

Comprehensible Answers to Précis Queries

Alkis Simitsis¹ and Georgia Koutrika²

¹ National Technical University of Athens,
Department of Electrical and Computer Engineering,
Athens, Greece

asimi@dbnet.ece.ntua.gr

² University of Athens,
Department of Computer Science,
Athens, Greece

koutrika@di.uoa.gr

Abstract. Users without knowledge of schemas or query languages have difficulties in accessing information stored in databases. Commercial and research efforts have focused on keyword-based searches. Among them, précis queries generate entire multi-relation databases, which are logical subsets of existing ones, instead of individual relations. The logical database subset contains not only items directly related to the query selections but also items implicitly related to them in various ways. Earlier work has identified the need of providing the naïve user with meaningful answers to his questions and has suggested the translation of précis query answer in narrative form. In this paper, we present a semi-automatic method that translates the relational output of a précis query into a synthesis of results. We describe a translator engine that uses a template mechanism for generating a précis in a narrative form through a set of reusable templates.

1 Introduction

The need for facilitating access to information stored in databases has been early recognized in the research community with initial efforts dating back to seventies [7]. Emergence of the World Wide Web has made information access possible to a growing number of people. A large fraction of information resides in databases, as libraries, museums, and other organizations publish their electronic contents on the Web. In the same time, most users have no specific knowledge of schemas or structured query languages for accessing information stored in a database. In this context, the need for facilitating access to information stored in databases becomes increasingly more important.

Existing efforts have mainly focused on facilitating querying over relational databases proposing either handling natural language queries [2, 13, 16] or free-form, i.e. keyword-based, queries [1, 18]. In this work, we focus on a relative, still novel, issue of generating meaningful answers to queries and we propose an approach to translate the relational output of a query into a form that resembles narration and is thus more comprehensible to the naïve user.

In particular, we consider the output of précis queries [11]. These are free-form queries that generate entire multi-relation databases, which are logical subsets of existing ones, instead of individual relations. The logical subset of the database generated by a précis query contains not only items directly related to the query selections but also items implicitly related to them in various ways. Logical database subsets are useful in many cases. Given large databases, enterprises often need smaller subsets that conform to the original schema and satisfy all of its constraints in order to perform realistic tests of new applications before deploying them to production. Likewise, software vendors need such smaller but correct databases to demonstrate new software product functionality. Additionally, non-expert users would rather expect a summary or précis of the information contained in a logical subset. For instance, a more meaningful response than the classic “tabular-form” answer to a query that asks about “Woody Allen” might be in the form of the following précis:

“Woody Allen was born on December 1, 1935 in Brooklyn, New York, USA. As a director, Woody Allen’s work includes Match Point (2005), Melinda and Melinda (2004), Anything Else (2003). As an actor, Woody Allen’s work includes Hollywood Ending (2002), The Curse of the Jade Scorpion (2001).”

A précis may be incomplete in many ways; for example, the abovementioned précis includes a non-exhaustive list of Woody Allen’s works. Nevertheless, it provides sufficient information in a comprehensible way to help one learn about “Woody Allen” and possibly identify new keywords for further searching. For example, one may decide to explicitly issue a new query about “Anything Else” or implicitly by following underlined topics (hyperlinks) to pages containing more relevant information.

Contributions. This paper deals with the presentation of a précis answer to a keyword query over a relational database. In brief, the contributions of this paper are the following.

- We extend the functionality of précis queries, by enriching the model with labels attached to its constructs. We propose a formal way to compose these labels through a simple to use language.
- We present a mechanism for the definition and instantiation of template labels.
- We present a semi-automatic method that translates the relational output of a précis query into a narrative synthesis of results.

Outline. The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes the general framework of précis queries. Section 4 presents a technique for the translation of the information produced by précis queries in a narrative form. Finally, 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, 12] and XML data [5, 6, 9]. Oracle 9i Text [19], Microsoft SQL Server [15] 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 “Woody Allen” would be in the form of relation-attribute pair, such as (Director, Name). In many cases, this answer may suffice, but in many practical scenarios it conveys little information about “Woody Allen”. A more complete answer containing, for instance, information about this director's movies and awards would be more meaningful and useful instead. 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., movies directed by Woody Allen. 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.

As we have already mentioned, 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. This is inspired by BAROQUE [17] and shields the user from the particularities of the underlying data schema and model in use. BAROQUE uses a network representation of a database and defined several types of relationships in order to support functions that scan this network. However, it only locates the position and the relationships in which an item participates. As the database representation adopted does not include joins, it cannot assemble answers split into several relations.

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, 14], NL and Schema Design [22, 13, 4], NL and DB interfaces [16, 2], and Question Answering [24, 21]. As far as we are aware of, 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 [23]. There exist some recent efforts that use phrasal patterns or question templates to facilitate the answering procedure [16, 21]. Also, there exists a recent experimental study [26] that compares NL Querying versus keyword search and supports the usefulness of the latter especially in the presence of complex queries.

This paper deals with the generation of meaningful answers from keyword queries and develops an approach to translate the relational output of a query into a form that resembles narration and is thus more comprehensible to a user. The process resembles those involved in handling natural language query 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 our approach uses well defined templates to rephrase relations and tuples. Furthermore, it has the advantage that it is not limited by any dictionary, because it concerns relational databases where the schemata are predictable and familiar to an expert, e.g., the dba; thus the template mechanism introduced later in this paper is sufficient for our aim. Moreover, précis queries are keyword queries which can lead to complex SQL queries whose form is only limited by the database schema graph. Works such [21] use a set of pre-defined question patterns, which cannot claim for completeness, i.e. this set is difficult to capture any possible query over a given database. Furthermore, 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 suggested in [11] and we elaborate on the idea of translating a logical database subset generated by a précis query into a narrative piece of information.

3 The Précis Query Framework

The purpose of this section is to provide essential background information on précis queries. First, we describe how a database can be modeled as a graph, and we introduce an example that we refer to throughout the paper. Next, we describe the précis query model and the system architecture of our framework.

3.1 Preliminaries

We consider the *database schema graph* $\mathbf{G}(\mathbf{v}, \mathbf{e})$ as a directed graph corresponding to a database schema \mathbf{D} . Nodes in \mathbf{v} are: (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. Edges in \mathbf{e} are: (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 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 as explained later. For simplicity in presentation, we assume that (a) primary keys are not composite; thus, an attribute from a relation joins to an attribute from another relation, and (b) these attributes have the same name. The common name of the joining attributes is tagged on the respective join edge between the two relations.

Therefore, a database graph is formally defined as a directed graph $\mathbf{G}(\mathbf{v}, \mathbf{e})$, where: $\mathbf{v} = \mathbf{R} \cup \mathbf{A}$, and $\mathbf{e} = \mathbf{\Pi} \cup \mathbf{J}$. The notation for its graphical representation is given in Fig. 1.

A *weight*, w , is assigned to each edge of the graph G . This is a real number in $[0, 1]$ representing the significance of the connection between the nodes involved. 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. 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. A directed join edge expresses the dependence of the source relation of the join on the target. The source relation indicates the relation already considered for the answer and the target corresponds to the relation that may be included influencing the final result, if the join is applied. For simplicity, we assume that there is at most one directed edge from one node to the same destination node.

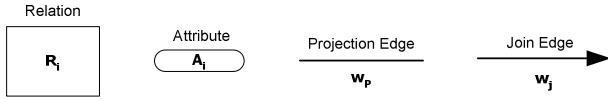


Fig. 1. Representation of graph elements

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 weight of constituent edges. This function should satisfy the condition that the weight decreases as the length of the path increases, based on human intuition and cognitive evidence.

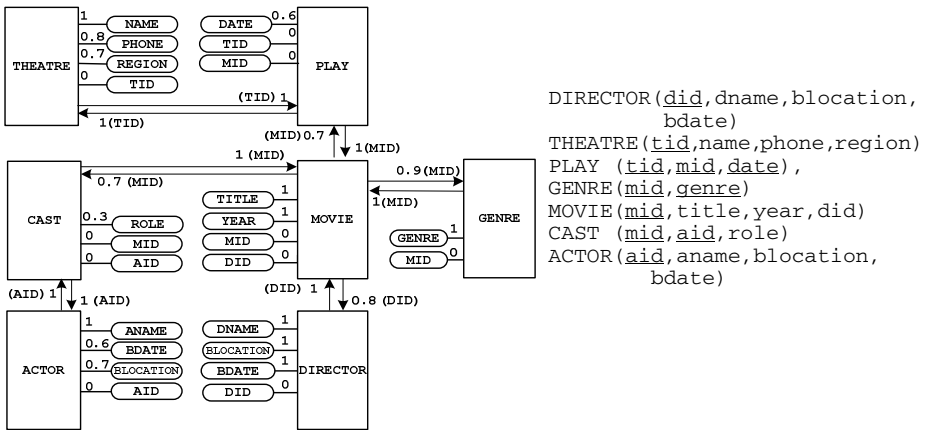


Fig. 2. An example database graph

Example. Consider a movies database¹ described by the schema presented in Fig. 2; primary keys are underlined. For instance, observe the two directed edges between MOVIE and GENRE. Movies and genres are related but one may consider that genres are more dependent on movies than the other way around. In other words, an answer regarding a genre should always contain information about related movies, while an answer regarding a movie may not necessarily contain information about its genres. For this reason, the weight of the edge from GENRE to MOVIE is set to 1, while the weight of the edge from MOVIE to GENRE is set to 0.9.

Using different weights on the graph's edges allows constructing different answers to the same query. Weights may be provided in different ways. They may be set by a user at query time using an appropriate user interface that enables interactive exploration of the contents of a database. A 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. Alternatively, sets of weights corresponding to different queries or groups of users may be stored in the system [20]. For instance, different sets would capture preferences of movie reviewers and filmgoers. The former may be typically interested in in-depth, detailed answers; using an appropriate set of weights would enable these users to explore larger parts of the database around a single précis query. On the other hand, cinema fans usually prefer shorter answers. In this case, a different set of weights would allow producing answers containing only highly related objects. Finally, multiple sets of weights corresponding to different user profiles may be stored in the system. Using user-specific weights allows generating personalized answers. For example, a user may be interested in the region where a theatre is located, while another may be interested in a theatre's phone.

However, the approach presented is general in that it does not depend on a specific weight-model.

3.2 Précis Query Model

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*.

A *logical database subset* \mathcal{D}' of \mathcal{D} satisfies the following:

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

The result of applying query Q on a database \mathcal{D} given a set of constraints \mathcal{C} is a logical database subset \mathcal{D}' of \mathcal{D} , such that \mathcal{D}' contains initial tuples for Q and any other tuple in \mathcal{D} that can be transitively reached by (foreign-key) joins on \mathcal{D} starting from

¹ www.imdb.com

some initial tuple, subject to the constraints in C . Possible constraints in C could include the maximum number of joins, the maximum number of tuples in D' and so forth. Using different constraints allows generating different answers for the same query and the same set of weights over the edges of the database graph. Similarly to weights, constraints may be specified at query time, or be pre-stored in the system.

3.3 System Architecture

Given a précis query $Q = \{k_1, k_2, \dots, k_m\}$, the following steps are performed in order to generate an answer.

A keyword may be found in more than one tuples and attributes of a single relation and in more than one relations. For this reason, the system uses an inverted index that returns for each term k_i in Q , a list of all its occurrences. A single keyword occurrence is a tuple $\langle R_j, A_{1j}, Tid_{1j} \rangle$, where Tid_{1j} is the id of a tuple in relation R_j that contains keyword k_i as part of the value of attribute A_{1j} . If no tuples contain the query tokens, the following steps are not executed.

Next, the system maps all initial relations returned from the inverted index on the database schema graph G and tries to find which part of the graph may contain information related to Q . The output of this step is the schema of the logical database subset D' involving initial relations and relations transitively joining to the former and a subset of their attributes that should be present in the result according to the constraints provided.

Finally, the system populates relations in the logical database subset starting from initial relations. More tuples from other relations are retrieved by join queries starting from initial relations and transitively expanding on the logical database subset schema graph. At the end of this phase, the logical database subset is produced.

More technical details for the two steps above, along with the algorithms involved, can be found in [11]. As we have already discussed, in this work, we are mainly concerned with the exploitation of the information stored in the logical database subset. In what follows, we take a step further towards facilitating access of information in databases. This is performed by using information conveyed by the database graph, which may be properly annotated to further enhance its semantics.

4 Translator

In this section, we present a semi-automatic method to render the SQL-like response of a précis query to a more user-friendly synthesis of results. 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 or the administrator of the database. Clearly, we do not anticipate the construction of a human-intelligent system; rather, we try to provide a user-friendly response through the composition of simple clauses.

4.1 Preliminaries

In our 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. 2, the relation MOVIE conceptually represents “movies” in real world; indeed, its name, MOVIE, captures its conceptual meaning. Moreover, the main characteristic of a “movie” is its “title”, thus, the relation MOVIE should have the TITLE as its heading attribute, since the word “title” captures the physical meaning of a “movie”.

Heading Attributes. The heading attribute, h_R , of a relation R is defined as the attribute whose name represents the physical meaning of that relation. By definition, the projection edge that connects a heading attribute with the respective relation has a weight 1 and this attribute is always present in the result of a précis query. A domain expert makes the selection of heading attributes, at the initial construction of the database graph.

We do not anticipate that all relations should have a heading attribute. For instance, a relation used only for storing *n-to-m* relationships between different entities (e.g., relation CAST in Fig. 2) does not require a heading attribute. Clearly, this is not a problem, since, in general, these relations are used only for the construction of paths that represent query answers and have no attributes in the logical database subset.

Labels. Each projection edge $e \in \Pi$ that connects an attribute a with its container relation R , is annotated by a label that signifies the meaning, in terms of natural language, of the relationship between this attribute and the heading attribute of the respective relation. For instance, with respect to the design of Fig. 2, a possible label attached to the projection edge between the relation MOVIE and its attribute YEAR may be: “the YEAR of a MOVIE (.TITLE)”; recall, that TITLE is the heading attribute of MOVIE.

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 TITLE of a MOVIE.

Each join edge $e \in \mathcal{J}$ between two relations has a label that signifies the relationship between the heading attributes of the relations involved; e.g., the GENRE (.GENRE) of a MOVIE (.TITLE). The label of a join edge that involves a relation without a heading attribute signifies the relationship between the previous and subsequent relations.

4.2 Template Mechanism

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 edges of the database schema graph.

Templates. 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 a narrative form.

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 TITLE is “title”. The name of a node should be determined by the designer/administrator of the database.

The template label $\text{label}(u, z)$ of an edge $e(u, z)$ formally comprises the following elements: (a) a unique identifier for the label in the database graph; (b) the name of the starting node, i.e. $l(u)$; (c) the name of the ending node, i.e. $l(z)$; and (d) several alphanumeric expressions.

A simple template label has the form:

$$\text{label}(u, z) = \text{expr}_1 + l(u) + \text{expr}_2 + l(z) + \text{expr}_3$$

where $\text{expr}_1, \text{expr}_2, \text{expr}_3$ are alphanumeric expressions and 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. A similar approach, but, still, in a totally different environment, can be found in [25] where the authors present a template mechanism for the description of ETL processes in a data warehouse environment. Below, we describe this language.

In a template, when we refer to the conceptual meaning of a node, we simply use its name. When an instance of the node is needed, then we use the node as a variable. There are two kinds of variables: *parameter variables* and *loop iterators*.

Parameter Variables. Parameter variables are marked with a @ symbol at their beginning and are replaced by values at instantiation time. For example, a template label for the projection edge $e(\text{PHONE}, \text{THEATRE})$ could be:

$\text{label}(\text{PHONE}, \text{THEATRE}) = \text{“The PHONE of the THEATRE @THEATRE.NAME is @PHONE”}$

where PHONE and THEATRE stand for the conceptual meaning of the nodes (attribute and relation, respectively) PHONE and THEATRE; i.e., $l(\text{PHONE}) = \text{“phone”}$ and $l(\text{THEATRE}) = \text{“theatre”}$ respectively. Moreover, @THEATRE.NAME and @PHONE are parameter variables, with possible values “ALPHAVILLE” and “12345”. In this case, a valid label for this edge can be the following:

“The phone of the theatre ALPHAVILLE is 12345”

In several cases, the values returned in a query result from a certain attribute could be more than one. Then, we use a list of parameters denoted as:

@<parameter name> []

For such lists, their length should be provided at instantiation time.

Loop Iterators. Loop iterators are implicitly defined in the loop constraint, as we will discuss later. In each round of the loop, all the properly marked appearances of the iterator in the loop body are replaced by its current value (similarly to the way a C preprocessor treats #DEFINE statements). Iterators that appear marked in the loop body are instantiated even when they are part of another string or a variable name. We mark such occurrences by enclosing them between \$. This functionality enables referencing all values of a parameter list and facilitates the creation of an arbitrary number of pre-formatted strings.

Functions. We employ a built-in function, $\text{arityOf}(\langle \text{list_of_parameters} \rangle)$, which returns the arity of a list of parameters, mainly in order to define upper bounds in loop iterators.

Loops. Loops enhance the genericity of the templates by allowing the designer to handle templates with unknown number of variables and with unknown arity for parameters involved. The general form of loops is:

```
[<simple constraint>] {(loop body)},
```

where simple constraint has the form:

```
<lower> <operator> <iterator> <operator> <upper>
```

We consider only linear increase with step equal to 1. Upper bound and lower bound (default value 1) can be arithmetic expressions involving `arityOf()` function calls, variables and constants. Valid arithmetic operators are `+`, `-`, `/`, `*` and valid comparison operators are `<`, `>`, `=`, all with their usual semantics. During iterations the loop body is reproduced and at the same time all the marked appearances of the loop iterator are replaced by its current value, as described before. Loop nesting is permitted. For instance, consider the following case:

```
[i<arityOf(MOVIE)] {MOVIE_{$i$}}
```

In this case, the lower bound has the default value (1), and the upper bound is limited by the number (arity) of attributes of the relation `MOVIE`. Thus, the iterator `i` takes value between 1 and the total attributes of `MOVIE`. As far as the loop body is concerned, it contains a parameter list that stores the attributes involved in the relation `MOVIE`.

For the example database depicted in Fig. 2, with respect to the relation `MOVIE`, the loop that represents its attributes has the following form:

```
[i<2] {MOVIE_{$i$}}
```

and at the instantiation of the parameters, we get the following results: `MOVIE_1 = TITLE` (first attribute) and `MOVIE_2 = YEAR` (second attribute).

Macros. We introduce macros to ease the definition and to improve the readability of templates. Macros facilitate attribute and variable name expansion. For instance, one major problem in defining a language for templates is the difficulty of dealing with attributes or attribute values of arbitrary arity. At the template level, it is not possible to pin-down the number of (a) attributes that are projected in the précis query, and (b) values of the involved attributes, to a specific value.

For example, in order to find out:

- (a) The attributes projected in a certain précis query we need to create a series of attributes like the following:

```
DEFINE MOVIES_LIST as
    [i<arityOf(MOVIE)] {MOVIE_{$i$},}
    [i=arityOf(MOVIE)] {MOVIE_{$i$}}
```

- (b) The titles of movies that correspond to a certain query, we need to create a series of values as follows:

```
DEFINE MOVIES_TITLES_LIST as
    [i<arityOf(@MOVIE.TITLE)] {@MOVIE.TITLE[{$i$}],}
    [i=arityOf(@MOVIE.TITLE)] {@MOVIE.TITLE[{$i$}]}
```

For the example database of Fig. 2, the attribute and value series are:

```
MOVIES_LIST = {TITLE, YEAR}
MOVIES_TITLES_LIST = {"Match Point",
                      "Melinda and Melinda",
                      "Anything Else"}
```

Note the existence of the two loops in each macro in order to avoid the presence of an erroneous “,” after the last value in each list.

4.3 Translation

Additionally, we present a method that parses the result database graph and composes a synthesis of query results in a narrative form.

The translation is realized separately for every occurrence of a token. At the end, the *précis* query lists all 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. It becomes obvious that for multiple attributes of the same relation we have to repeat several times the same subject. To avoid this, a domain expert should have attached suitable expressions in the projection edges, in order to allow the construction of complex sentences that make sense.

For instance, consider Fig. 2. Assume that in relation `DIRECTOR` the labels of the projection edges that connect the heading attribute, `DNAME`, with attributes `BDATE` and `BLOCATION`, which store information about the birth data and birth location of a director, are the following:

```
label(hR, BDATE) = @DNAME + " was born" + " on " + @BDATE
label(hR, BLOCATION) = @DNAME + " was born" + " in " + @BLOCATION
```

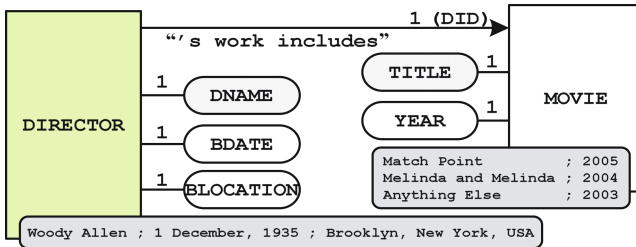
When both attributes are involved in the answer, then the clause derived from the `DIRECTOR` relation could be as follows:

```
"@DNAME was born on @BDATE in @BLOCATION"
```

This operation is realized as a simple find-and-replace mechanism, namely `resolve_common_expressions`, which finds common expressions in the clauses that respond to each label attached to a projection edge. In the example above, the common expressions are `@DNAME` and “ was born ”.

The procedure used for the translation of the information stored in a relation is depicted in Fig. 3.

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. In the example of Fig. 2, the `DIRECTOR` relation is connected to the `MOVIE` relation through the `DID` key. The subject of the respective clause will be the `DNAME` attribute, while the rest is constructed in a sense similar to the one described before. The procedure terminates

Algorithm Translation of a Relation (TR)Input: a relation R , a set of tokens \mathbf{T} , a database graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ Output: an array of sentences $\text{Sentence}[]$ **Begin****For each** token $t \in \mathbf{T}$ Sentence[t] = ''Let $R \in \mathbf{V}$ be the container relation of t Let h_R be the heading attribute of R clause[t, h_R] = 1(h_R)**For each** attribute a in R , $a \neq h_R$ clause[t, a] = label(h_R, a)**End for**Sentence[t] = Sentence[t] + resolve_common_expressions(clause[])**End for****End.****Fig. 3.** The algorithm TR**Fig. 4.** A part of our example database

when the traversal of the databases graph is complete. In addition, for each attribute projected in the answer, a hyperlink may be created. When a user follows a hyperlink, a new précis query is submitted containing the hyperlink's text.

Consider the case of the database of Fig. 2. Assume that a logical database subset concerning a user's question (e.g., the token "Woody Allen") involves only the relations DIRECTOR and MOVIE, while the schema of the logical subset is depicted in Fig. 4. At first, we consider the case of "Woody Allen" as a director. We construct the template clause that derives from the DIRECTOR relation, as before:

@DNAME + " was born" + " on " + @BDATE + " in " + @BLOCATION

Next, we built the respective template clause that derives from the MOVIE relation:

@TITLE + " (" + @YEAR + ")"

Then, we proceed with the clause composed by the join relationship that connects the relations DIRECTOR and MOVIE. The template label of this relationship is represented with the following formula:

label(DIRECTOR,MOVIE) = expr_1 + @DNAME + expr_2 + MOVIE_LIST

The macro `MOVIE_LIST` and the expressions may be defined as:

```
DEFINE MOVIE_LIST as
  [i<arityOf(@TITLE)]
    {@TITLE[$i$]+ " (+@YEAR[$i$]+) ," }
  [i=arityOf(@TITLE)]
    {@TITLE[$i$]+ " (+@YEAR[$i$]+) ." }
expr_1 ← "As a director, "
expr_2 ← "'s work includes "
```

Therefore, the result of the *précis* query for the token “Woody Allen” located in the relation `DIRECTOR` will be:

“Woody Allen was born on December 1, 1935 in Brooklyn, New York, USA. As a director, Woody Allen’s work includes Match Point (2005), Melinda and Melinda (2004), Anything Else (2003).”

In the general case, if we enrich the above example with the constraint that only projections with weight equal to or greater than 0.9 should be present in the answer and up to three tuples should be retrieved per relation, then the logical subset contains also the relations `GENRE`, `ACTOR`, and `CAST`.

In this case, we retain the previous result about `DIRECTOR` and `MOVIE`, and we proceed with the clause composed by the join relationship between the `MOVIE` and `GENRE` relations. The template label of this relationship is represented with the following formula:

```
label(MOVIE,GENRE) = @TITLE + expr_2 + GENRE_LIST
DEFINE GENRE_LIST as
  [i<arityOf(@GENRE)] {@GENRE[$i$]+ " ," }
  [i=arityOf(@TITLE)] {@GENRE[$i$]+ " ." }
expr_2 ← " is "
```

Therefore, the result of the *précis* query for the token “Woody Allen” located in the relation `DIRECTOR` will be:

“Woody Allen was born on December 1, 1935 in Brooklyn, New York, USA. As a director, Woody Allen’s work includes Match Point (2005), Melinda and Melinda (2004), Anything Else (2003). Match Point is Drama, Thriller. Melinda and Melinda is Comedy, Drama. Anything Else is Comedy, Romance.”

In a similar way, we construct the result of the *précis* query for the token “Woody Allen” located in the relation `ACTOR`. Recall that the relation `CAST` does not have a heading attribute. Thus, the designer should enrich the join edges that interconnect the three relations `ACTOR`, `CAST`, and `MOVIE` with an appropriate label. An example template label could be the following:

“As an actor, @ACTOR’s work includes `MOVIE_LIST`”

and so, given that the label for join relationship between the `MOVIE` and `GENRE` relations is constructed as before, the result of the *précis* query will be:

“As an actor, Woody Allen’s work includes Hollywood Ending (2002), The Curse of the Jade Scorpion (2001), Picking Up the Pieces (2000). Hollywood Ending is Comedy, Drama. The Curse of the Jade Scorpion is Comedy, Drama. Picking Up the Pieces is Comedy, Fantasy.”

As we mentioned before, if there does not exist any information that both instance values refer to the same physical entity, then, the answer of the précis query comprises two parts, one for each occurrence of the token as shown in the example above. Otherwise, the answers can be merged to produce a fancier result.

5 Conclusions and Future Work

Précis queries are free-form queries that generate entire multi-relation databases, which are logical subsets of existing ones, instead of individual relations. The logical subset of the database generated by a précis query contains not only items directly related to the query selections but also items implicitly related to them in various ways. Earlier work has identified the need of providing the naïve user with meaningful answers to his/her questions and has suggested the translation of a précis query answer in narrative form.

In this paper, we have extended précis queries by presenting a semi-automatic method that turns the relational output of a précis query into a narrative synthesis of results. 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 or the administrator of the database. More specifically, we have extended the functionality of précis queries, by enriching the model with labels attached to its constructs. Moreover, we have proposed a formal way to compose these labels through a simple to use language. Also, we have presented a template mechanism for the definition and instantiation of template labels. Finally, we have proposed a semi-automatic method that translates the relational output of a précis query into a narrative synthesis of results.

Clearly, as we have already stressed, we do not anticipate the construction of a human-intelligent system; rather, we try to provide a user-friendly response through the composition of simple clauses, so that a user without any particular knowledge of relational schemas or languages may understand and use the information returned to him/her.

We are currently experimenting with users to solidify the evidence on the effectiveness of our approach. From the feedback obtained, we plan to improve the performance/“intelligence” of the translator presented in this paper. In a similar line of research, a challenging issue is the extension of précis queries to providing ranked or top-k results.

Acknowledgments. 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.

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. Androutopoulos, 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/extenders/textinformation/.
11. G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. In *ICDE*, 2006.
12. 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.
13. 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.
14. E. Metais. Enhancing information systems management with natural language processing techniques. *DKE*, 41, pp. 247-272, 2002.
15. Microsoft. *SQL Server 2000*. url: <http://msdn.microsoft.com/library/>.
16. M. Minock. A Phrasal Approach to Natural Language Interfaces over Databases. In *NLDB*, pp. 181-191, 2005.
17. A. Motro. Baroque: A browser for relational databases. *ACM Trans. Inf. Syst.*, 4(2), pp. 164-181, 1986.
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. E. Sneiders. Automated Question Answering Using Question Templates That Cover the Conceptual Model of the Database. In *NLDB*, pp. 235-239, 2002.
22. V.C. Storey, R.C. Goldstein, H. Ullrich. Naive Semantics to Support Automated Database Design. *IEEE TKDE*, 14(1), pp. 1-12, 2002.
23. V.C. Storey. Understanding and Representing Relationship Semantics in Database Design. In *NLDB*, pp. 79-90, 2001.
24. A. Toral, E. Noguera, F. Llopis, and R. Munoz. Improving Question Answering Using Named Entity Recognition. In *NLDB*, pp. 181-191, 2005.
25. P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos. A Generic and Customizable Framework for the Design of ETL Scenarios. *Information Systems*, 30(7), pp. 492-525, 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.