

An Enhanced Search Interface for Information Discovery from Digital Libraries

Georgia Koutrika^{1,*} and Alkis Simitsis^{2,**}

¹ University of Athens,
Department of Computer Science,
Athens, Greece
koutrika@di.uoa.gr

² National Technical University of Athens,
Department of Electrical and Computer Engineering,
Athens, Greece
asimi@dbnet.ece.ntua.gr

Abstract. Libraries, museums, and other organizations make their electronic contents available to a growing number of users on the Web. A large fraction of the information published is stored in structured or semi-structured form. However, most users have no specific knowledge of schemas or structured query languages for accessing information stored in (relational or XML) databases. Under these circumstances, the need for facilitating access to information stored in databases becomes increasingly more important. Précis queries are free-form queries that instead of simply locating and connecting values in tables, they also consider information around these values that may be related to them. Therefore, the answer to a précis query might also contain information found in other parts of the database. In this paper, we describe a précis query answering prototype system that generates personalized presentation of short factual information précis in response to keyword queries.

1 Introduction

Emergence of the World Wide Web has given the opportunity to libraries, museums, and other organizations to make their electronic contents available to a growing number of users on the Web. A large fraction of that information is stored in structured or semi-structured form. However, most users have no specific knowledge of schemas or (semi-)structured query languages for accessing information stored in (relational or XML) databases. Under these circumstances, the need for facilitating access to information stored in databases becomes increasingly more important.

Towards that direction, existing efforts have mainly focused on facilitating querying over structured data proposing either handling natural language queries [2, 14, 17] or free-form queries [1, 18]. However, end users want to achieve their goals

* This work is partially supported by the Information Society Technologies (IST) Program of the European Commission as part of the DELOS Network of Excellence on Digital Libraries (Contract G038-507618).

** This work is co-funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS.

with a minimum of cognitive load and a maximum of enjoyment [12]. In addition, they often have very vague information needs or know a few buzzwords. Therefore, the usefulness of keyword-based queries, especially compared to a natural language approach in the presence of complex queries, has been acknowledged [26].

Consider a digital collection of art works made available to people on the Web. A user browses the contents of this collection with the purpose of learning about “Michelangelo”. If this need is expressed as a free-form query, then existing keyword searching approaches focus on finding and possibly interconnecting entities that contain the query terms, thus they would return an answer as brief as “*Michelangelo: painter, sculptor*”. This answer conveys little information to the user and more importantly does not help or encourage him in searching or learning more about “Michelangelo”. On the other hand, a more complete answer containing, for instance, biographical data and information about this painter’s work would be more meaningful and useful instead. This could be in the form of the following précis:

“Michelangelo (March 6, 1475 - February 18, 1564) was born in Caprese, Tuscany, Italy. As a painter, Michelangelo's work includes Holy Family of the Tribune (1506), The Last Judgment (1541), The Martyrdom of St. Peter (1550). As a sculptor Michelangelo's work includes Pieta (1500), David (1504).”

A précis is often what one expects in order to satisfy an information need expressed as a question or as a starting point towards that direction. Based on the above, support of free-form queries over databases and generation of answers in the form of a précis comprises an advanced searching paradigm helping users to gain insight into the contents of a database. A précis may be incomplete in many ways; for example, the abovementioned précis of “Michelangelo” includes a non-exhaustive list of his works. Nevertheless, it provides sufficient information to help someone learn about Michelangelo and identify new keywords for further searching. For example, the user may decide to explicitly issue a new query about “David” or implicitly by following underlined topics (hyperlinks) to pages containing relevant information.

In the spirit of the above, recently, précis queries have been proposed [11]. These are free-form queries that instead of simply locating and connecting values in tables, they also consider information around these values that may be related to them. Therefore, the answer to a précis query might also contain information found in other parts of the database, e.g., frescos created by Michelangelo. This information needs to be “assembled” -in perhaps unforeseen ways- by joining tuples from multiple relations. Consequently, the answer to a précis query is a whole new database, a logical database subset, derived from the original database compared to flattened out results returned by other approaches. This subset is useful in many cases and provides to the user much greater insight into the original data.

The work that we describe in this paper focuses on design and implementation issues of a précis query answering prototype with the following characteristics:

- Support of a keyword-based search interface for accessing the contents of the underlying collection.
- Generation of a logical subset of the database that answers the query, which contains not only items directly related to the query selections but also items implicitly related to them in various ways.

- Personalization of the logical subset generated and hence the précis returned according to the needs and preferences of the user as a member of a group of users.
- Translation of the structured output of a précis query into a synthesis of results. The output is an English presentation of short factual information précis.

Outline. Section 2 discusses related work. Section 3 describes the general framework of précis queries. Section 4 presents the design and implementation of our prototype system, and Section 5 concludes our results with a prospect to the future.

2 Related Work

The need for free-form queries has been early recognized in the context of databases [18]. With the advent of the World Wide Web, the idea has been revisited. Several research efforts have emerged for keyword searching over relational [1, 3, 8, 13] and XML data [5, 6, 9]. Oracle 9i Text [19], Microsoft SQL Server [16] and IBM DB2 Text Information Extender [10] create full text indexes on text attributes of relations and then perform keyword queries.

Existing keyword searching approaches focus on finding and possibly interconnecting tuples in relations that contain the query terms. For example, the answer for “Michelangelo” would be in the form of relation-attribute pair, such as (Artist, Name). In many practical scenarios, this answer conveys little information about “Michelangelo”. A more complete answer containing, for instance, information about this artist's works would be more useful. In the spirit of the above, recently, précis queries have been proposed [11]. The answer to a précis query is a whole new database, a logical database subset, derived from the original database. Logical database subsets are useful in many cases. However, naïve users would rather prefer a friendly representation of the information contained in a logical subset, without necessarily understanding its relational character. In earlier work [11], the importance of such representation constructed based on information conveyed by the database graph, has been suggested. A method for generating an English presentation of the information contained in a logical subset as a synthesis of simple SPO sentences has been proposed [21]. The process resembles those involved in handling natural language queries over relational databases in that they both involve some amount of additional predefinitions for the meanings represented by relations, attributes and primary-to-foreign key joins. However, natural language query processing is more complex, since it has to handle ambiguities in natural language syntax and semantics whereas this approach uses well defined templates to rephrase relations and tuples.

The problem of facilitating the naïve user has been thoroughly discussed in the field of natural language processing (NLP). For the last couple of decades, several works are presented concerning NL Querying [26, 15], NL and Schema Design [23, 14, 4], NL and DB interfaces [17, 2], and Question Answering [25, 22]. Related literature on NL and databases, has focused on totally different issues such as the interpretation of users' phrasal questions to a database language, e.g., SQL, or to the automatic database design, e.g., with the usage of ontologies [24]. There exist some recent efforts that use phrasal patterns or question templates to facilitate the answering procedure [17, 22]. Moreover, these works produce pre-specified answers,

where only the values in the patterns change. This is in contrast to *précis* queries, which construct logical subsets on demand and use templates and constructs of sentences defined on the constructs of the database graph, thus generating dynamic answers. This characteristic of *précis* queries also enables template multi-utilization.

In this paper, we built upon the ideas of [11, 20, 21] and we describe the design and implementation of a system that supports *précis* queries for different user groups.

3 The Précis Query Framework

The purpose of this section is to provide background information on *précis* queries.

Preliminaries. We consider the *database schema graph* $\mathbf{G}(\mathbf{V}, \mathbf{E})$ as a directed graph corresponding to a database schema \mathcal{D} . There are two types of nodes in \mathbf{V} : (a) *relation nodes*, \mathbf{R} , one for each relation in the schema; and (b) *attribute nodes*, \mathbf{A} , one for each attribute of each relation in the schema. Likewise, edges in \mathbf{E} are the following: (a) *projection edges*, $\mathbf{\Pi}$, each one connects an attribute node with its container relation node, representing the possible projection of the attribute in the system’s answer; and (b) *join edges*, \mathbf{J} , from a relation node to another relation node, representing a potential join between these relations. These could be joins that arise naturally due to foreign key constraints, but could also be other joins that are meaningful to a domain expert. Joins are directed for reasons explained later. Therefore, a database graph is a directed graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$, where: $\mathbf{V} = \mathbf{R} \cup \mathbf{A}$, and $\mathbf{E} = \mathbf{\Pi} \cup \mathbf{J}$.

A *weight*, w , is assigned to each edge of the graph \mathbf{G} . This is a real number in the range $[0, 1]$, and represents the significance of the bond between the corresponding nodes. Weight equal to 1 expresses strong relationship; in other words, if one node of the edge appears in an answer, then the edge should be taken into account making the other node appear as well. If a weight equals to 0, occurrence of one node of the edge in an answer does not imply occurrence of the other node. Based on the above, two relation nodes could be connected through two different join edges, in the two possible directions, between the same pair of attributes, but carrying different weights. For simplicity, we assume that there is at most one directed edge from one node to the same destination node.

A directed path between two relation nodes, comprising adjacent join edges, represents the “implicit” join between these relations. Similarly, a directed path between a relation node and an attribute node, comprising a set of adjacent join edges and a projection edge represents the “implicit” projection of the attribute on this relation. The weight of a path is a function of the weights of constituent edges, which should satisfy the condition that the estimated weight should decrease as the length of the path increases, based on human intuition and cognitive evidence. In our system, we have considered the product of weights over a path.

Logical Database Subsets. Consider a database \mathcal{D} properly annotated with a set of weights and a *précis query* Q , which is a set of tokens, i.e. $Q = \{k_1, k_2, \dots, k_m\}$. We define as *initial relation* any database relation that contains at least one tuple in which one or more query tokens have been found. A tuple containing at least one query token is called *initial tuple*.

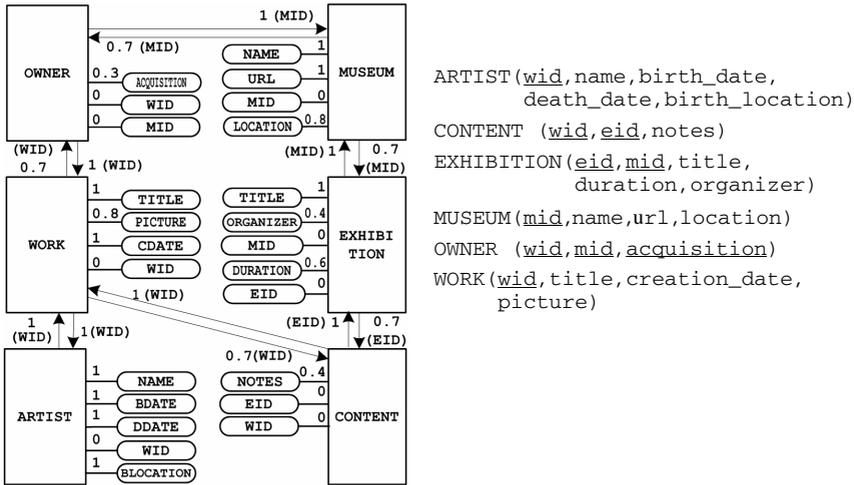


Fig. 1. An example database graph

A logical database subset D' of D satisfies the following:

- The set of relation names in D' is a subset of that in the original database D .
- For each relation R_i' in the result D' , its set of attributes in D' is a subset of its set of attributes in D .
- For each relation R_i' in the result D' , the set of its tuples is a subset of the set of tuples in the original relation R_i in D (when projected on the set of attributes that are present in the result).

The result of applying query Q on a database D given a set of constraints C is a logical database subset D' of D , such that D' contains initial tuples for Q and any other tuple in D that can be transitively reached by joins on D starting from *some* initial tuple, subject to the constraints C [11]. Possible constraints could be the maximum number of attributes in D' , the minimum weight of paths in D' , the maximum number of tuples in D' and so forth. Using different constraints and weights on the edges of the database graph allows generating different answers for the same query.

Précis Patterns. Weights and constraints may be provided in different ways. They may be set by the user at query time using an appropriate user interface. This option is attractive in many cases since it enables interactive exploration of the contents of a database. This bears a resemblance to query refinement in keyword searches. In case of précis queries, the user may explore different regions of the database starting, for example, from those containing objects closely related to the topic of a query and progressively expanding to parts of the database containing objects more loosely related to it. Although this approach is quite elegant, the user should spend some time on a procedure that may not always seem relevant to his need for a certain answer. Thus, weights and criteria may be pre-specified by a designer, or stored as part of a profile corresponding to a user or a group of users. In particular, in our framework, we have adopted the use of patterns of logical subsets corresponding to different queries or groups of users, which are stored in the system [20]. For instance, different patterns would be used to capture preferences of art reviewers and art fans.

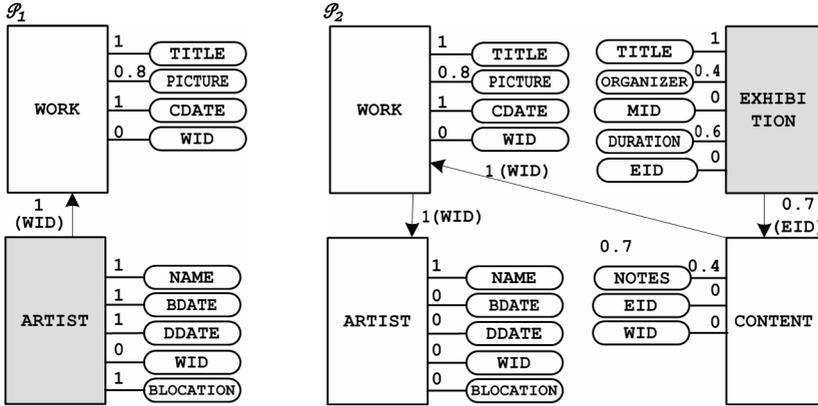


Fig. 2. Example précis patterns

Formally, given the database schema graph G of a database D , a *précis pattern* is a directed rooted tree $\mathcal{P}(\mathbf{v}, \mathbf{e})$ on top of G annotated with a set of weights. Given a query Q over database D , a précis pattern $\mathcal{P}(\mathbf{v}, \mathbf{e})$ is *applicable* to Q , if its root relation coincides with an initial relation for Q . The result of applying query Q on a database D given an applicable pattern \mathcal{P} is a logical database subset D' of D , such that:

- The set of relation names in D' is a subset of that in \mathcal{P} .
- For each relation R_i' in the result D' , its set of attributes in D' is a subset of its set of attributes in \mathcal{P} .
- For each relation R_i' in the result D' , the set of its tuples is a subset of the set of tuples in the original relation R_i in D (when projected on the set of attributes that are present in the result).

In order to produce the logical database subset D' , a précis pattern \mathcal{P} is enriched with tuples extracted from the database based on constraints, such as the maximum number of attributes in D' , the maximum number of tuples in D' and so on.

Example. Consider the database graph presented in Fig. 1. Observe the two directed edges between `WORK` and `OWNER`. Works and owners are related but one may consider that owners are more dependent on works than the other way around. In other words, an answer regarding an owner should always contain information about related works, while an answer regarding a work may not necessarily contain information about its owner. For this reason, the weight of the edge from `OWNER` to `WORK` is set to 1, while the weight of the edge from `WORK` to `OWNER` is 0.7. Précis patterns corresponding to different queries and/or groups of users may be stored in the system. In Fig. 2, patterns \mathcal{P}_1 and \mathcal{P}_2 correspond to different queries, regarding artists and exhibitions, respectively (the initial relation in each pattern is shown in grey).

4 System Architecture

In this section, we describe the architecture of a prototype précis query answering system, depicted in Fig. 3.

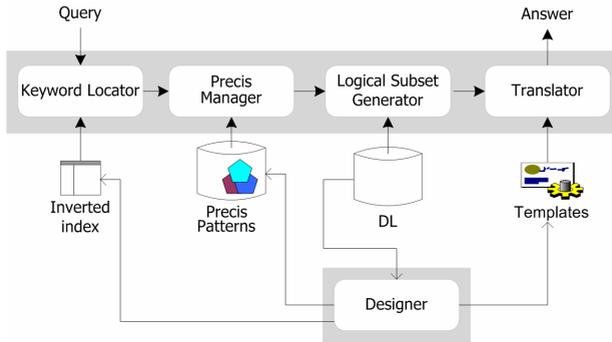


Fig. 3. System Architecture

Each time a user poses a question, the system finds the initial relations that match this query, i.e. database relations containing at least one tuple in which one or more query tokens have been found (Keyword Locator). Then, it determines the database part that contains information related to the query; for this purpose, it searches in a repository of précis patterns to extract an appropriate one (Précis Manager). If an appropriate pattern is not found, then a new one is created and registered in the repository. Next, this précis pattern is enriched with tuples extracted from the database according to the query keywords, in order to produce the logical database subset (Logical Subset Generator). Finally, an answer in the form of a précis is returned to the user (Translator). The creation and maintenance of the inverted index, patterns and templates is controlled through a Designer component. In what follows, we discuss in detail the design and implementation of these components.

Designer Interface. This module provides the necessary functionality that allows a designer to create and maintain the knowledge required for the system to operate, i.e.:

- *inverted index*: with a click of a button, the designer may create or drop the inverted index for a relational database.
- *templates*: through a graphical representation of a database schema graph, the designer may define templates to be used by the Translator.
- *user groups*: the designer may create pre-specified groups of users. Then, when a new user registers in the system, he may choose the group he belongs to.
- *patterns*: through a graphical representation of a database schema graph, the designer may define précis patterns targeting different groups of users and different types of queries for a specific domain. These are stored in a repository.

Manual creation of patterns and user groups assumes good domain and application knowledge and understanding. For instance, the pattern corresponding to a query about art works would probably contain the title and creation date of art works along with the names of the artists that created them and museums that own them; whereas the pattern corresponding to a query about artists would most likely contain detailed information about artists such as name, date and location of birth, and date of death along with titles of works an artist has created. Furthermore, different users or groups of users, e.g., art reviewers vs. art fans, would be interested in different logical subsets for the same query. We envision that the system could learn and adapt précis patterns

for different users or groups of users by using logs of past queries or by means of social tagging by large numbers of users. Then, the only work a designer would have to do would be the creation of templates.

Keyword Locator. When a user submits a précis query $Q = \{k_1, k_2, \dots, k_m\}$, the system finds the initial relations that match this query, i.e. database relations containing at least one tuple in which one or more query tokens have been found. For this purpose, an inverted index has been built, which associates each keyword that appears in the database with a list of occurrences of the keyword. Modern RDBMS' provide facilities for constructing full text indices on single attributes of relations (e.g., Oracle9i Text). In our approach, we chose to create our own inverted index, basically due to the following reasons: (a) a keyword may be found in more than one tuple and attribute of a single relation and in more than one relation; and (b) we consider keywords of other data types as well, such as date and number.

At its current version, the system considers that query keywords are connected with the logical operator OR . Keywords enclosed in quotation marks, e.g., "Leonardo da Vinci", are considered as one keyword that must be found in the same tuple. This means that the user can issue queries such as "Michelangelo" OR "Leonardo da Vinci", but not queries such as "Michelangelo" AND "Leonardo da Vinci", which would essentially ask about the connection between these two entities/people. We are currently working on supporting more complex queries involving operators AND and NOT .

Based on the above, given a user query, Keyword Locator consults the inverted index, and returns for each term k_i in Q , a list of all *initial relations*, i.e. $k_i \rightarrow \{R_j\}$, $\forall k_i$ in Q . (If no tuples contain the query tokens, then an empty answer is returned.)

Précis Manager. Précis Manager determines the schema of the logical database subset, i.e. the database part that contains information related to the query. This should involve initial relations and relations around them containing relevant information. The schema of the subset that should be extracted from a database given a précis query may vary depending on the type of the query issued and the user issuing the query. Patterns of logical subsets corresponding to different queries or groups of users are stored in the system. For instance, different patterns would be used to capture preferences of art reviewers and fans.

Each time an individual poses a question, Précis Manager searches into the repository of précis patterns to extract those that are appropriate for the situation. If users are categorized into groups, then this module examines only patterns assigned to the group the active user belongs to. Based on the initial relations identified for query Q , one or more applicable patterns may be identified. Recall that a précis pattern $\mathcal{P}(\mathbf{v}, \mathbf{e})$ is applicable to Q , if its root relation coincides with an initial relation for Q . For instance, given a query on "David", a pattern may correspond to artists ("Michelangelo") and another to owners ("Accademia di Belle Arti, Florence, Italy").

If none is returned for a certain initial relation, then the request is propagated to a Schema Generator. This module is responsible for finding which part of the database schema may contain information related to Q . The output of this step is the schema D' of a logical database subset comprised of: (a) relations that contain the tokens of Q ; (b) relations transitively joining to the former, and (c) a subset of their attributes that should be present in the result, according to the preferences registered for the user that poses the query. (For more details, we refer the interested reader to [20].) After its

creation, the schema of the logical database subset is stored in the graph database as a pattern associated with the group that the user submitting the query belongs to.

Logical Subset Generator. A précis pattern selected from the previous step is enriched with tuples extracted from the database according to the query keywords, in order to produce the logical database subset. For this purpose, the Logical Subset Generator starts from the initial relations where tokens in Q appear. Then, more tuples from other relations are retrieved by (foreign-key) join queries starting from the initial relations and transitively expanding on the database schema graph following edges of the pattern. Joins on a précis pattern are executed in order of decreasing weight. In other words, a précis pattern comprises a kind of a “plan” for collecting tuples matching the query and others related to them. At the end of this phase, the logical database subset has been produced.

Translator. The Translator is responsible for rendering a logical database subset to a more user-friendly synthesis of results. This is performed by a semi-automatic method that uses templates over the database schema. In the context of this work, the presentation of a query answer is defined as a proper structured management of individual results, according to certain rules and templates predefined by a designer. The result is a user-friendly response through the composition of simple clauses.

In this framework, in order to describe the semantics of a relation R along with its attributes in natural language, we consider that relation R has a conceptual meaning captured by its name, and a physical meaning represented by the value of at least one of its attributes that characterizes tuples of this relation. We name this attribute the *heading* attribute and we depict it as a hachured rounded rectangle. For example, in Fig. 1, the relation *ARTIST* conceptually represents “artists” in real world; indeed, its name, *ARTIST*, captures its conceptual meaning. Moreover, the main characteristic of an “artist” is its name, thus, the relation *ARTIST* should have the *NAME* as its heading attribute. By definition, the edge that connects a heading attribute with the respective relation has a weight 1 and it is always present in the result of a précis query. A domain expert makes the selection of heading attributes.

The synthesis of query results follows the database schema and the correlation of relations through primary and foreign keys. Additionally, it is enriched by alphanumeric expressions called *template labels* mapped to the database graph edges.

A *template label*, $label(u, z)$ is assigned to each edge $e(u, z) \in \mathbf{E}$ of the database schema graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$. This label is used for the interpretation of the relationship between the values of nodes u and z in natural language.

Each projection edge $e \in \mathbf{\Pi}$ that connects an attribute node with its container relation node, has a label that signifies the relationship between this attribute and the heading attribute of the respective relation; e.g., the *BIRTH_DATE* of an *ARTIST* (*.NAME*). If a projection edge is between a relation node and its heading attribute, then the respective label reflects the relationship of this attribute with the conceptual meaning of the relation; e.g., the *NAME* of an *ARTIST*. Each join edge $e \in \mathbf{J}$ between two relations has a label that signifies the relationship between the heading attributes of the relations involved; e.g., the *WORK* (*.TITLE*) of an *ARTIST* (*.NAME*). The label

of a join edge that involves a relation without a heading attribute signifies the relationship between the previous and subsequent relations.

We define as the label l of a node n the name of the node and we denote it as $l(n)$. For example, the label of the attribute node `NAME` is “name”. The name of a node is determined by the designer/administrator of the database. The template label $label(u, z)$ of an edge $e(u, z)$ formally comprises the following parts: (a) `lid`, a unique identifier for the label in the database graph; (b) $l(u)$, the name of the starting node; (c) $l(z)$, the name of the ending node; (d) $expr_1, expr_2, expr_3$ alphanumeric expressions. A simple template label has the form:

$$label(u, z) = expr_1 + l(u) + expr_2 + l(z) + expr_3$$

where the operator “+” acts as a concatenation operator.

In order to use template labels or to register new ones, we use a simple language for templates that supports variables, loops, functions, and macros.

The translation is realized separately for every occurrence of a token. At the end, the *précis* query lists all the clauses produced. For each occurrence of a token, the analysis of the query result graph starts from the relation that contains the input token. The labels of the projection edges that participate in the query result graph are evaluated first. The label of the heading attribute comprises the first part of the sentence. After having constructed the clause for the relation that contains the input token, we compose additional clauses that combine information from more than one relation by using foreign key relationships. Each of these clauses has as subject the heading attribute of the relation that has the primary key. The procedure ends when the traversal of the databases graph is complete. For further details, we refer the interested reader to [21].

User Interface. The user interface of our prototype comprises a simple form where the user can enter one or more keywords describing the topic of interest. Currently, the system considers that query keywords are connected with the logical operator `OR`. This means that the user can ask about “Michelangelo” `OR` “Leonardo da Vinci”, but cannot submit a query about “Michelangelo” and Leonardo da Vinci”, which essentially would ask about the connection between these two entities/people.

Before using the system, a user identifies oneself as belonging to one of the existing groups, i.e. art reviewers or fans. Fig. 4 displays an example of a user query and the answer returned by the system. Underlined topics are hyperlinks. Clicking such a hyperlink, the user implicitly submits a new query regarding the underlined topic. For example, clicking on “David” will generate a new *précis* regarding this sculpture. Hyperlinks are defined on heading attributes of relations.

Although extensive testing of the system with a large number of users has not taken place yet, a small number of people have used the system to search for pre-selected topics as well as topics of their interest and reported their experience. This has indicated the following:

- The *précis* query answering paradigm allows users with little or no knowledge of the application domain schema, to quickly and easily gain an understanding of the information space.
- Naïve users find *précis* answers to be user-friendly and feel encouraged to use the system more.

What can you tell me about:

▼

Michelangelo (March 6, 1475 - February 18, 1564) was born in Caprese, Tuscany, Italy. As a painter, Michelangelo's work includes [Holy Family of the Tribune \(1506\)](#), [The Last Judgment \(1541\)](#), [The Martyrdom of St. Peter \(1550\)](#). As a sculptor Michelangelo's work includes [Pieta \(1500\)](#), [David \(1504\)](#).

Fig. 4. Example précis query

- By providing précis of information as answers and hyperlinks inside these answers, the system encourages users to get involved in a continuous search-and-learn process.

5 Conclusions and Future Work

We have described the design, prototyping and evaluation of a précis query answering system with the following characteristics: (a) support of a keyword-based search interface for accessing the contents of the underlying collection, (b) generation of a logical subset of the database that answers the query, which contains not only items directly related to the query selections but also items implicitly related to them in various ways, (c) personalization of the logical subset generated and hence the précis returned according to the needs and preferences of the user as a member of a group of users, and (d) translation of the structured output of a précis query into a synthesis of results. The output is an English presentation of short factual information précis. As far as future work is concerned, we are interested in implementing a module for learning précis patterns based on logs of queries that domain users have issued in the past. In a similar line of research, we would like to allow users to provide feedback regarding the answers they receive. Then, user feedback will be used to modify précis patterns. Another challenge will be the extension of the translator to cover answers to more complex queries. Finally, we are working towards the further optimization of various modules of the system.

References

1. S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, pp. 5-16, 2002.
2. I. Androutsopoulos, G.D. Ritchie, and P. Thanisch. Natural Language Interfaces to Databases - An Introduction. *NL Eng.*, 1(1), pp. 29-81, 1995.
3. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pp. 431-440, 2002.
4. A. Dusterhoft, and B. Thalheim. Linguistic based search facilities in snowflake-like database schemes. *DKE*, 48, pp. 177-198, 2004.

5. D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6), 2000.
6. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRank: Ranked keyword search over XML documents. In *SIGMOD*, pp. 16-27, 2003.
7. L. R. Harris. User-Oriented Data Base Query with the ROBOT Natural Language Query System. *VLDB 1977*: 303-312.
8. V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, pp. 850-861, 2003.
9. V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pp. 367-378, 2003.
10. IBM. *DB2 Text Information Extender*. url: www.ibm.com/software/data/db2/extender/textinformation/.
11. G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. In *ICDE*, 2006.
12. G. Marchionini. Interfaces for End-User Information Seeking. *J. of the American Society for Inf. Sci.*, 43(2), 156-163, 1992.
13. U. Masermann, and G. Vossen. Design and implementation of a novel approach to keyword searching in relational databases. In *ADBIS-DASFAA*, pp. 171-184, 2000.
14. E. Metais, J. Meunier, and G. Levreau. Database Schema Design: A Perspective from Natural Language Techniques to Validation and View Integration. In *ER*, pp. 190-205, 2003.
15. E. Metais. Enhancing information systems management with natural language processing techniques. *DKE*, 41, pp. 247-272, 2002.
16. Microsoft. *SQL Server 2000*. url: <http://msdn.microsoft.com/library/>.
17. M. Minock. A Phrasal Approach to Natural Language Interfaces over Databases. In *NLDB*, pp. 181-191, 2005.
18. A. Motro. Constructing queries from tokens. In *SIGMOD*, pp. 120-131, 1986.
19. Oracle. *Oracle 9i Text*. url: www.oracle.com/technology/products/text/.
20. A. Simitsis, and G. Koutrika. Pattern-Based Query Answering. In *PaRMa*, 2006.
21. A. Simitsis, and G. Koutrika. Comprehensible Answers to Précis Queries. In *CAiSE*, pp. 142-156, 2006.
22. E. Sneider. Automated Question Answering Using Question Templates That Cover the Conceptual Model of the Database. In *NLDB*, pp. 235-239, 2002.
23. V.C. Storey, R.C. Goldstein, H. Ullrich. Naive Semantics to Support Automated Database Design. *IEEE TKDE*, 14(1), pp. 1-12, 2002.
24. V.C. Storey. Understanding and Representing Relationship Semantics in Database Design. In *NLDB*, pp. 79-90, 2001.
25. A. Toral, E. Noguera, F. Llopis, and R. Munoz. Improving Question Answering Using Named Entity Recognition. In *NLDB*, pp. 181-191, 2005.
26. Q. Wang, C. Nass, and J. Hu. Natural Language Query vs. Keyword Search: Effects of Task Complexity on Search Performance, Participant Perceptions, and Preferences. In *INTERACT*, pp. 106-116, 2005.