Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Information Systems 31 (2006) 33–56

www.elsevier.com/locate/infosys

Conversational querying

Yannis E. Ioannidis[a], Stratis D. Viglas[b,*]

[a]Department of Informatics & Telecommunications, University of Athens, Athens, Hellas, Greece
[b]School of Informatics, University of Edinburgh, Crichton Street, Edinburgh EH8 9LE, UK

Received 21 February 2003; received in revised form 16 July 2004; accepted 25 September 2004

Abstract

The traditional interaction mechanism with a database system is through the use of a query language, the most widely used one being SQL. However, when one is facing a situation where he or she has to make a minor modification to a previously issued SQL query, either the whole query has to be written from scratch, or one has to invoke an editor to edit the query. This, however, is not the way we converse with each other as humans. During the course of a conversation, the preceding interaction is used as a context within which many incomplete and/or incremental phrases are uniquely and unambiguously interpreted, sparing the need to repeat the same things again and again. In this paper, we present an effective mechanism that allows a user to interact with a database system in a way similar to the way humans converse. More specifically, incomplete SQL queries are accepted as input which are then matched to identified parts of previously issued queries. Disambiguation is achieved by using various types of semantic information. The overall method works independently of the domain under which it is used (i.e., independently of the database schema). Several algorithms that are variations of the same basic mechanism are proposed. They are mutually compared with respect to efficiency and accuracy through a limited set of experiments on human subjects. The results have been encouraging, especially when semantic knowledge from the schema is exploited, laying a potential foundation for conversational querying in databases.

Keywords: Databases; SQL; Human computer interaction; Information retrieval

1. Introduction

Suppose someone asks us the following question:

What time does CS207 start?

and after our answer a second question is immediately posed:

CS507?

We find no difficulty in interpreting the incomplete second question as:

What time does CS507 start?

In most cases, such partial questions are completed by the participants of a conversation, resulting in unambiguous new questions that they

[*]Corresponding author. Tel.: +44 131 650 2691; fax: +44 131 650 6513.

E-mail addresses: yannis@di.uoa.gr (Y.E. Ioannidis), sviglas@inf.ed.ac.uk (S.D. Viglas).

0306-4379/$ - see front matter © 2004 Elsevier B.V.. All rights reserved.
doi:10.1016/j.is.2004.09.002

have no trouble answering. Humans have the ability to use several factors towards completing such partial questions. Common-sense knowledge is one of them. The context of the conversation is another.

When interacting with a database management system (DBMS), we essentially start a dialogue shell with the DBMS's front end, where user and system 'speak' the same language (typically SQL). In traditional ad hoc DBMS interaction, consecutive queries are unrelated to each other, usually. Cases where the next query is an alteration of the previous one are relatively few. Therefore, it is acceptable for the user to either rewrite the whole query from scratch (if it is small), or invoke an editor and make the desired modifications.

When one considers, however, some of the applications of database technology, i.e., data mining and decision support, the situation changes dramatically. The user essentially explores the data and obtains different views or subspaces of it so that important patterns or other characteristics may be identified. This is achieved by issuing sequences of interrelated queries, which tend to be large and complicated. Rewriting every query from scratch is then out of the question, while editing the previous query becomes quite tedious and counterproductive. Whereas the user should be operating in a continuous [QUERY-ANALYZE]* cycle focusing on data exploration (Fig. 1(a)), he/she is operating in a [QUERY-ANALYZE-EDIT]* cycle (Fig. 1(b)). Instead of posing to the system the query that comes to his/her mind, the user has to enter the editor, find the appropriate places in the query text, make the necessary changes, and only then submit the new query. This forces a continuous context switch in the mental operation model of the user, reducing his/her effectiveness.

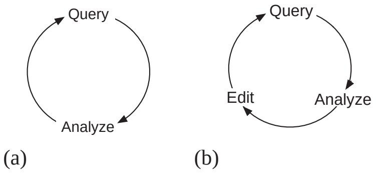


Fig. 1. The two exploration modes. (a) Uninterrupted exploration mode. (b) Exploration mode interrupted by editing.

It would be better if the human–DBMS interactions were similar to those between humans conversing, as illustrated above. The user would only have to give so much as a 'hint' to the system, just the new/different part of the query. The system would then have to understand what this hint implies and make all the necessary alterations to the original query, producing the next one. Effectively operating in the two-step cycle (Fig. 1(a)), the user's attention would thus be devoted exclusively to data exploration. We call this form of interaction with a DBMS *Conversational Querying*.

Conversational Querying is not only meaningful and desirable for textual–language interactions but for visual ones as well. First, although more pleasant than its SQL counterpart, editing of visual queries remains an interruption in the user's flow of thought, an artificial extra step in every cycle of exploration (Fig. 1(b)) that is better eliminated. Second, at the visual level, there are many human–DBMS interaction styles potentially available whose nature is such that offering them to users requires support of Conversational Querying as well. For example, consider a relation EXPERIMENT(w, x, z, y), containing the results of some experiments, i.e., containing the value of the output parameter y for various values of the input parameters w, x, and z (the composite primary key of the relation). Assume that the result of a query

```
select x,y,z
from EXPERIMENT
where (x mod 10 = 0) and (w = 3.14159)
```

is visualized as in Fig. 2(a). Whether the original query was posed in textual form as above or through some visual query tool is not relevant to this discussion. Seeing the resulting graphs, a user is quite likely to request more points in the area of x between 30 and 50, since the value of y changes dramatically there. A rather natural way to do this is for the user to indicate the area of interest *directly on the result visualization* as in Fig. 2(b). The system then takes into account the overall

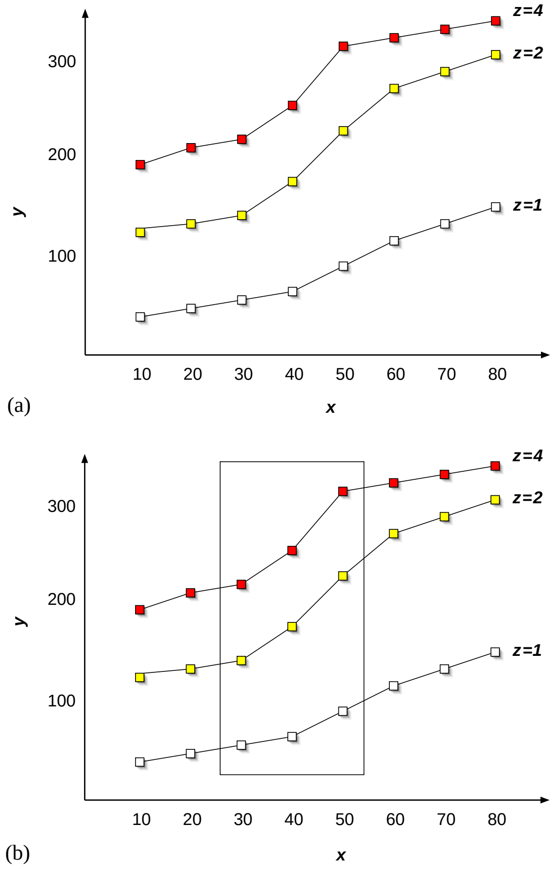


Fig. 2. Visual querying. (a) Visualization of original query results. (b) Visual conversational query using visualization of previous results.

context of the user's action and generates the query

```
select *
from EXPERIMENT
where (x mod 2 = 0) and (x between 30 and
50) and (w = 3.14159)
```

to obtain the requested data.[1] Clearly, this query is simply an alteration of the previous one. Essentially, the user is engaged in (visual) Conversational Querying simply having to specify the difference of the two queries.

This paper formulates Conversational Querying and introduces a mechanism to support it. The focus is on a core subset of SQL queries that is critical in any attempt to deal with the problem. Having devised an overall approach that works on this subset, it is rather straightforward to generalize it to work on other textual queries as well. Furthermore, as demonstrated above, this mechanism should form the basis for supporting Conversational Querying in the context of visual query tools. The approach is generic and domain independent. It is based on specific manipulations of the parse trees of SQL queries and identifying interchangeable parts of subtrees. Whenever necessary, disambiguation is achieved through the exploitation of semantic knowledge. Although the use of parse trees is quite generic and one can argue that the methodology can be applied to any unambiguous grammar, the incorporation of semantic knowledge into the disambiguation scheme is what tailors the application specifically to query languages and in particular SQL. Preliminary experiments have been carried out, giving an encouraging head start for the development of a Conversational Querying mechanism in contemporary DBMSs.

2. Related work

Natural language processing: Conversation is a common characteristic of human interaction. As such, problems relevant to the ones we address have been given considerable attention in natural language processing. In particular, the two most common notions are *ellipsis* and *anaphora resolution* [2–6]. The case for ellipsis arises when, in the context of a conversation, words or phrases are missing in a given and/or subsequent sentences. The *antecedent* is what is mentioned in earlier sentences and is missing (i.e., implied) in later ones. Anaphora resolution (also known as ellipsis resolution) is the process of identifying the missing terms in order to give the sentence complete semantics. Key concepts in resolution are text cohesion (relations between words or expressions) and text coherence (relations between clauses or sentences) [7].

[1] We are actually in the process of developing a system with such functionality in the context of the ZOO project on desktop experiment management [1].

Given the inherent complexity of natural language, there is no uniform way of addressing the issue of resolution, but different strategies are used for different types of phrases. Nevertheless, we can identify some common underlying principles by using the seminal work of Lakoff and Johnson [8] as a foundation. There the authors present the concept of a *metaphor* as central to expression. Using metaphors, one can 'cross' from what is termed the *conceptual expression*, i.e., what someone thinks, to what is termed the *linguistic expression*, i.e., what someone says using natural language. Unfortunately, no meaning can be characterized as 'objective' or 'absolute', so in general, there exist multiple metaphors for the same conceptual or linguistic expression. What makes natural language understanding possible is (a) the fact that there are common conceptual metaphors shared by the participants in a conversation, which allow them to move from related concepts to seemingly unrelated and different linguistic expressions without sacrificing their common understanding, and (b) the fact that metaphors mean consistently within a text (though variously in many texts), i.e., what is usually called *metaphorical coherence*.

The whole process can be visualized by the top part of Fig. 3. Conceptual expression can be mapped to linguistic expression using different metaphors. Metaphors can be thought of as functions that allow this crossing between the conceptual and the linguistic realms. Once a metaphor is chosen in the context of a conversation (highlighted by a shaded oval in Fig. 3, metaphorical coherence 'guarantees' that it will remain constant throughout the conversation.

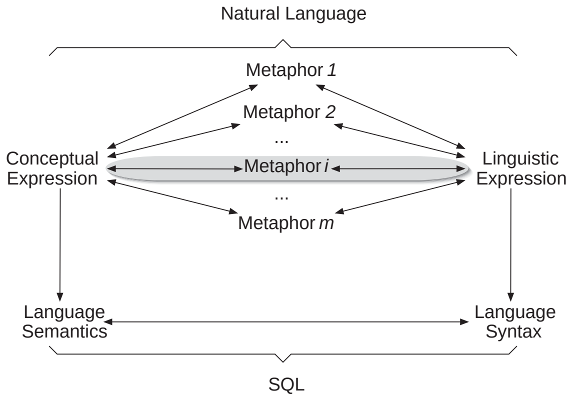


Fig. 3. The difference between natural language and SQL.

According to the above discussion, it is central to any language processing system built on conversational principles to identify the *context* of a conversation. The reason is that, by identifying the context, one is able to identify the metaphor used. Context identification has been traditionally treated in the natural language processing literature as an issue of semantics, due to the complexity of the language. For example, one possible way of addressing the problem at the semantic level is through the use of *centering* [9], where local linguistic changes are interpreted at the greater discourse coherence level. Other approaches include studies like [10,11]. The latter is based on the computing and functional programming foundations of λ-calculus, as are *Montague* grammars [12], which attempt to assign formal semantics to a language. In all these studies, the goal is to identify the semantics of a discourse. This leads to identification of the relevant metaphor used, which permits 'all' linguistic expressions to be interpreted unambiguously.

When dealing with human–computer communication using a restricted language, like SQL, one may draw an analogy to natural language communication, with semantics playing the role of conceptual expression and syntax playing the role of linguistic expression. The transition, however, from semantics to syntax in SQL, i.e., the 'metaphor' is straightforward and does not suffer from the complexities of natural language. In this paper, in particular, we deal with SQL queries of the following generic syntax form:

`select` *attribute list*
`from` *table list*
`where` *condition list*.

Consider the case where a database schema is fixed. Tables can only appear in the *from*-clause, attributes can only appear in the *select*-clause while conjuncts or disjuncts of predicates appear in the *where*-clause. Each type can be easily identified at the syntax level, without having to do any semantic inference.[2]

Conversational querying in an artificial language like SQL requires context identification just as in natural languages. The main difference is that, in this case, there is only one metaphor between the language semantics and its syntax, and therefore, we do not have to deal with any complex or ambiguous mappings between conceptual and linguistic constructs. This is illustrated at the bottom part of Fig. 3, Our thesis is that, by focussing on the syntactic representation of queries, we are able to achieve our goal. Our claim is re-enforced by the results of the experiments we conducted with human subjects. In none of the experiments was there any obvious need for more complicated coherence mechanisms to be introduced. Furthermore, the system achieved high performance for our metrics (see also Section 5.4), providing further evidence of the applicability of our approach in our restricted environment.

Databases: In the database field, there have been efforts of providing natural language interfaces as querying front-ends, with IBM's USL [13] and the studies of [14,15] being prominent examples. The focus of USL was on using well-formed, unambiguous natural language-like constructs in order to directly query and manipulate the database. As such, it can be thought of as a more user-friendly data definition and manipulation language for relational databases. Indeed, USL was aimed at the non-technical user. Being less complicated than standard SQL, there were opportunities of addressing ellipsis resolution, albeit at a smaller and rather different scale than in this paper. Our approach is quite different: first, we do not deal with natural language at all; we deal with the standard querying language used for relational databases, SQL. Second, in USL the effort was to provide a core of grammar rules that were independent of any natural language (e.g., Spanish or German in [13]) so that the front-end would be portable across users speaking different native languages. The expressive power of the underlying grammar was therefore restricted and would not be easily expanded to handle complicated querying constructs. We focus on the query language itself and its inherent constructs; this renders our approach capable of handling a number of complicated querying interactions with the database.

In [14] the authors try to address the problem of natural language database front-ends by coupling the interface with the database schema. Given this observation, the user can be presented with a richer language interface, but the approach suffers in terms of applicability to different schemata: the natural language interface has to change in order to adhere to another schema. In contrast, our approach is generic and independent of the database schema.

The authors of [15] address mainly the issue of semantics by taking a significantly different direction. Instead of focussing on the language level, they focus on the database level and enrich the meta-data stored in the database to account for semantic information of the stored data. As such, the mapping from natural language to SQL is easier since all the disambiguation information is essentially stored in the database itself. Our approach differs in that we do not aim at making any modifications to the storage requirements of a system. Rather, by operating strictly on the user interface level and performing minimal semantic checks outside of the database engine, we are able to address queries of a good subset of SQL.

In the context of disambiguation for databases, the closest issue to Conversational Querying is that of disambiguating queries with incomplete path expressions or missing foreign-key joins [16,17]. Beyond a superficial similarity, however, in that they both deal with queries that are not syntactically complete and valid, the two issues are rather different.

[2] The situation is slightly more complex when it comes to values (e.g., in selections in the *where*-clause), where treatment of each value requires going beyond just the syntactic level. With careful refinement, however, this problem can be reduced to a rather straightforward range matching problem. For example, consider the case of a database with birth-year and age as two attributes. In such a scenario, it is rather easy to differentiate between the two, as values of each type will have quite different ranges. Such a simple semantic matching strategy is what we have implemented in our system (see also Section 4.3.1).

Conversational querying has some characteristics common with the areas of *cooperative query answering* [18,19] and *associative query answering* [20,21]. CoBase [18] is an important representative of such systems. It enlarges the scope of unsuccessful queries by broadening the search range to areas near those of the original query in the hope of making them successful. This query rewriting is achieved mainly through the use of semantic knowledge (the database schema) and the use of a learning mechanism. Both these aspects are related to our work in the sense that we use similar mechanisms for conversational querying. The specific goals of the two efforts, however, as well as the details are quite different. First, our emphasis is on how a user can pose queries more efficiently, while in CoBase and other cooperative systems, the focus is on the query answer. Second, the use of semantic knowledge by CoBase is critical to the validity of the system; that is, the approach used is domain dependent and without any schema information it cannot produce results. On the other hand, our use of semantic knowledge simply enhances the effectiveness of our approach, which is in principle, domain independent. Third, learning in CoBase attempts to discover associations of semantic (schema) concepts the user makes in queries, which can then be used for query enhancements. On the other hand, the goal of learning in our work is to discover the particular way of thinking by a user with respect to some particular aspects of conversational querying. Hence the techniques used by the two systems are quite different. Fourth, we should also mention that, to the best of our knowledge, no experiments have been conducted on human subjects within CoBase, something that has been very important in our work to provide usability and performance metrics.

Other: Another marginally related area is *collaborative discourse* systems [22,23]. In these systems, a dialogue exists between the user and the system. The system tries to accomplish a goal, and whenever a subgoal is undecidable, the system asks for the user's help to provide more facts so it can reach a decision. Experiments on human subjects have been conducted, but the concepts they wanted to prove and the metrics they used were entirely different from ours so no direct comparison seems plausible.

3. Problem formulation

3.1. Data requests

In a typical exploratory interaction with a DBMS, the requests that a user poses to the system are of two different kinds:

Initialization request: This is the first request in the exploration of a new vein of thought. Typically, this is a conventional, complete database query, containing several selections, joins, and other operators on the database.

Follow-up request: This is a new request that is very similar to the one immediately before it, and has the answers of the latter as a reference point. In most cases, the conceptual formulation of such a query by the user is through the previous query and consists only of the necessary modifications to it, i.e., the query is incomplete. Hence, a follow-up request is *context sensitive*, where the previous queries form its context. The need for such requests is due to the nature of data exploration: users essentially navigate, step by step, through a multi-dimensional space of parameters that define the data space under study. Consequently, this type of request represents the most common during exploratory data interactions.[3]

A system supporting Conversational Querying should accept as its input both complete and incomplete queries. Both types of queries should be translated into a uniform way of representation. In the case of a complete query (initialization request), the system should pass it through for processing as is. In the case of an incomplete query (follow-up request), the system should use the common representation of this and all relevant previous queries to identify the necessary correspondences between parts of them, modify some earlier query based on the new one so that the latter becomes complete. If it is successful and

[3] In what follows, any combination of the terms *follow-up* or *incomplete* with *request* or *query* are considered synonyms and are used interchangeably.

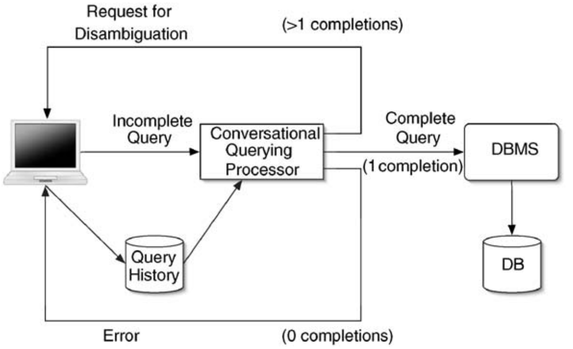


Fig. 4. Architecture and data flow of a system supporting conversational querying.

identifies one such completion, it should pass it for processing. If it identifies multiple such completions (a case of ambiguity), it should go back to the user for more information. Finally, if it identifies no such completion, it should return an error. The architecture and data flow of a system supporting Conversational Querying is depicted in Fig. 4.

3.2. Internal representation

To support Conversational Querying, a system should employ an internal representation for queries that has the following properties:

- Each query, complete or incomplete, should have a unique translation to the representation. The representation should be consistent, i.e., whether the query is complete or incomplete should not affect the nature of its translation to the representation.
- The representation should be easily manageable. It should facilitate the system in recognizing the correspondence of query parts and identifying key elements of the query (such as keywords), and provide a basis for an efficient matching algorithm.
- The query-to-representation translation should be reversible: given an instance of the internal representation, the corresponding query should be easily reconstructed.

Given the aforementioned requirements, we have chosen the *parse tree* that corresponds to the SQL query as the representation. It fulfills the above as follows:

- The grammar of SQL leads to unambiguous parse trees. By extending the grammar, it is possible to handle both initialization and follow-up requests in a uniform way. Every symbol in the grammar is denoted as a start symbol. In this way, a parser will consider as correct input any meaningful part of an SQL query.
- Traversing a parse tree is well-defined (e.g., depth-first, breadth-first, pre-ordered, post-ordered). In fact, the ways in which a parse tree can be traversed are a point of study in the context of this paper. Furthermore, annotations can be used to differentiate the nodes of the parse tree for faster recognition.
- If the parse tree is well-annotated, a pre-order traversal will lead to the construction of the SQL query that generates it.
- Moreover, the parse-tree choice seems to have a natural relationship with the way users think. In particular, users tend to modify elements of queries that are of a semantic nature. These elements usually correspond to terminal symbols in the grammar, and the parse tree incorporates this notion in its structure (i.e., terminal symbols become leaf nodes).

Although simple, this representation proves sufficiently powerful, yielding quantitatively satisfactory results, as shown in Section 5.

3.3. Query focus

The approach that is proposed in this paper is based upon the identification of a *focus* for each query, which can be thought of as the *context* of a conversation in natural language processing terms (see also Section 2). This is a specific part of a query that is used to identify the context of the subsequent incomplete queries. The system tries to interpret such queries within the current focus. For instance, given an initial complete query of the form

```
select library_name
from libraries
```

`where city = 'Detroit' and state = 'Michigan'`

and a follow-up request of the form

`city = 'East Lansing'`

the clause `city = 'East Lansing'` becomes the system's current focus. If a subsequent query of the form

`'Holland'`

is issued, then this query is handled within the current focus and interpreted as looking for libraries in the city of Holland, MI, as opposed to the city East Lansing in Holland (the Netherlands)!

More formally, the basic concepts that surround the focus of a query are defined as follows:

Focus: Any section of an SQL query that acts as the context within which we interpret subsequent follow-up queries. At the representation level, a focus is a subtree of a valid SQL parse tree.

Focal point: This is the point of origin of a focus. At the representation level, it is the root node of the parse tree that corresponds to the focus.

Incomplete query: This is any part of a valid SQL query that can act as a focus, i.e., that corresponds to a symbol of the SQL grammar treated as a start symbol.

Completed query: This is a query that results after the modifications, which the system decides should take place in an earlier query, are carried out.

The following are the main operations defined on the focus. They are visually depicted in Fig. 5.

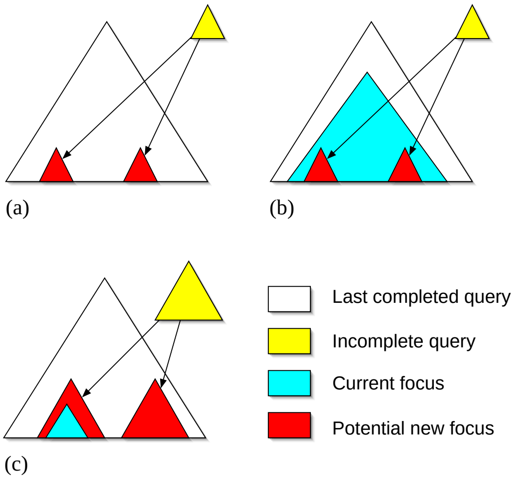


Fig. 5. Focus handling: (a) identification, (b) interpretation, and (c) expansion and relocation of the focus.

- *Identification*: Given an earlier complete query Q_c and a submitted incomplete query Q_i, all possible positions within Q_c that may correspond to Q_i are identified. One of them will be chosen as the focus and will be replaced by Q_i to form the new complete query. Given the use of the parse tree as our internal representation for both complete and incomplete queries, these positions are some non-terminal nodes of the tree. The kind of information needed to decide which position is most appropriate is based on the query structure, type of information, as well as the data semantics of the values appearing in the query.
- *Interpretation*: This is identical to identification only when a focus already exists, i.e., when the current focus is something less than a complete query. The focus can either remain the same or be narrowed (i.e., move to a *sub-focus*).
- *Expansion* and *relocation* of the focus: If an incomplete query cannot be interpreted within the system's current focus, there are two possibilities. First, the focus may be directly expanded (move to a *super-focus*). That is, the focus remains within the same query but is enlarged to include more information. Equivalently, the focal point moves up the parse tree, on the path connecting the current focus to the root. Second, the focus may be relocated. This occurs when neither a sub-focus nor a super-focus can be identified as the new focus. The new focus can either exist within the same complete query's parse tree, i.e., on a previously unexplored branch, or on a completely different parse tree. The latter requires that the system maintains a *repository* of previously issued as well as previously completed parse trees.

3.4. Behavior examples

In this section, we present a few examples that provide more insight into the system's desired behavior. These examples are merely fragments of larger queries, which are not given here to avoid unnecessary clutter. They are all taken on the following simple self-explanatory schema on libraries (primary keys in italics):

Library(*lname*, city)
Author(*name*, citizenship, byear, bplace)
Book(*isbn*, title, author, publisher, pyear, pplace)
Customer(*name*, byear, city)
Stock(*isbn*, *lname*, quantity)
Bindex(*isbn*, *subject*)
Publisher(*pname*, city, country, eyear)

3.4.1. *Focus identification*

Consider the complete query

```
select author.name, bindex.isbn,
author.citizenship
from bindex, author, publisher, book
where author.byear >= 1967 and
  bindex.subject = ‘Query Processing’
  and
  author.bplace = ‘Athens’
  book.pplace = publisher.city
  ...
```

which tries to retrieve names of authors born after 1967 who have written books on query processing, who were born in Athens and whose books were published in the same city where the publisher resides in. If it is followed by the incomplete one

```
‘The Theory of Semi-Joins’
```

The system should be in a position to understand that, at the representation level, two focuses can be identified (`‘Athens’` and `‘Query Processing’`). Although for the human the implied focus is obvious, the system may not necessarily tell them apart if it regards them both simply as strings. This is an instance of *population ambiguity*, which is discussed in subsequent sections.

3.4.2. *Focus interpretation*

Consider the query

```
select book.pplace, library.city,
customer.name
from book, library, stock, customer
where book.subject = ‘Query
Processing’ and
      stock.quantity = 3 and
      book.author = ‘DeWitt’ and
      book.pyear >= customer.byear
      ...
```

which tries to retrieve books about query processing written by DeWitt, the stock quantity of which is equal to 3 and which have been borrowed by customers who were born after the book was published. If it is followed by the request

```
stock.quantity > 4 and
book.author != ‘DeWitt’
```

and then followed by

```
= ‘Codd’
```

the system should understand that (after both follow-up requests) the user implies the query:

```
select book.pplace, library.city,
customer.name
from book, library, stock, customer
where book.subject = ‘Query
Processing’ and
      stock.quantity > 4 and
      book.author = ‘Codd’ and
      book.pyear >= customer.byear
      ...
```

This is not the only query, however, that could be considered valid. From the initial complete query, another completion can be deduced:

```
select book.pplace, library.city,
customer.name
from book, library, stock, customer
where book.subject = ‘Query
Processing’ and
      stock.quantity = 3 and
      book.author = ‘Codd’ and
      book.pyear >= customer.byear
      ...
```

This is an instance of *chronological ambiguity*. It stems from whether we interpret a follow-up request within the context of the *last completed* query (which is the immediately previous one) or the *last issued* query (which is the most recent query whose focus was given by the user and is enclosing the current one). This is also a matter of discussion in subsequent sections.

3.4.3. Focus expansion or relocation

Sometimes, it is impossible to identify a focus within a given one. In this case, the focus should be expanded or relocated. For example, consider the query

```
select library.city, bindex.subject,
library.lname
from book, bindex, library, author,
publisher
where library.lname = 'Gustav Library'
and
   author.bplace = library.city and
   book.pplace = publisher.city and
   bindex.isbn = 857
   ...
```

which tries to find a book that is available in Gustav Library, which is published in the same city as the publisher's, and whose author was born in the same city that the library is in. If it is followed by the requests (in this order)

1. `bindex.isbn = 758`
2. `book.pplace < > publisher.city and bindex.isbn = 123`

then the system would not be able to identify a sub-focus, since after the first follow-up request, the focus would have been set to `bindex.isbn = 758`. The system would need to expand that focus. Likewise, if the user then issued the incomplete query

```
library.lname = 'Harvard Library'
```

the system would have to relocate the focus to reflect this new follow-up request, since neither narrowing it nor expanding it works.

3.4.4. Additional features

The examples given in the previous sections concentrated on alterations of the *where*-clause of the query. In the following examples, we outline some additional desirable features.

Condition adding. Often the user wants to refine the output of a query by adding a new condition. For example, consider the query

```
select author.bplace, library.city,
author.citizenship
from library, author
where author.bplace != library.city
and
     library.lname = 'Moffit Library'
```

which requests authors of books of Moffit Library who were not born in the library's city. If it is followed by the request

```
and author.bplace = 'Athens'
```

the system should identify that the implied query is:

```
select author.bplace, library.city,
author.citizenship
from library, author
where author.bplace != library.city
and
   library.lname = 'Moffit Library' and
   author.bplace = 'Athens'
```

and set the focus accordingly. An alternative to this approach would be to state the follow-up query as

```
author.bplace = 'Athens'
```

i.e., without the keyword `and`. The system should then fail to identify a match, realize that this is another refinement predicate and alter the issued query in the same way as before.

Projection list expansion. There could be a case where the user wants to simply expand the projection list of a query. This can be accomplished by issuing a follow-up request consisting of the attribute to be added. For instance, given the previous completed query, if the follow-up request is

```
author.byear
```

the system should generate the new query

```
select author.bplace, library.city,
author.citizenship, author.byear
from library, author
```

```
where author.bplace != library.city
and
   library.lname = 'Moffit Library' and
   author.bplace = 'Athens'
```

Keyword-clause alteration. Given the last completed query above and facing a follow-up request such as

```
select author.name, library.lname
```

the system should modify the target list of the query and generate

```
select author.name, library.lname
from library, author
where author.bplace != library.city
and
   library.lname = 'Moffit Library' and
   author.bplace = 'Athens'
```

Condition dropping. There is no implicit way of achieving condition dropping given our focus approach. The solution we have come up with is to add the keyword drop to the framework to allow the user to drop conditions. For instance, if the follow-up request to the previous query is

```
drop author.bplace = 'Athens'
```

the system should identify the condition dropping request and generate the query

```
select author.name, library.lname
from library, author
where author.bplace != library.city
and
     library.lname = 'Moffit Library'
```

4. Focus management algorithm

Based on the above concepts, we have developed and implemented an algorithm to manage the focus of the system and the operations on it throughout a user's exploratory session. Given an incomplete query, the parse tree under the current focus is searched exhaustively in a depth-first manner, until all potential sub-focuses are identified (identification or interpretation). These are all subtrees that could be replaced by the tree of the incomplete query to generate a valid complete query. If more than one possible sub-focus exists, the system exploits some heuristics in order to reach a decision regarding the most promising of them. This new sub-focus becomes the system's current focus. In case no sub-focus can be found, the system searches the rest of the parse tree that the current focus resides in, in order to identify a different focus. This new search of the parse tree comprises ascending the path that contains the current focus (expansion), as well as searching the siblings of the various nodes encountered (relocation). The ordering in which nodes are checked provides space for several variations of the basic algorithm. These variations will be presented in Section 4.2. If this fails as well, then the system searches for a focus in earlier parse trees (relocation).

The basic algorithm for focus management is presented more formally in Algorithm 1 in the form of a routine, named *handleFocus*, which accepts as its unique argument the incomplete query's parse tree (Q). Details are analyzed in subsequent subsections. Some symbols used in the description are the following:

$\mathcal{F}$: The system's current focus.

$U(x)$: The set of nodes in the subtree rooted at x.

$\propto$: The analogy predicate, which denotes that two parse subtrees can replace each other (Section 4.1).

best(S,Q) : Given a set of possible focuses (S), this routine identifies the most suitable to be matched to the incomplete query Q (Section 4.3).

ascend(F) : A routine that ascends the parse tree from the position of the current focus upwards, trying to find a new focus in other branches of the parse tree as suited (Section 4.2).

extractNewFocus(Q) : When no focus can be identified in the current parse tree, this routine tries to find a focus in the parse trees of previously issued queries (Section 4.4).

Algorithm 1 handleFocus(Q): The focus management routine

```
if $\mathscr{F} \propto Q$ then
   return;
else
   $U_f$:=$\{u : u \in U(\mathscr{F}), u \propto Q\}$;
   if $U_f \neq \emptyset$ then
      $\mathscr{F}$:= best($U_f, Q$);
   else
      $\mathscr{F}$:= ascend($Q$);
      if $\mathscr{F}$ = null then
         $\mathscr{F}$:= extractNewFocus($Q$);
      end if
   end if
end if
```

4.1. Syntactic analogy

In this section, we present the definition of parse-subtree analogy (predicate $\propto$), i.e., on the interchangeability of parse subtrees within larger parse trees. Based on the examples of Section 3.4, two parse subtrees are considered syntactically analogous if one of the following holds:

- The root nodes are of the same type and correspond to keyword clauses.
- The root nodes are of the same type and are terminal nodes.
- One of the root nodes corresponds to an empty syntactic tree and can be expanded to one that is syntactically analogous to the other root node.
- The root nodes are non-terminal nodes and cannot be expanded to other nodes, they have the same number of children, and their children are syntactically analogous.

Examples of pairs of analogous trees are given in Fig. 6.

4.2. Variations of expansion/relocation

Ascending the parse tree for focus expansion/relocation comprises the following steps:

Parent-check: Moving on the path from the current focus to the root of the parse tree, testing analogy with the subtrees rooted at the nodes on that path.

Sibling-check: Testing analogy with the subtrees rooted at the siblings of the nodes in the focus-to-root path.

Algorithm 2 ascend(Q): Ascend the parse tree in a ((*parent-check*) (*sibling-check*)*)$^+$ fashion

```
while parent($\mathscr{F}$) $\neq$ null do
   $x$:=$\mathscr{F}$
   $\mathscr{F}$:=par($\mathscr{F}$);
   if $\mathscr{F} \propto Q$ then
      return $\mathscr{F}$;
   else
      $U_f$:=$\{u : u \in U(\mathscr{F}), u \propto Q, u \neq x\}$;
      if $U_f \neq \emptyset$ then
         $\mathscr{F}$:=best($U_f, Q$);
         return $\mathscr{F}$;
      end if
   end if
end while
return null;
```

Different orders in which the execution of these two steps is interleaved lead to four basic variations of the ascending algorithm. Their names and their corresponding regular expressions are as follows:

Ancestors	((*parent-check*)(*sibling-check*)*)$^+$
Ancestors (queue)	(*parent-check*)$^+$(*sibling-check*)* (queue)
Ancestors (stack)	(*parent-check*)$^+$(*sibling-check*)* (stack)
Trees	((*sibling-check*)$^+$(*parent-check*))$^+$

Both the second and third variation test all of the siblings after all the nodes on the focus-to-root path have been tested, but differ in the data structure in which the siblings of the nodes tested

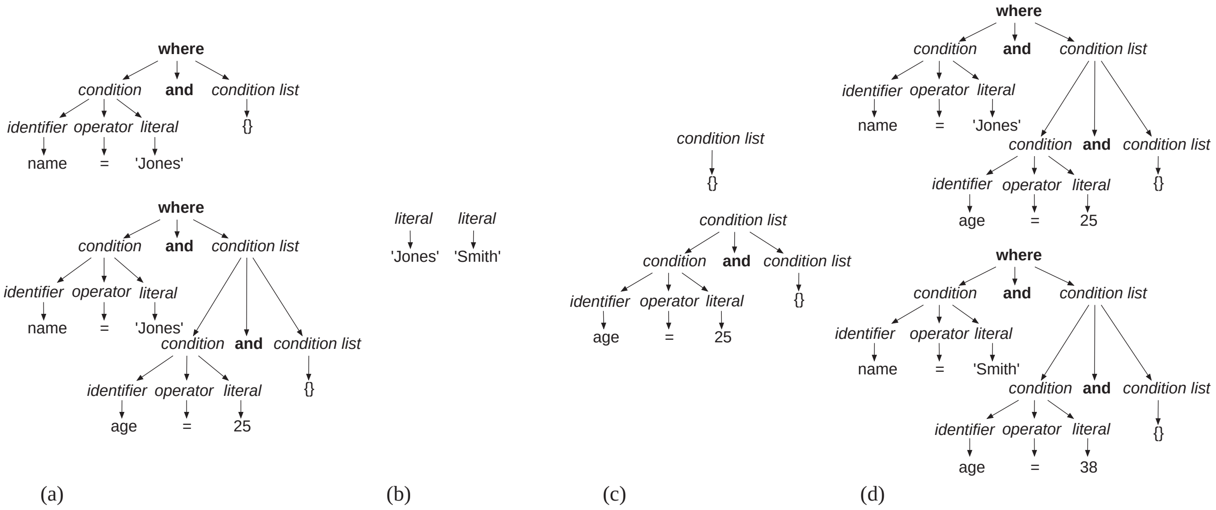


Fig. 6. Different cases of syntactically analogous parse trees. (a) Analogous keyword clauses. (b) Same-type terminal nodes. (c) Branch expansion. (d) Non-expandable non-terminal nodes.

while ascending are placed: they may be stored in a *queue*, thus giving precedence to the nodes appearing at the lower levels of the parse tree, or they may be stored in a *stack*, thus giving precedence to the nodes appearing at the higher levels. All four versions of the ascending algorithm are presented textually in Algorithms 2–5 and schematically in Fig. 7.[4]

As a basis case, we also consider a fifth algorithm where each new incomplete query initiates a search right at the root of the previous complete query's parse tree. This version disregards the notion of focus altogether. It is studied only for comparison purposes and completeness.

4.3. Disambiguation

Ambiguity is clearly a major problem that such an automatic query completion algorithm faces. A system that supports conversational querying must try to disambiguate a query. Otherwise, the system would not be helpful to the user; for example, if the user had to go through several possible completed queries to identify the correct one, then he or she would fall back to an interrupted exploration model, which is what we are trying to avoid.

There are two basic forms of ambiguity that may appear:

Algorithm 3 ascend(Q): Ascend the parse tree in a $((\textit{sibling-check})^{+}\ (\textit{parent-check}))^{+}$ fashion

```
while $parent(\mathscr{F}) \neq$ null do
  $x := \mathscr{F}$;
  $\mathscr{F} :=$ parent($\mathscr{F}$);
  $U_f := \{u : u \in U(\mathscr{F}), u \propto Q, u \neq x\}$;
  if $U_f \neq \emptyset$ then
    $\mathscr{F} :=$ best($U_f, Q$);
    return $\mathscr{F}$;
  else
    if $\mathscr{F} \propto Q$ then
      return $\mathscr{F}$;
    end if
  end if
end while
return null;
```

Algorithm 4 ascend(Q): Ascend the parse tree in a $(\textit{parent-check})^{+}\ (\textit{sibling-check})^{*}$ (queue) fashion

```
$q :=$ new Queue;
```

[4] The only new function that appears in the algorithms is *parent*, which is nothing more than a routine that returns the parent of any given node of the parse tree. It returns *null* if the node is the root.

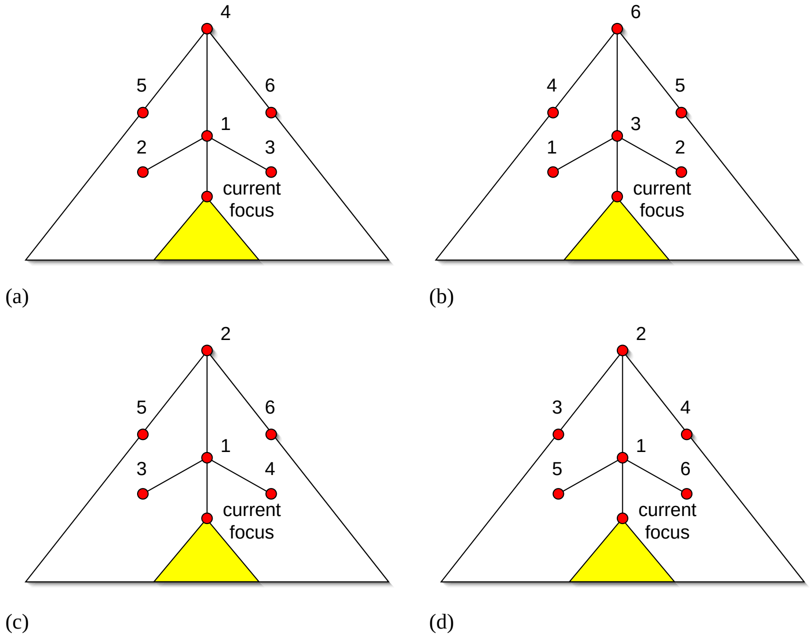


Fig. 7. The four basic variations of trying other nodes in the parse tree as possible focuses. The numbers indicate the time points at which the various nodes are tested. (a) ((*parent-check*) (*sibling-check*)*)$^+$, (b) ((*sibling-check*)$^+$(*parent-check*))$^+$, (c) (*parent-check*)$^+$(*sibling-check*)* (queue), (d) (*parent-check*)$^+$ (*sibling-check*)* (stack).

```
while $parent(\mathscr{F}) \neq$ null do
   $x := \mathscr{F}$;
   $\mathscr{F} := \mathrm{par}(\mathscr{F})$;
   if $\mathscr{F} \propto Q$ then
      return $\mathscr{F}$;
   else
      $U_f := \{u : u \in U(\mathscr{F}), u \neq x\}$;
      store $U_f$ in $q$;
   end if
end while
while $q$ not empty do
   $x :=$ remove from $q$;
   $U_x^q := \{u : u \in U(x), u \propto Q\}$;
   if $U_x^q \neq \emptyset$ then
      $\mathscr{F} := \mathrm{best}(U_x^q, Q)$;
      return $\mathscr{F}$;
   end if
end while
return null;
```

Population ambiguity: This stems from the fact that more than one possible completion for a query may exist. For example, if the incomplete query is just an arithmetic value there could be several such values appearing in a previously issued complete query. Which one is meant by the user?

Algorithm 5 ascend(Q): Ascend the parse tree in a (*parent-check*)$^+$ (*sibling-check*)* (stack) fashion

```
$s :=$ new Stack;
while $par(\mathscr{F}) \neq$ null do
   $x := \mathscr{F}$;
   $\mathscr{F} := \mathrm{par}(\mathscr{F})$;
   if $\mathscr{F} \propto Q$ then
      return $\mathscr{F}$;
   else
      $U_f := \{u : u \in U(\mathscr{F}), u \neq x\}$;
      push $U_f$ in $s$;
   end if
end while
while $s$ not empty do
   $x :=$ pop from $s$;
```

```
    $U^s_x$:={$u : u \in U(x), u \propto Q$};
    if $U^s_x \neq \emptyset$ then
        $\mathscr{F}$:=best($U^s_x$, $Q$);
        return $\mathscr{F}$;
    end if
end while
```

Chronological ambiguity: This appears after at least one incomplete query has been completed and sent to the database. If a subsequent incomplete query is issued, which query should be modified: the last completed query or the last query issued in complete form?

We treat the two types of ambiguity in separate subsections.

4.3.1. Population ambiguity

To handle population ambiguity, we have developed a mechanism that uses various criteria to identify the most promising choice among many alternative completions. This mechanism is implemented in routine *best* and its performance can dramatically affect the quality of the system's solutions. In particular, we have identified several *quality factors* against which each possible solution is *scored*. The solution that has the greatest score is returned as the winner. The quality factors identified thus far are the following:

1. Distance from previous values. The closer a value is to a previously appearing value, the higher the probability that it references that value. For instance, if two possible values appear in the previous completed query, 0.4 and 1000, and the incomplete query is the value 0.8, then it is highly likely that the incomplete query is referring to 0.4.

2. Chronological information. The older a possible solution is (i.e., the earlier the construct has been introduced to a query), the less likely that it is the one referred to by the user. In other words, if a value has remained unchanged for many queries it is likely that it is a constant for the current exploration by the user and is not to be changed.

Algorithm 6 best(S, Q): Possible solution selection based on quality factors

```
for all $p \in S$ do
    score[$p$]:=0;
    for all *quality factor* do
        score[$p$] := score[$p$] + score($p, Q$, *quality-factor*);
    end for
end for
return $p$ : $\max_p${score[$p$]};
```

3. Histogram information. The system can create a usage histogram for referenced values. The semantic category of values that appears to have the most references is most likely to be referenced in the future as well.

Clearly, additional quality factors that may be appropriate can be used as well. In the current implementation of the algorithms, the distance from previous values is the only one used.

Algorithm 6 gives an algorithmic description of the overall approach. Routine *best* accepts as arguments the collection of possible solutions (S) as well as the incomplete query's parse tree (Q). It then assigns a score for each quality factor, accumulating the scores for each possible solution in a linear array (*score[ ]*). The index of the array with the maximal score corresponds to the most promising solution.

A key issue that arises in the definition of several quality factors is what the exact candidate set of solutions is. For value difference, for example, should one consider all numbers the same? For histogram information, how should the semantic categories be defined? Although the algorithms are domain independent (i.e., they do not depend on the database schema to work properly), their effectiveness increases if knowledge about the values appearing in the database is used. We have therefore enhanced the algorithms with the ability to take into account semantic enhancements of the typing system of the database. In particular, the algorithms assume the existence of trees of refinements (called *semantic trees*) of the primitive data types used. An example of such refinements is depicted in Fig. 8. Each value is characterized by a general data type (e.g., integer) as well as a more refined semantic type (e.g., publication year). These trees are now templates for further context that the matching algorithms can take advantage of and limit the choices of value matching. This is

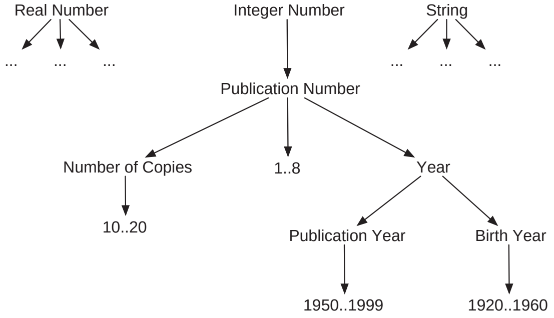


Fig. 8. Semantic enhancements of the basic data types.

better understood through an example. Consider the following complete query

```
select book.author,
in_stock.lib_name, customer.cname
from customer, book, in_stock
where customer.byear < = 1946 and
    book.pyear != 1993 and
    in_stock.quantity = 2 and
    book.title = 'Multikey Retrieval'
```

and the two follow-up requests `4` and `1947`. With the semantic trees, both can be interpreted unambiguously. The first one refers to a quantity value, while the second refers to a birth year. Furthermore, consider a subsequent follow-up request `1955`. Although this value belongs to two semantic categories, the request can still be interpreted unambiguously, because a context has already been established under 'birth year' from the previous query. Since the value is given without further annotations, it should be clear that the intended meaning is to replace the value `1947`. Given above is an instance of the general algorithm we have developed for handling context in semantic trees much like the way it is done in parse trees.

4.3.2. Chronological disambiguation

In Section 3.4.2 we introduced the concept of *chronological ambiguity*. In preliminary experimentation with the system, we found that this issue was completely a matter of personal perception, with no clear answer on what humans perceive as proper. People seem to be divided into three categories: those who give precedence to changing the *last issued* query, those who give precedence to changing the *last completed* query, and those who are aware of the ambiguity and accept both as plausible.

Fortunately, our experiments showed that humans are at least consistent and do not switch categories depending on the situation. Therefore, we have developed an algorithm that takes advantage of this consistency, learns the inclination of the user and interprets queries accordingly. In particular, initially the algorithm assumes that both interpretations are valid and always returns to the user asking for what he/she means by the incomplete query. After a few iterations, if the user shows a consistent preference (as is typically the case), the system 'locks' onto it.

The above is valid as long as the number of follow-up requests and the time since the initialization request was given remain short. If many queries have been given or a long time has passed (where 'many' and 'long' are defined through certain thresholds we have experimented with), then the user more or less 'forgets' what the initialization request was and concentrates on the last completed query. The overall chronological disambiguation algorithm is presented in Algorithm 7.

4.4. Searching in earlier parse trees

In the basic focus management routine (Algorithm 1) there is the possibility that no focus can be extracted from the parse tree containing the current focus. In this case, a new routine is called which searches through the system's repository of previously issued queries, trying to identify a focus within each of them. The algorithm is quite simple. It searches chronologically through the complete queries' parse trees, and on the first parse tree where it can identify a focus, sets this portion of the parse tree as the system's current focus. The algorithm is presented in Algorithm 8 as a routine, named extractNewFocus, which takes as its single argument the incomplete query's parse tree. The only point worth mentioning about the routine is the existence of set $\mathscr{R}$, which denotes the system's repository of previously issued and completed queries.

Algorithm 7 disambiguate(): The basic chronological disambiguation algorithm

```
if the user is characterized then
    no change
else if the time that has passed from the last issued query or the number of follow-up request are beyond the limits then
    set the user's preference on the last completed query
else if we are beyond the time limits in which the user should be characterized then
    set the user's preference on his current inclination
else
    present both alternatives and ask the user to evaluate them
end if
```

Algorithm 8 extractNewFocus(Q): New focus extraction

```
for all $s \in \mathscr{R}$ do
    $U_s := \{u : u \in U(s), u \propto Q\}$;
    if $U_s \neq \emptyset$ then
        $\mathscr{F} := \mathrm{best}(U_s, Q)$;
        return $\mathscr{F}$;
    end if
end for
return null;
```

5. Implementation and experimentation

We have implemented all versions of the algorithm and have experimented with them using human subjects. In the following sections, we describe the details of the experimental setup and the results.

5.1. Restrictions

Although our overall approach is valid for any textual language, at this point the implementation of the algorithms is restricted as follows:

- The query language is a core subset of SQL, in particular conjunctive queries. Expanding the system to the full SQL language should present no major problems.
- The implemented parser generates only unambiguous parse trees. This is in accordance with the representation specifications presented in the beginning of Section 3.
- The system accepts only one change at a time, i.e., every follow-up request consists of only one alteration to the original query.

5.2. Measures of effectiveness

To measure the effectiveness of our algorithms, we built upon methodologies that have been used in query language comparisons [24] and general information retrieval [25].

Let U be the set of all possible completions a user implies by issuing a follow-up request to an initializing query. Let S be the possible completions the system identifies. From these two sets stem two important metrics for the effectiveness of information retrieval systems, namely *precision* and *recall* [25]. *Recall* is defined as the proportion of relevant answers that are retrieved, i.e., $|U \cap S|/|U|$. *Precision* is defined as the proportion of retrieved answers that are relevant, i.e., $|U \cap S|/|S|$ (see also Fig. 9(a)). An ideal system has recall and precision values of 100%.

For a system of Conversational Querying, precision and recall over all possible interpretations of queries a user may pose may not be all that interesting. In such a system there are two critical concerns:

- *absolute soundness*: if the system determines that an incomplete query is uniquely interpretable,

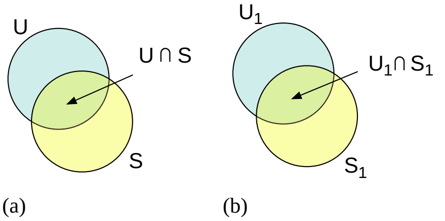


Fig. 9. Performance metrics. (a) Precision ($|U \cap S|/|U|$) and recall ($|U \cap S|/|S|$). (b) System-singular ($|S_1 \cap U_1|/|S_1|$) and user-singular ($|S_1 \cap U_1|/|U_1|$) queries.

then this is indeed the case for the user as well and the interpretation is what the user has in mind, i.e., precision and recall are 100% in all such cases;

- close to *absolute completeness*: if the user poses an incomplete query that he/she thinks is uniquely interpretable, the system does the same in the great majority of cases as well.

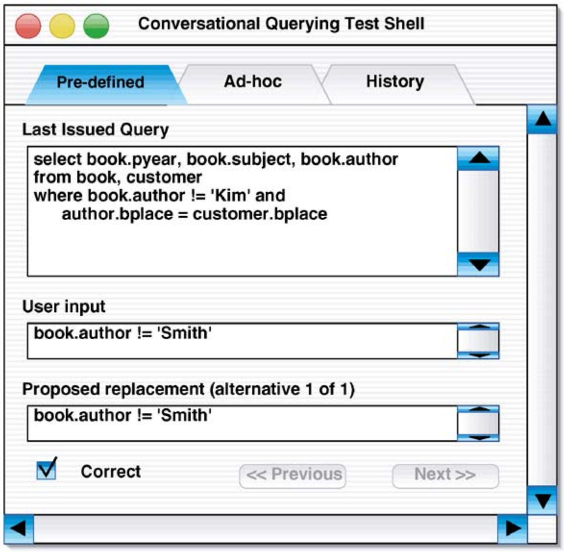
Based on the above, we have introduced two versions of the notion of *singularity*. Let U_1 be the set of queries the user thinks should be uniquely interpreted (*user-singular queries*) and S_1 the set of queries the system interprets uniquely (*system-singular queries*). The *system's singularity* is defined as $|S_1 \cap U_1|/|S_1|$, i.e., it measures the fraction of times the user agrees with the system's interpretation that a query is uniquely interpretable. Likewise, the *user's singularity* is defined as $|S_1 \cap U_1|/|U_1|$, i.e., it measures the fraction of times the system matches the user's ability to uniquely interpret a query. For a system to be successful, the following should hold:

- *absolute soundness*: system's singularity should be exactly equal to 1, and so should be precision and recall for queries in S_1;
- *close to absolute completeness*: user's singularity should be as close to 1 as possible, and so should be precision and recall for queries in U_1.

The assumption here is that the system interpreting an ambiguous query uniquely and sending it off for execution is harmful, whereas the system occasionally asking for help when things are clear for the user is acceptable (see system architecture in Fig. 4) .

5.3. Experimental methodology

The participants in our experimentation framework consisted of eleven persons of the Department of Informatics and Telecommunications at the University of Athens. Of these participants, one had a Ph.D. in Computer Science, two were Ph.D. candidates and the remaining eight were undergraduate Computer Science majors with various degrees of experience in SQL.



Last Issued Query

```
select book.pyear, book.subject, book.author
from book, customer
where book.author != 'Kim' and
     author.bplace = customer.bplace
```

User input

book.author != 'Smith'

Proposed replacement (alternative 1 of 1)

book.author != 'Smith'

☑ Correct

Fig. 10. The Graphical User Interface used for our experimentation.

The participants were presented with a graphical user interface as shown in Fig. 10.[5] In the topmost part of the window, one could see the last complete given SQL query. In this particular example, that query selects customers, book subjects and author names of books written by authors other than 'Kim' who share the same birthplace as the customers that borrowed them. In the middle part of the window, one could see the user's input, i.e., the modification to the original query the user wants to perform. In the bottom part of the window, one could see the system's response, that is, its perception of what the user means. More specifically, the system can identify only one possible modification to the original query (the 'Alternative 1 of 1' part of the system's response) and waits for the user to acknowledge whether the proposed modification is the intended one or not (the 'Correct' check-box next to the system's response). This way, the success or failure of the system could be measured. We should also note that, since our interest was only at the query

[5] Note that this is an interface for experimentation and not necessarily for use in actual conversations with a DBMS.

perception level, there was no real DBMS under this interface during experimentation.

We investigated sequences of incomplete queries that corresponded to the following alterations in the focus: *continuous narrowing*, *continuous expansion*, *narrowing followed by expansion*, *expansion followed by narrowing*, *relocation*, and ad hoc.[6] For the first five, the sequences were synthetic ones that we had pre-defined as follows: complete queries were generated randomly over a fixed database schema, each followed by a series of random incomplete queries. Of these, we elaborate briefly on the last one only, as the rest are self-explanatory. *Relocation* experiments were conducted in three forms. In the first one, we attempted to alter whole qualifications appearing in the query. In the second one, we provided follow-up requests consisting of changes in values appearing in the initialization request. In the third one, we experimented with the algorithms in bigger examples consisting of more qualifications and follow-up requests for each complete query. The objective of these experiments was to test the semantic and chronological disambiguation procedures.

For the above experiments, the participants were to interact with the system and decide whether the prespecified incomplete queries were accurate or ambiguous, according to their opinion. The sequence of operations they had to go through for each query was the following:

- Submit the (complete or incomplete) query
- Call the matching algorithm
- Evaluate the returned interpretations with regard to whether or not they were considered correct (important for precision).
- Identify further interpretations if the participant could think of more (important for recall).

In ad hoc experimentation, we asked the participants to use the system in any way they desired, i.e., issue any sequence of queries they wanted. Although not having any actual interaction with data could have degraded the quality of the queries, the results seemed not to be affected.

A final issue to touch upon is the alternative of performing 'Wizzard of Oz' experiments instead of the experimental methodology we decided to follow. Such tests, although they might be helpful in providing a better understanding of what is expected from the system by the user, are not immediately applicable due to the real-time nature of the application we want to test. Interaction, when through a SQL shell, cannot be emulated by a human given the millisecond response time expected by a database system.

For all types of experiments, none of the participants knew how the internal matching algorithms worked. They were only asked to grade the system's behavior. During a user session, statistics were collected based on which the aforementioned metrics (precision, recall, singularity) were calculated.

5.4. Results

5.4.1. Overall performance

In Fig. 11, we present the overall performance of the algorithms in terms of precision and recall. The figure depicts each algorithm's average performance for these measurements over all users and all queries. All algorithms behave quite well. However, we see that, on the average, the *Trees* algorithm has distinctly the best performance with regard to the measures of effectiveness. The intuition behind this result is as follows: people do not seem to take sudden leaps of the conversational focus, which would result in a relocation of

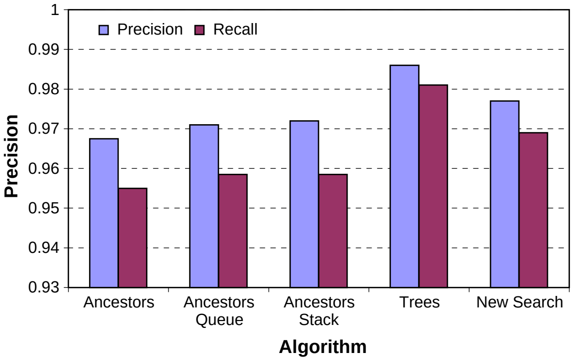


Fig. 11. Overall performance of the algorithms in terms of precision and recall.

[6] For a summary of the queries used, visit http://www.di.uoa.gr/~stratis/queries.

the focus to a distant area of the syntactic tree. The *Trees* algorithm first looks at siblings of the current focus, that is, nodes at the same level in the tree. These seem conceptually closer to the current focus than its parent which is what all other algorithms look at first.

It is interesting to spend some time on the discrepancies between user expectations and the completed queries the system proposed. For the most part, there was no consistent discrepancy other than what can be attributed to the inherent behavior of the algorithms. For instance, in relocation experiments, the users were consistent in noting the *Trees* algorithm as the best one and ranking the other algorithms as worse. The cases where the *Trees* algorithm was wrong, in terms of its precision and recall metrics, were the ones in which multiple possible completions of the same semantic category were available. Consider for example a query containing two year dates classified as *publication years* where the focus has been put on one of those years; if a follow-up request was made with an additional year date (and the system was *not* being used in an ad hoc fashion) some users thought it was equally possible that the follow-up request referred to either of the two years. Other than cases like the one just described, there was no repeatable discrepancy between the users' expectations and the system's decisions.

5.4.2. Time and space costs

In this section, we present the time and space cost of the algorithms. The time cost is the actual time the algorithm needed to interpret a follow-up request, while the space cost is the number of steps (i.e., movements within the parse tree) the algorithm needed for the interpretation. The number of steps is an approximation of space since it corresponds to the length of the traversal the algorithm will make in order to reach a new focus, during which analogous information is maintained. Both of these measurements should remain low. As far as time measurements are concerned, they appear to remain similar for all algorithms, while space shows some difference. As expected, the new search algorithm almost always needed more steps to reach a new solution, something that stems from the fact that each time it searches the whole parse tree. For the remaining algorithms, their behavior is comparable, with the Ancestors-Stack and Ancestors-Queue algorithms almost constantly taking fewer steps. This behavior is depicted in Figs. 12(a) and (b).

Since the algorithms' performance in terms of time and space appears to be relatively invariant and the *Trees* algorithm shows superior effectiveness, we concentrate on that algorithm for the rest of our result presentation.

5.4.3. Performance in singular queries

In Table 1, we present the performance of the *Trees* algorithm in the cases of system-singular and user-singular queries (see Section 5.2 for the definition of singularity). As mentioned earlier, the goal is for a system to have values equal to 1 for system singularity and very close to 1 for user singularity, and similarly for precision and recall.

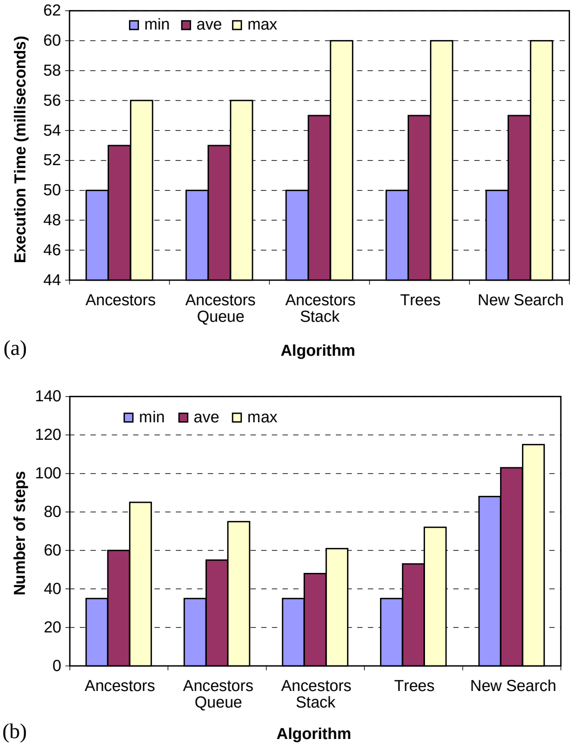


Fig. 12. Time and space costs of the algorithms. (a) Time cost of the algorithms (milliseconds). (b) Space cost (number of steps) of the algorithms.

Table 1
System's performance in singular queries

	Singularity	Precision	Recall
System	0.996	0.987	0.987
User	0.987	0.984	0.978

We see that in both cases the results are very close to that goal, essentially proving the effectiveness of our approach and, more generally, the ability to develop sound systems that support Conversational Querying.

5.4.4. *Effect of focus alteration*

The *Trees* algorithm's behavior for singular queries in all types of query sequences is tested in Figs. 13(a) and (b). In 'oscillation' experiments (narrowing, expansion, narrowing-expansion, and expansion-narrowing), the algorithm yielded considerably high results.

An interesting result is that ad hoc queries almost always yield better results than the other types of queries. The reason is that the ad hoc queries the users issued were a combination of focus narrowing and relocation queries, with a bigger concentration of relocation ones. Since the performance is high in both of these categories, the overall performance in their combination remains high, and the overall statistics actually elevate it to better scores.

5.4.5. *Non-user-singular solutions*

There were some cases where the issued query was considered ambiguous by the participant. These, however, were few, which is why we present only one summarizing table. Note that in these cases, system singularity had to remain as low as possible. That would mean that the system would not falsely regard an ambiguous input as one that can be interpreted uniquely. Precision and recall, on the other hand, had to remain high in the sense that system and user perceive this ambiguity in the same way. In most cases, our system succeeded in identifying the ambiguity, yielding high measurements in precision and recall as well. The results are shown in Table 2.

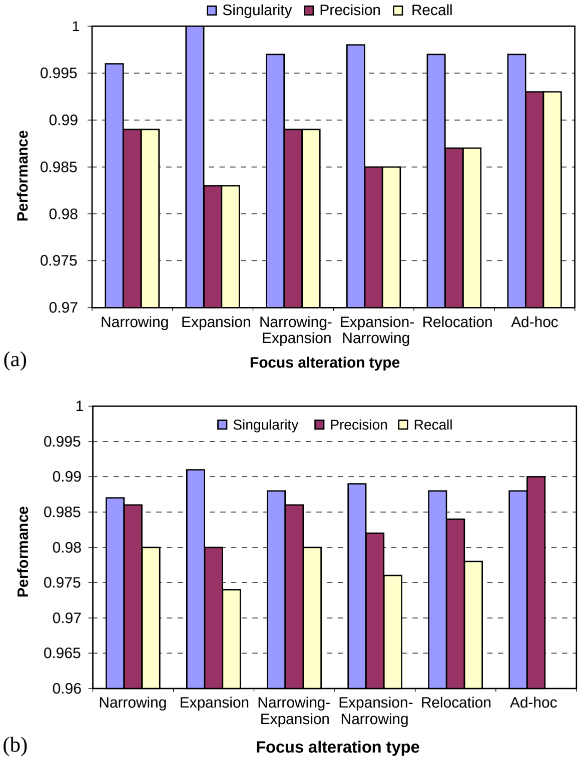


Fig. 13. Performance for system- and user-singular queries. (a) System performance by focus alteration type, regarding system-singular queries. (b) System performance by focus alteration type, regarding user-singular queries.

Table 2
System's performance in ambiguous queries

	Singularity	Precision	Recall
System	0.009	0.997	0.995

5.4.6. *User characterization*

The focus of this section is on the increase in effectiveness our algorithm achieves by trying to characterize the user as one giving precedence to the last issued or the last completed query. Figs. 14(a) and (b) depict the values of singularity, precision, and recall before and after the user is characterized based on his/her behavior on three follow-up requests and interpretations. We see that the system's performance increases substantially after user characterization. Note that we

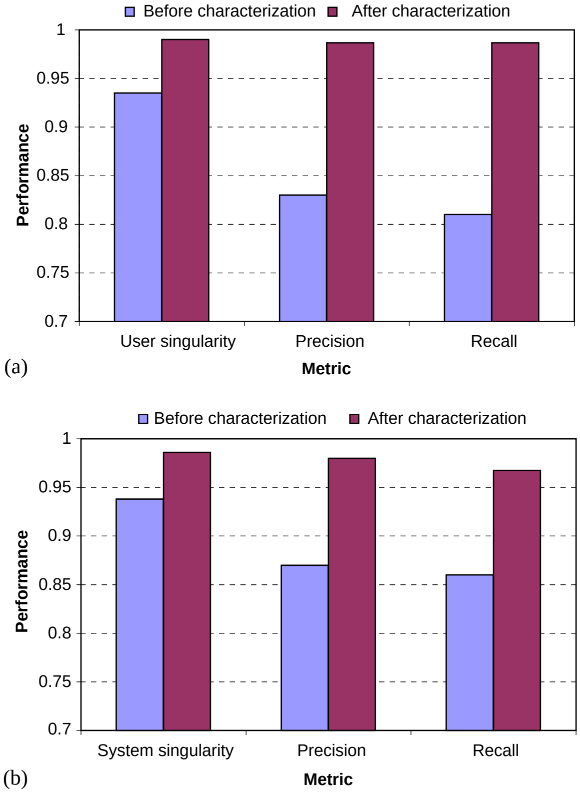


Fig. 14. Performance gains from user characterization. (a) Performance gain earned by characterizing the user for system-singular queries. (b) Performance gain earned by characterizing the user for user-singular queries.

experimented with 'locking' a user's preference after several different numbers of queries and 3 was the best trade-off point between result quality and time spent before characterization.

5.5. Discussion

There are several observations from the overall experimentation.

1. *Once agreement is reached between user and algorithm, performance is always high.* In most cases, once the user and the algorithm agreed on the singularity of an interpretation, the system almost always provided the right one.
2. *Some algorithms are better than others.* In relocation experiments, some algorithms yielded constantly better results than the rest. The *Trees* algorithm is the one that had the overall better performance with efficiency measures remaining constantly over 96–97%. A close competitor is the *Ancestors-Stack* algorithm. The latter had better spatial performance, although the former was in close proximity in that sector. Time seems to play no significant role.
3. *Ad hoc queries tend to be relocation queries.* In ad hoc experimentation, the users initially posed a query, and then started fine-tuning, it changing the different constants that appeared, thus relocating the query focus.
4. *Focus is the way to go.* The *new search* algorithm was used as a basis for comparison of our strategy. In some cases, this algorithm has better performance than some of the other ones, but in most cases it is worse. Singularity is constantly lower for this algorithm when compared with the focus-based ones.
5. *People seem to be divided in categories.* Each participant showed a preference that was constant throughout the experiment. More specifically, of the 11 participants, 4 showed a constant preference towards the last completed query, 6 towards the last issued query, while one seemed to give equal preference to both alternatives.

An important issue related to what we have presented stems from the potential generalization of our approach from conjunctive queries, like the ones we have experimented with, to full OLAP-style queries, which are not going to be as small as the ones we have presented (indeed, they could span multiple pages). In such a scenario, the concept of identifying a focus through the use of a parse tree may not be so straightforward. We expect, however, that the queries in such an environment will be formulated through some user interface that is more elaborate than the one we used in our experimentation. Under these circumstances, additional information becomes available, such as even 'cleaner' focus identification through the use of parts of the interface as hints to the system. For instance, in the visual examples presented in Figs. 2(a) and (b) there is a clear hint of which portion of the query is to be changed (the portion corresponding to the area in the rectangle.) Identifying focus mappings in complex OLAP scenarios and more elaborate user interfaces present challenging directions for future work.

6. Conclusions and future work

We have presented an effective mechanism that can be used as a starting point for the implementation of Conversational Querying in contemporary databases. We have mapped the problem of automatic completion of incomplete queries to the problem of finding corresponding structures in a parse tree. The key concept was that for any given parse tree, a specific subtree has to be identified that will act as the focus of any interaction, i.e., the context of the conversation between the user and the system. We have proposed and evaluated several algorithms that can be used for the basic focus operations, namely interpretation, expansion and relocation. Given the ever increasing number of database users that need to interact with a DBMS's front-end in an exploratory way, we believe that the work we presented provides a foundation for the development of a more flexible user-interface that would be helpful to both naive as well as knowledgeable users.

The future work we plan to undertake mostly stems from various aspects of the mechanism we have presented. The first step in a more concrete evaluation of our strategy is the conducting of a more comprehensive experiment involving a large number of human subjects, ad hoc interaction between the user and the system, realistic queries and large schemes. A second step is the further exploitation of domain-specific and semantic knowledge. The way to handle the underlying database schema in order to extract all the appropriate semantic information that can be used to exploit this domain-specific knowledge is another issue. Chronological ambiguity is also an issue. People's opinions seem to be divided on the subject, although the limits between the two categories are vague. Further exploration on this matter has to be carried out.

Finally, the most ambitious part focuses on incorporating our strategy into a visual query system. While some systems are already developed, our strategy, if successful, will provide additional functionality and processing speed of the visual queries. It will not be necessary for each new visual query to be created from scratch from the query graph, since one can act on alterations of the textual representation of the visual query, which, as was shown in Section 5.4.2, is a really fast process. Moreover, additional information (such as spatial information) can be used in order to improve the system's performance.

Acknowledgements

We are indebted to the anonymous reviewers for a large number of useful comments in the earlier versions of this paper. We would also like to thank Dr. Anya Sotiropoulou, Dr. Costas Vassilakis, George Boukeas, Loukas Dimopoulos, Tina Galoussi, George-Dimitrios Kapos, Fragkiskos Pentaris, Vassilis Stoumpos, Orestis Telelis, Vasso Toutouzi, and Kyriakos Zervoudakis for their valuable contribution in conducting the experiments. Without them, the presented results would be substantially poorer in quality.

References

[1] Y.E. Ioannidis, M. Livny, S. Gupta, N. Ponnekanti, ZOO: a desktop experiment management environment, in: T.M. Vijayaraman, A.P. Buchmann, C. Mohan, N.L. Sarda (Eds.), Proceedings of 22th International Conference on Very Large Data Bases, Morgan Kaufmann, Los Altos, CA, 1996, pp. 274–285.

[2] F.A. Barros, Semi-Automatic Anaphora Resolution in Portable Natural Language Interfaces, Lecture Notes in Computer Science, vol. 1159, 1996.

[3] S. Carberry, A pragmatics-based approach to ellipsis resolution, Am. J. Comput. Linguistics (1989).

[4] J.G. Carbonell, R.D. Brown, Anaphora resolution: a multi-strategy approach, in: Proceedings of the International Conference of Computational Linguistics, Budapest, Hungary, 1988.

[5] S. Lorenz, Presupposition, anaphora, and reasoning about change, in: Bernd Neumann (Ed.), Proceedings of the 10th European Conference on Artificial Intelligence, Vienna, Austria, Wiley, New York, 1992, pp. 533–537.

[6] R.M. Weischedel, N.K. Sondheimer, An improved heuristic for ellipsis processing, in: ACL Proceedings, 20th Annual Meeting, 1982, pp. 85–88.

[7] M.A.K. Halliday, R. Hasan, Cohesion in English, Longman, New York, 1976.

[8] G. Lakoff, M. Johnson, Metaphors We Live By, University of Chicago Press, Chicago, 1983.

[9] B. Grosz, A. Joshi, S. Weinstein, Centering: a framework for modeling the local coherence of discourse, Comput. Linguistic. 2 (21) (1995) 203–225.

[10] S.G. Pulman, Comparatives and Ellipsis, in: Proceedings of the Fifth European Meeting of the Association for Computational Linguistics, 1991, pp. 1–6.

[11] G. Huet, A unification algorithm for typed lambda calculus, J. Theoret. Comput. Sci. 1 (1) (1975) 27–57.

[12] R. Montague, Formal Philosophy: Selected Papers of Richard Montague, Yale University Press, New Haven, CT, 1974.

[13] L. de Sopena, Natural language grammars for an information system, in: Proceedings of the Annual ACM Conference on Research and Development in Information Retrieval, 1983, pp. 75–80.

[14] F.J. Damerau, Problems and some solutions in customization of natural language database front-ends, ACM Trans. Inform. Systems (TOIS) 3 (2) (1985) 165–184.

[15] D.G. Shin, Semantics modeling issues for processing natural language database queries, in: Proceedings of the 1990 ACM annual conference on Cooperation, 1990, pp. 8–14.

[16] Y.E. Ioannidis, Y. Lashkari, Incomplete path expressions and their disambiguation, in: SIGMOD Conference, 1994, pp. 138–149.

[17] E. Sciore, Query abbreviations in the entity-relationship model, Inform. Systems (1994).

[18] W.W. Chu, Q. Chen, M. Merzbacher, CoBase: a cooperative database system, Non-Standard Queries and Answers, 1994.

[19] G. Zhang, W.W. Chu, F. Meng, G. Kong, Query formulation from high-level concepts for relational databases, in: User Interfaces to Data Intensive Systems, Edinburgh, Scotland, 1999.

[20] W.W. Chu, G. Zhang, Associative query answering via query feature similarity, in: International Conference on Intelligent Information Systems, The Bahamas, 1997.

[21] G. Fouque, W.W. Chu, H. Yau, A case-based reasoning approach associative query answering, in: Proceedings of the Eighth International Symposium Methodologies for Intelligent Systems, Charlotte, NC, 1994.

[22] A.W. Biermann, C.I. Guinn, D.R. Hipp, R.W. Smith, Efficient collaborative discourse: a theory and its implementation, in: ARPA Human Language Technology Workshop, Princeton, NJ, March 1993.

[23] C. Guinn, A.W. Biermann, Conflict resolution in collaborative discourse, in: International Joint Conference on Artificial Intelligence Workshop: Computational Models of Conflict Management in Cooperative Problem Solving, Chambery, France, 1993.

[24] M. Jarke, Y. Vassiliou, A framework for choosing a database query language, Comput. Surve. 17 (3) (1985).

[25] G. Salton, Automatic Text Processing: The transformation, Analysis, and Retrieval of Information by Computer, Addison-Wesley Publishing, Reading, MA, 1989.