query. However, three obstacles lie in the way of reaching this goal.

First, the linguistic parse tree generated from an off-the-shelf parser may be incorrect. Second, the structure of the linguistic parse tree does not directly reflect the relationship between the nodes from the database's perspective. Consider the following three sentence fragments: (a) author who has more than 50 papers, (b) author who has more papers than Bob, and (c) author whose papers are more than those of Bob. The linguistic parse structures of these three sentence fragments are very different while their semantic meanings are similar. Third, natural language sentences often contain elliptical expressions. Take the parse tree in Figure 3 (c) as an example. Although the relationship between each pair of nodes is clear, it still has multiple possible interpretations.

In this section, we describe the construction of the Parse Tree Structure Adjustor in detail.

5.1 Query Tree

The semantic coverage of our system is essentially constrained by the expressiveness of SQL. So, given a database, we represent our semantic coverage as a subset of parse trees, in which each such parse tree explicitly corresponds to a SQL statement and all such parse trees could cover all possible SQL statements (with some constraints). We call such parse trees *Query Tree*s. As such, interpreting a natural language query (currently represented by a linguistic parse tree and the mapping for each its node) is indeed the process of mapping the query to its corresponding query tree in the semantic coverage.

We defined in Figure 5 the grammar of the parse trees that are syntactically valid in our system (all terminals are different types of nodes defined in Figure 4.). Query trees are the syntactically valid parse trees whose semantic meanings are reasonable, which will be discussed in Section 5.3, or approved by the user. Given the three obstacles in interpreting a linguistic parse tree, as we have discussed before, there is often a big gap between the linguistic parse tree and its corresponding query tree, which makes the mapping between them difficult. In our system, we take the following two strategies to make the mapping process accurate.

First, our system explains a query tree in natural language, which enables the user to verify it. Query trees are intermediates between natural language sentences and SQL statements. Thus the translation from a query tree to a natural language sentence is quite straightforward, compared to that from a SQL statement [5].

Second, given a natural language query, our system will generate multiple candidate query trees for it, which can significantly enhance the probability that one of them is correct. The problem is that, when the query is complex, there may be many candidate query trees, which are similar to each other. To show the user more candidate query trees without burdening them too much in verifying them, we do the mapping in two rounds and communicate with the user after each round. In the first round, we return the top k parse trees, which are syntactically valid according to the grammar defined and can be obtained by only reformulating the nodes in the parse tree. Each such parse tree represents a rough interpretation for the query and we call them valid parse trees. In the second round, if there are any implicit nodes, they will be inserted to the chosen (or default) valid parse tree to generate its exact interpretation. Our system inserts implicit nodes one by one under the supervision of the user. In such a way, suppose that there are k' possible implicit nodes in each of the k valid parse tree, the user only needs to verify k valid parse trees and k' query trees instead of all $k * 2^{k'}$ candidate query trees. Figure 3 (c) shows a valid parse tree generated in the first round, while this valid parse tree is full-fledged to the query tree in Figure 3 (d) after inserting implicit nodes.

5.2 Parse Tree Reformulation

In this section, given a linguistic parse tree, we reformulate it in multiple ways and generate its top k rough interpretations. The basic idea in the algorithm is to use subtree move operations to edit the parse tree until it is syntactically valid according to the grammar we defined. The resulting algorithm is shown in Figure 6. Each time, we use the function $adjust(tree)$ to generate all the possible parse trees in one subtree move operation (line 6). Since the number of possible parse trees grows exponentially with the number of edits, the whole process would be slow. To accelerate the process, our algorithm evaluates each new generated parse tree and filter out bad parse trees directly (line 11 - 12). Also, we hash each parse tree into a number and store all the hashed numbers in a hash table (line 10). By checking the hash table (line 8), we can make sure that each parse tree will be processed at most once. We also set a parameter t as the maximum number of edits approved (line 8). Our system records all the valid parse trees appeared in the reformulation process (line 13 - 14) and returns the top k of them for the user to choose from (line 15 - 16). Since our algorithm stops after t edits and retains a parse tree only if it is no

worse than its corresponding parse tree before the last edit (line 8), some valid parse trees may be omitted.

Algorithm 1: QueryTreeGen(*parseTree*)

```
1: results ← φ; HT ← φ
2: PriorityQueue.push(parseTree)
3: HT.add(h(tree))
4: while PriorityQueue ≠ φ do
5:   tree = PriorityQueue.pop()
6:   treeList = adjust(tree)
7:   for all tree' ∈ treeList do
8:     if tree' not exists in HT && tree'.edit < t then
9:       tree'.edit = tree.edit+1;
10:      HT.add(h(tree'));
11:      if evaluate(tree') >= evaluate(tree) then
12:        PriorityQueue.add(tree')
13:        if tree' is valid
14:          results.add(tree')
15: rank(results)
16: Return results
```

Figure 6: Parse Tree Reformulation Algorithm.

To filter out bad parse trees in the reformulating process and rank the result parse trees, we evaluate whether a parse tree is desirable from three aspects.

First, a good parse tree should be valid according to the grammar defined in Figure 5. Second, the nodes next to each other in the parse tree should be close to each other on the schema graph. Third, the parse tree should be similar to the original linguistic parse tree, which is measured by the number of the subtree move operations used in the transformation. When ranking the all the generated parse trees (line 15 in Figure 6), our system takes all the three factors into account. However, in the tree adjustment process (line 11), to reduce the cases when the adjustments stop in local optima, we only consider the first two factors, in which the first factor dominates the evaluation.

5.3 Implicit Nodes Insertion

Natural language sentences often contain elliptical expressions, which make some nodes in their parse trees implicit. In this section, for a rough interpretation, which is represented by a valid parse tree, we obtain its exact interpretation by detecting and inserting implicit nodes.

In our system, implicit nodes mainly exist in complex conditions, which correspond to the conditions involving aggregations, nestings, and non-FKPK join constraints. As can be derived from Figure 5, the semantic meaning of a complex condition is its comparison operator node operating on its left and right subtrees. When implicit nodes exist, such syntactically valid conditions are very likely to be semantically "unreasonable." To detect such unreasonable query logics, we define the concept of core node.

Definition 1 (Core Node). *Given a complex condition, its left (resp. right) core node is the name node that occurs in its left (right) subtree with no name node as ancestor.*

Inspired from [19], given a complex condition, we believe that its left core node and right core node are the concepts that are actually compared. So they should have the same type (map to the same schema element in the database). When they are in different types, we believe that the actual right core node, which is of the same type as the left core node, is implicit.

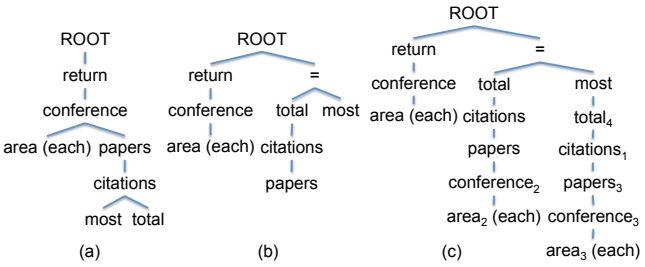


Figure 7: (a) Linguistic Parse Tree for Query 3 in Figure 1. (b) Valid Parse Tree. (c) Query Tree.

The name nodes in a left subtree are always related to the name nodes under the "SELECT" node. Take the parse tree in Figure 7 (b) as an example. The nodes "conference" and "area" are related to the nodes "citations" and "paper." In our system, we connect them by duplicating the name nodes under the "SELECT" node and inserting them to left subtree. Each of the nodes inserted in this step is considered as the same entity with its original node and marked "outside" for the translation in Section 6.

Furthermore, the constraints for the left core node and the right core node should be consistent. Consider the parse tree in Figure 3 (c). Its complex condition compares the number of papers by an author in VLDB after 2000 with the number of all the papers by Bob (in any year on any conference or journal), which is unfair. As such, the constraints of "in VLDB" and "after 2000" should be added to the right subtree. Note that the nodes duplicated from "outside" nodes are also marked "outside" and are considered corresponding to the same entity with the original nodes.

The last kind of implicit node is the function node. We consider two cases where function nodes may be implicit. First, the function node "count" is often implicit in the natural language sentences. Consider the parse tree in Figure 3 (c). The node "paper" is the left child of node "more" and it maps to the relation "Publication", which is not a number attribute. The comparison between papers is unreasonable without a "count" function. Second, the function nodes operating on the left core node should also operate on the right core node. Figure 7 (c) shows an example for this case. We see that the function node "total" operates on the left core node "citations" but does not operate on the right core node "$citations_1$." Our system detects such implicit function node and insert "$total_4$" to the right core node.

In our system, the detection and insertion of implicit nodes is just an inference of the semantic meaning for a query, which cannot guarantee the accuracy. As such, the whole process is done under the supervision of the user.

6. SQL GENERATION

In the cases when the query tree does not contain function nodes or quantifier nodes, which means the target SQL query will not have aggregate functions or subqueries, the translation is quite straightforward. The schema element mapped by the name node under the SELECT node is added to the SELECT clause. Each value node (together with its

```
1. Block 2: SELECT SUM(Publication.citation_num) as sum_citation,
2.                  Conference.cid, Domain.did
3.          FROM Publication, Conference, Domain, ConferenceDomain
4.          WHERE Publication.cid = Conference.cid
5.                  AND Conference.cid = ConferenceDomain.cid
6.                  AND ConferenceDomain.did = Domain.did
7.          GROUP BY Conference.cid, Domain.did

8. Block 3: SELECT MAX(block4.sum_citation) as max_citation,
9.                  block4.cid, block4.did
10.         FROM (CONTENT OF BLOCK4) as block4
11.         GROUP BY block4.did

12.Block 1: SELECT Conference.name, Domain.name
13.         FROM Conference, Domain, ConferenceDomain
14.                 (CONTENT OF BLOCK2) as block2
15.                 (CONTENT OF BLOCK3) as block3
16.         WHERE Conference.cid = ConferenceDomain.cid
17.                 AND ConferenceDomain.did = Domain.did
18.                 AND block2.citation_num = block3.max_citation
19.                 AND Conference.cid = block2.cid
20.                 AND Conference.cid = block3.cid
21.                 AND Domain.did = block2.did
22.                 AND Domain.did = block3.did
```

Figure 8: Translated SQL Statement

operation node if specified) is translated to a selection condition and added to the WHERE clause. Finally, a FK-PK join path is generated, according to the schema graph, to connect each name node and its neighbors. Such an FK-PK join path is translated into a series of FK-PK join conditions and all the schema elements in the FK-PK join path are added to the FROM clause.

When the query tree contains function nodes or quantifier nodes, the target SQL statements will contain subqueries. In our system, we use the concept of *block* to clarify the scope of each target subquery.

Definition 2 (Block). *A block is a subtree rooted at the select node, a name node that is marked "all" or "any", or a function node. The block rooted at the select node is the main block, which will be translated to the main query. Other blocks will be translated to subqueries. When the root of a block b_1 is the parent of the root of another block b_2, we say that b_1 is the direct outer block of b_2 and b_2 is a direct inner block of b_1. The main block is the direct outer block of all the blocks that do not have other outer blocks.*

Given a query tree comprising multiple blocks, we translate one block at a time, starting from the innermost block, so that any correlated variables and other context is already set when outer blocks are processed.

Example 2. *The query tree shown in Figure 7 (c) consists of four blocks: b_1 rooted at node "return", b_2 rooted at node "total", b_3 rooted at node "most", and b_4 rooted at node "total$_4$." b_1 is the main block, which is the direct outer block of b_2 and b_3. b_3 is the direct outer block of b_4. For this query tree, our system will first translate b_2 and b_4, then translate b_3 and finally translate the main block b_1.*

For each single block, the major part of its translation is the same as the basic translation as we have described. In addition, some SQL fragments must be added to specify the relationship between these blocks.

Example 3. *Consider the query tree in Figure 7 (c), whose target SQL statement is shown in Figure 8 (the block b_4 is omitted since it is almost the same as b_2). In the query tree, b_4 is included by b_2 while b_2 and b_3 are included by b_1 as direct inner blocks. Thus their corresponding subqueries are added to the FROM clause of their direct outer blocks (line 10 and line 14 - 15). The complex condition rooted at node "=" is translated to the condition in line 18. The nodes "conference$_2$", "area$_2$", "conference$_3$" and "area$_3$" are marked "outside" in the implicit node insertion, which means they correspond to the same entity as the nodes "conference" and "area" in the main block. These relationships are translated to the conditions in line 19 - 22.*

Relation	#tuples	Relation	#tuples
Publication	2.45 M	Author	1.25 M
cite	20.3 M	Domain	24
Conference	2.9 K	Journal	1.1 K
Organizations	11 K	Keywords	37 K

Figure 9: Summary Statistics for MAS Database.

7. EXPERIMENTS

The motivation of our system is to enable non-technical users to compose logically complex queries over relational databases and get perfect query results. So, there are two crucial aspects we must evaluate: the quality of the returned results (effectiveness) and whether our system is easy to use for non-technical users (usability).

Effectiveness. Evaluating the effectiveness of NaLIR is a challenging task. The objective in NaLIR is to allow users to represent SQL statements using natural language. In our experiments, the effectiveness of our system was evaluated as the percentage of the queries that were perfectly answered by our system. (Note that this is a stiff metric, in that we get zero credit if the output SQL query is not perfect, even if the answer set has a high overlap with the desired answer). Since the situations where users accept imperfect/wrong answers would cause severe reliability problems, for the cases when the answers were wrong, we recorded whether the users were able to recognize such failures, whether from the answers themselves or from the explanations generated by our system. Also, for the failure queries, we analyzed the specific reasons that caused such failures.

Usability. For the correctly processed queries, we recorded the actual time taken by the participants. In addition, we evaluated our system subjectively by asking each participant to fill out a post-experiment questionnaire.

7.1 Experiments Design

The experiment was a user study, in which participants were asked to finish the query tasks we designed for them.

Data Set and Comparisons. We used the data set of Microsoft Academic Search (MAS). Its simplified schema graph and summary statistics are shown in Figure 1 and Figure 9, respectively. We chose this data set because it comes with an interesting set of (supported) queries, as we will discuss next.

We compared our system with the faceted interface of the MAS website. The website has a carefully designed ranking system and interface. By clicking through the site, a user is able to get answers to many quite complex queries. We enumerated all query logics that are "directly supported" by the MAS website and can be accomplished by SQL state-

Easy: Return all the conferences in database area.
Medium: Return the number of papers in each database conference.
Hard: Return the author who has the most publications in database area.

Figure 10: Sample Queries in Different Complexity.

ments. "Directly supported" means that the answer of the query can be obtained in a single webpage (or a series of continuous webpages) without further processing. Through exhaustive enumeration, we obtained a set of 196 query logics and we marked their complexity according to the levels of aggregation/nesting in their corresponding SQL. Sample queries with different complexity are shown in Figure 10. In the query set, the number of easy/medium/hard queries are 63/68/65, respectively.

A central innovation in NaLIR is the user interaction as part of query interpretation. To understand the benefit of such interaction, we also experimented with a version of NaLIR in which the interactive communicator was disabled, and the system always chose the default (most likely) option.

Participants. 14 participants were recruited with flyers posted on a university campus. A questionnaire indicated that all participants were familiar with keyword search interfaces (e.g. Google) and faceted search interfaces (e.g. Amazon), but had little knowledge of formal query languages (e.g. SQL). Furthermore, they were fluent in both English and Chinese.

Procedures. We evenly divided the query set into 28 task groups, in which the easy/medium/hard tasks were evenly divided into each task group. This experiment was a within-subject design. Each participant randomly took three groups of tasks and completed three experimental blocks. In the first (resp. second) experimental block, each participant used our system without (with) the Interactive Communicator to accomplish the tasks in her first (second) task group. Then in the third experimental block, each participant used the MAS interface to do her third task group. For each task group, the participants started with sample query tasks, in order to get familiar with each interface.

For our system, it is hard to convey the query task to the participants since any English description would cause bias in the task. To overcome this, we described each query task in Chinese and asked users to compose English query sentences. Since English and Chinese are in entirely different language families, we believe this kind of design can minimize such bias.

7.2 Results and Analysis

Effectiveness. Figure 11 compares the effectiveness of our system (with or without the interactive communicator) with the MAS website. As we can see, when the interactive communicator was disabled, the effectiveness of our system decreased significantly when the query tasks became more complex. Out of the 32 failures, the participants only detected 7 of them. Actually, most of undetected wrong answers were aggregated results, which were impossible to verify without further explanation. In other undetected failures, the participants accepted wrong answers mainly because they were not familiar with what they were querying. In the 7 detected failures, although the participants were aware of the failure, they were not able to correctly reformulate the queries in the time constraint. (In 5 of the detected failures, the participants detected the failure only

	with Interaction	without Interaction	MAS
Simple:	34/34	26/32	20/33
Medium:	34/34	23/34	18/32
Hard:	20/30	15/32	18/33

Figure 11: Effectiveness.

	Mapper	Reformulation	Insertion	Translation
w/o Interaction	15	19	0	0
with Interaction	0	10	0	0

Figure 12: Failures in each Component.

because the query results were empty sets). The situation got much better when the interactive communicator was involved. The users were able to handle 88 out of the 98 query tasks. For the 10 failed tasks, they only accepted 4 wrong answers, which was caused by the ambiguous (natural language) explanations generated from our system. In contrast, the participants were only able to accomplish 56 out of the 98 tasks using the MAS website, although all the correct answers could be found. In the failure cases, the participants were simply not able to find the right webpages, which often required several clicks from the initial search results.

Figure 12 shows the statistics of the specific components that cause the failures. We can see that our system could always correctly detect and insert the implicit parse tree nodes, even without interactive communications with the user. Also, when the query tree was correctly generated, our system translated it to the correct SQL statement. When the interactive communicator was enabled, the accuracy in the parse tree node mapper improved significantly, which means for each the ambiguous parse tree node, the parse tree node mapper could at least generate one correct mapping in the top 5 candidate mappings, and most importantly, the participants were able to recognize the correct mapping from others. The accuracy in parse tree structure reformulation was also improved when the participants were free to choose from the top 5 candidate valid parse trees. However, when the queries were complex, the number of possible valid parse trees was huge. As a result, the top 5 guessed interpretations could not always include the correct one.

Usability. The average time needed for the successfully accomplished query tasks is shown in Figure 13. When the interactive communicator was disabled, the only thing a participant could do was to read the query task description, understand the query task, translate the query task from Chinese to English and submit the query. So most of the query tasks were done in 50 seconds. When the interactive communicator was enabled, the participants were able to read the explanations, choose interpretations, reformulate the query according to the warnings, and decide to whether to accept the query results.

It is worth noting that, using our system (with interactive communicator), there was no instance where the participant became frustrated with the natural language interface and abandoned his/her query task within the time constraint. However, in 9 of the query tasks, participants decided to stop the experiment due to frustration with the MAS website. According to the questionnaire results, the users felt that MAS website was good for browsing data but not well de-

	with Interaction	without Interaction	MAS
Simple:	48	34	49
Medium:	70	42	67
Hard:	103	51	74

Figure 13: Average Time Cost (s).

signed for conducting specific query tasks. They felt NaLIR can handle simple/medium query tasks very well but they encountered difficulties for some of the hard queries. In contrast, the MAS website was not sensitive to the complexity of query tasks. Generally, they welcomed the idea of an interactive natural language query interface, and found our system easy to use. The average level of satisfaction with our system was 5, 5 and 3.8 for easy, medium, and hard query tasks, respectively, on a scale of 1 to 5, where 5 denotes extremely easy to use.

8. RELATED WORK

The problem of constructing Natural Language Interfaces to DataBases (NLIDB) has been studied for several decades, starting from early systems based on manually crafted grammars [1]. While quite successful in their specific scenario, the manually crafted grammars are hard to scale up, both to other domains and to new natural language expressions.

Later, researchers begin to build NLIDBs that can learn semantic grammars from training examples, in which the grammars can be improved incrementally by adding new examples [3, 4, 14, 15, 18]. A major obstacle is that these methods depend crucially on having examples of natural language queries paired with meaning representations, which requires substantial human effort to obtain [10].

As pointed out in [13], the major usability problem in NLIDBs is its limited reliability. Natural language sentences do not have formally defined semantics. The goal of NLIDBs is to infer the user's query intent, which cannot guarantee accuracy due to ambiguities. To deal with the reliability issue, PRECISE [12, 13] defines a subset of natural language queries as semantically tractable queries and precisely translates these queries into corresponding SQL queries. However, natural language queries that are not semantically tractable will be rejected by PRECISE.

The idea of interactive NLIDB was discussed in previous literature [1, 6, 8]. Early interactive NLIDBs [1, 6] mainly focus on generating cooperative responses from query results (over-answering). NaLIX [8] takes a step further, generating suggestions for the user to reformulate her query when it is beyond the semantic coverage. This strategy greatly reduces the user's burden in query reformulation. This paper is a short version of NaLIR [7], to which given a natural language query, we explain to the user how we process her query and interactively resolve ambiguities with the user. As a result, under the supervision of the user, our system could confidently handle rather complex queries.

9. CONCLUSION AND FUTURE WORK

We have described an interactive natural language query interface for relational databases. Given a natural language query, our system first translates it to a SQL statement and then evaluates it against an RDBMS. To achieve high reliability, our system explains to the user how her query is actually processed. When ambiguities exist, for each ambiguity, our system generates multiple likely interpretations for the user to choose from, which resolves ambiguities interactively with the user. The query mechanism described in this paper has been implemented, and actual user experience gathered. Using our system, even naive users are able to accomplish logically complex query tasks, in which the target SQL statements include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things.

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

[3] R. Ge and R. Mooney. A statistical semantic parser that integrates syntax and semantics. In *CoNLL*, pages 9–16, 2005.

[4] R. J. Kate and R. J. Mooney. Using string-kernels for learning semantic parsers. In *ACL*, 2006.

[5] 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.

[6] D. Küpper, M. Strobel, and D. Rösner. Nauda - a cooperative, natural language interface to relational databases. In *SIGMOD Conference*, pages 529–533, 1993.

[7] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

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

[9] Y. Li, H. Yang, and H. V. Jagadish. Nalix: A generic natural language search environment for XML data. *ACM Trans. Database Syst.*, 32(4), 2007.

[10] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[11] M. Minock. A STEP towards realizing codd's vision of rendezvous with the casual user. In *PVLDB*, pages 1358–1361, 2007.

[12] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

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

[14] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *EMCL*, pages 466–477, 2001.

[15] Y. W. Wong and R. J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

[16] Z. Wu and M. S. Palmer. Verb semantics and lexical selection. In *ACL*, pages 133–138, 1994.

[17] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.

[18] J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, 1996.

[19] Y. Zeng, Z. Bao, T. W. Ling, H. V. Jagadish, and G. Li. Breaking out of the mismatch trap. In *ICDE*, pages 940–951, 2014.