Pattern-Based Query Answering

Alkis Simitsis[1], Georgia Koutrika[2]

[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 structured 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. A logical database subset contains not only items directly related to the query selections but also items implicitly related to them in various ways. Existing approaches to précis query answering assume that a database is pre-annotated with a set of weights, and when a query is issued, an ad-hoc logical subset is constructed on the fly. This approach has several limitations, such as dependence on users for providing appropriate weights and constraints for answering précis queries, and difficulty to capture different query semantics and user preferences. In this paper, we propose a pattern-based approach to logical database subset generation. Patterns of logical subsets corresponding to different queries or user preferences may be recognized and stored in the system. Each time a user poses a question, the system searches in a repository of précis patterns to extract an appropriate one. Then, this is enriched with tuples extracted from the database, in order to produce the logical database subset.

1. Introduction

The need for facilitating access in information stored in a database for users with no specific knowledge of schemas or structured query languages has been acknowledged, especially in the context of web accessible databases, as libraries, museums, and other organizations publish their electronic contents on the Web. Towards this direction, current commercial and research efforts have focused on keyword-based searches. Among them, *précis queries* are free-form queries that generate entire multi-relation databases, which are logical subsets of existing ones, instead of individual relations [10]. The logical subset of a 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. This subset is useful in many cases and provides to the user much greater insight into the original data.

For instance, a user asking about "Woody Allen" would probably like to know a little bit more than that "*Woody Allen is a director*". A more meaningful response would 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).*"

This response provides sufficient information to help someone learn about Allen and identify new keywords for further searching. For example, the user may decide to explicitly issue a new query about "*Anything Else*" or implicitly by following underlined topics (hyperlinks) to pages containing more relevant information. On the other hand, 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. Based on the above, support of précis queries over databases and generation of logical database subsets comprises an advanced searching paradigm helping users to gain insight into the contents of a database.

Given a précis query, a system would first determine the schema of the logical database subset, i.e. the database part that contains information related to the query, and then extract tuples from the database with the use of appropriate SQL queries in order to populate this subset. 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. For instance, the logical subset corresponding to a query about movies would probably contain the title, year and duration of movies along with the names of directors and actors; whereas the logical subset corresponding to a query about actors would most likely contain detailed information about actors such as name, date and location of birth, and nationality and only titles of movies an actor has starred in. Furthermore, different users or groups of users, e.g., movie reviewers vs. filmgoers, would be interested in different logical subsets for the same query.

Existing approaches to précis query answering assume that each entity and relationship of a database is pre-annotated with a weight determining its significance for a certain user [10]. When a query is issued, the appropriate logical subset is constructed on the fly based on syntactic criteria issued by the user at query time or pre-stored in the system. This approach has several drawbacks: dependence on users for providing appropriate weights and criteria for answering précis queries, difficulty to capture different query semantics and user preferences in the same time, and inefficient execution since a logical subset is generated from scratch each time a query is issued.

However, as the examples above illustrate, patterns of logical subsets corresponding to different queries or groups of users may be recognized and stored in the system. For instance, different patterns would be used to capture preferences of movie reviewers and filmgoers. In this context, each time a user poses a question, the system searches in a repository of *précis patterns* to extract an appropriate one. Then,

this précis pattern is enriched with tuples extracted from the database according to the query keywords, in order to produce the logical database subset.

Furthermore, apart from the benefit of getting a pre-stored schema for a logical subset of a database instead of creating from scratch a new one, we exploit the presence of précis patterns in our framework in a two-fold manner: (a) incremental population of a logical database subset, and (b) pre-storing answers for the most frequent précis queries.

Contributions. In brief, the contributions of our paper are the following.

- We propose a pattern-based approach to logical database subset generation. Précis patterns may capture semantics of different précis queries or preferences of different user groups and improve the efficiency of generation of logical database subsets from précis queries.
- We present the architecture of a system that produces logical database subsets according to précis queries posed by individuals using précis patterns extracted from the repository and describe methods that implement the required functionality.
- We discuss two optimization techniques that are used to further improve the efficiency and effectiveness of the system: incremental population of a logical database subset and using pre-stored answers.

Outline. The rest of the paper is structured as follows. In Section 2, we present related work. In Section 3, we describe the general framework of précis queries and introduce précis patterns. In Section 4, we describe our approach of answering queries using précis patterns and we sketch the techniques used for incremental population of a logical database subset and pre-storing answers in the system. Finally, in Section 5, we conclude 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. Motro [14] described the idea of using tokens, i.e. value of either data or metadata, when accessing information instead of structured queries, and proposed an interface that understands such utterances by interpreting them in a unique way, i.e. complete them to proper queries. With the advent of the World Wide Web, the idea has been revisited. In particular, recent approaches on keyword searches in databases [1, 2, 3, 6, 7, 11] extended the idea of tokens to values that may be part of attribute values. An answer to a keyword search is a set of ranked tuples. Oracle 9i Text [15], Microsoft SQL Server [12] and IBM DB2 Text Information Extender [9] create full text indexes on text attributes of relations and then perform keyword queries. Keyword search over XML databases has also attracted interest recently [4, 5, 8].

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 [10] 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 flatten out results returned by other approaches. Additionally, a complementary research effort provides a method towards the translation of a précis query answer into a narrative form, in order to return results such the one in the introduction about “Woody Allen” [19].

In this paper, we built upon the approach suggested in [10] and we revisit the idea of a logical database subset generated by a précis query by recognizing the existence of précis patterns, i.e. patterns of logical database subsets that capture semantics of different précis queries or preferences of different user groups and improve the efficiency of a précis query answering system.

3. The Précis Query Framework

3.1 Preliminaries

We consider the *database schema graph* $\mathbf{G(V, E)}$ as a directed graph corresponding to a database schema D. There are two types of nodes in $\mathbf{V}$:

- *relation nodes*, $\mathbf{R}$, one for each relation in the schema;
- *attribute nodes*, $\mathbf{A}$, one for each attribute of each relation in the schema.

Likewise, edges in $\mathbf{E}$ are the following:

- *projection edges*, Π, each one connects an attribute node with its container relation node, representing the possible projection of the attribute in the system’s answer;
- *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. For simplicity in presentation, we assume (a) that primary keys are not composite; thus, an attribute from a relation joins to an attribute from another relation, and (b) that these attributes have the same name. For convenience, we do not depict the joining attributes in both relations; instead, 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 $\mathbf{G(V, E)}$, where: $\mathbf{V} = \mathbf{R} \cup \mathbf{A}$, and $\mathbf{E} = \Pi \cup \mathbf{J}$. The notation for the graphical representation of a database schema graph is depicted in Fig. 1.

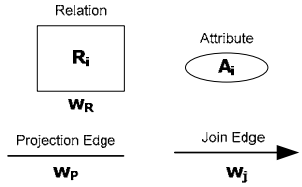


Fig. 1. Representation of graph elements

A *weight*, w, is assigned to each edge of the graph 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. A directed join edge expresses the dependence of the source relation of the join on the target one. The source relation indicates the relation already considered for the answer and the target corresponds to the relation that may be included, if the join is taken into account. For simplicity, we assume that there is at most one 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 weight of constituent edges. In principle, one may imagine several functions. All of them, however, should satisfy the condition that the weight decreases as the length of the path increases, based on human intuition and cognitive evidence [17].

Consider a database D properly annotated with a set of weights and a *précis query* Q, which is a set of tokens, i.e. $Q=\{k_1, k_2, \ldots, k_m\}$. 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* 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 (foreign-key) joins on D starting from *some* initial tuple, subject to the constraints in C [10]. Possible constraints in C could include the maximum number of attributes in D', the minimum weight of paths in the database schema graph, the maximum number of joins, the maximum number of

tuples in D' and so forth. Using different constraints and weights on the edges of the database schema allows generating different answers for the same query.

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, there is a major disadvantage: apart of the difficulty of browsing efficiently a database schema, per se, the user should spend some time with a procedure that does not seem relevant to his/her need for a certain answer. Weights and criteria may be pre-specified by a designer, or may be stored as part of a profile corresponding to a user or a group of users.

However, finding an appropriate set of weights to annotate a database is difficult as we explain below. Depending on users for providing appropriate weights for producing meaningful answers to précis queries is not acceptable, at least for the majority of them. Furthermore, weights may depend on the query and the user issuing the query, thus finding a unique set of weights for a database capturing different query semantics and user preferences altogether may not be possible. Finally, in the case of a system serving a large number of users, generating a logical subset from scratch each time a query is issued turns to be time consuming.

Therefore, in this paper, we propose a different approach. Patterns of logical subsets corresponding to different queries or groups of users may be recognized and stored in the system. For instance, different patterns would be used to capture preferences of movie reviewers and filmgoers.

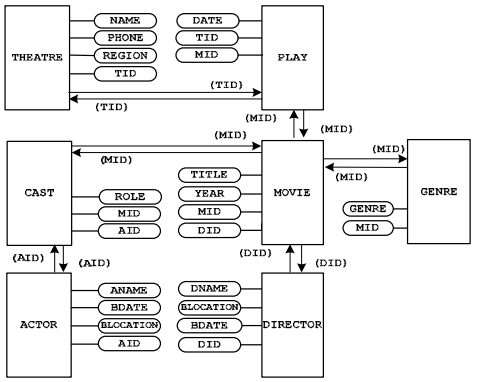
3.2 Précis Patterns

Formally, given the database schema graph G of a database D, a *précis pattern* is a directed rooted tree $\mathcal{P}(V,E)$ on top of G annotated with a set of weights. Given a query Q over database D, a précis pattern $\mathcal{P}(V,E)$ 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 pattern $\mathcal{P}$ is enriched with tuples derived from the database based on constraints in C. Possible constraints can be the maximum number of attributes, the maximum number of tuples, and so forth.




```
DIRECTOR(did,dname,blocation,
         bdate)
THEATRE(tid,name,phone,region)
PLAY (tid,mid,date),
GENRE(mid,genre)
MOVIE(mid,title,year,did)
CAST (mid,aid,role)
ACTOR(aid,aname,blocation,bdate)
```

Fig. 2. An example database graph

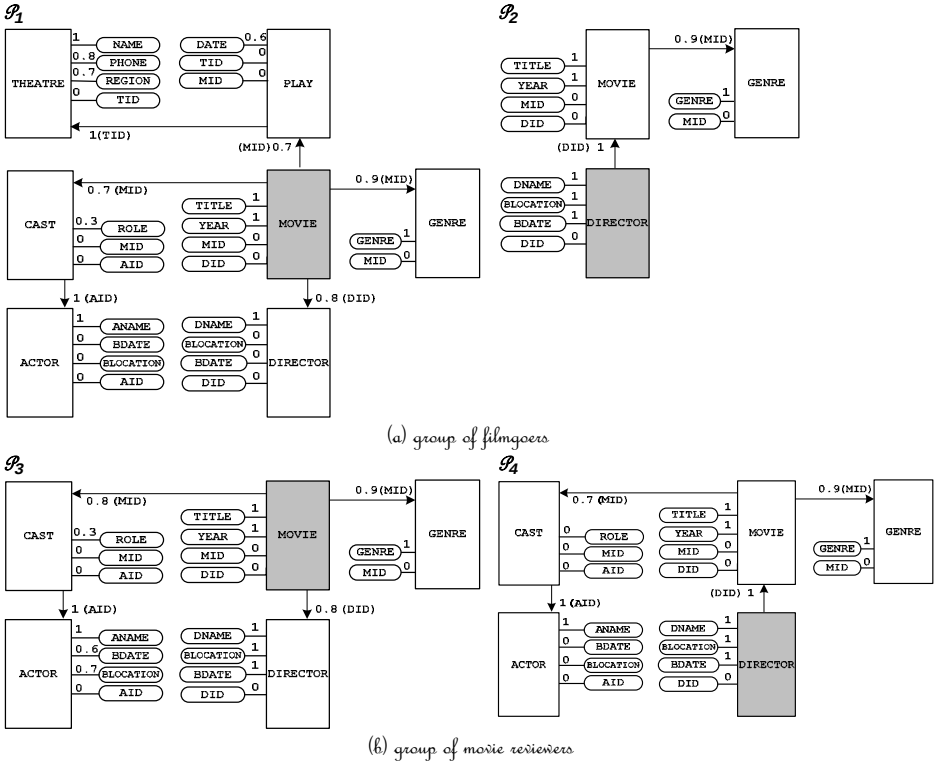


Fig. 3. Example précis patterns

3.3 An Example Database

Consider a movies database [16] described by the schema presented in Fig. 2; primary keys are underlined. The corresponding database graph is depicted in Fig. 2 too. On top of this graph, précis patterns may be recognized and stored in the system. Patterns

may correspond to different queries. In Fig. 3, $\mathcal{P}_1$ and $\mathcal{P}_2$ are patterns corresponding to different types of queries, i.e. regarding movies and directors, respectively (as indicated by the initial relations colored grey in each pattern). Different précis patterns may be also used to capture preferences of different groups of users. For instance, $\mathcal{P}_3$ and $\mathcal{P}_4$ are different patterns regarding movies and directors, respectively. $\mathcal{P}_1$ and $\mathcal{P}_2$ might capture preferences of filmgoers whereas $\mathcal{P}_3$ and $\mathcal{P}_4$ might correspond to movie reviewers. Assume that, $\mathcal{P}_1$ captures the fact that a filmgoer would be interested in information about theatres playing specific movies, while a movie-reviewer would not, as expressed in $\mathcal{P}_3$.

From the discussion above, it becomes apparent that there is an *n*-to-*m* correspondence between (group/user) profiles and patterns. As Fig. 4 shows, a pattern $\mathcal{P}_i$ may be used by more than one profile and a profile G_j may involve more than one pattern.

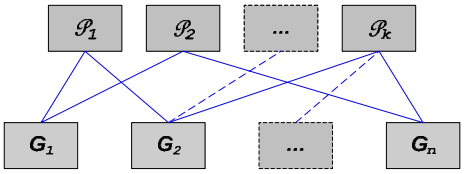


Fig. 4. Correspondence between patterns and profiles

Although an extensive analysis of précis patterns creation procedures is out of the scope of this paper, as an example, we refer two typical ways:

Manual creation. Pre-specified patterns may be created by a designer targeting different groups of users and different types of queries for a specific domain.

Semi-automatic creation. The system is trained using logs of queries that domain users have issued in the past. No matter how précis patterns are initially created, the system may adapt those associated with a specific user by learning from the queries this user submits to the system. In this way, the system may provide personalized answers to précis queries.

4. Answering Queries Using Précis Patterns

In this section, we describe a framework that generalizes the usage of précis queries based on patterns.

System Architecture. The system architecture of our approach is depicted in Fig. 5. 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 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 (Database Generator).

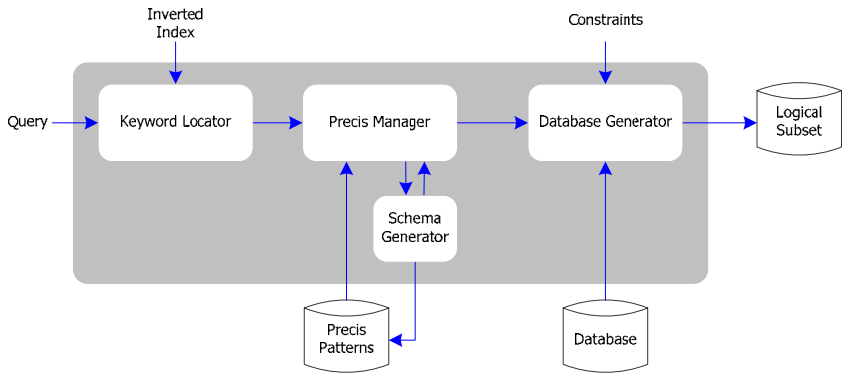


Fig. 5. System architecture

In more details, first, the user submits a précis query $Q=\{k_1, k_2, \ldots, k_m\}$. A set of constraints C may be additionally provided to determine tuples extracted from the database, in order to produce the logical database subset. The following steps are performed.

Keyword Locator. An inverted index 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). However, in our approach, we chose to create our own inverted index (technical details are out of the scope of this paper, but can be found in [18]), 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 tokens of other data types as well, such as date and number. Based on this inverted index, *Keyword Locator* 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, the following steps are not executed.)

Précis Manager. Next, instead of creating an ad-hoc logical subset for the particular query and user, *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.

Précis patterns are directed rooted trees $\mathcal{P}(V,E)$ that are stored in a graph database depicted as Précis Patterns in Fig. 5. An indexing mechanism $Index$ is needed for the search in the graph database. For this purpose, we adopt GraphGrep presented by Shasha et al. [20]. Recall that a précis pattern $\mathcal{P}(V,E)$ is applicable to Q, if its root relation coincides with an initial relation for Q. Thus, given the initial relations and a group of users, the index outputs the appropriate patterns. If none is returned for a certain initial relation, then the request is propagated to the *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 [18].) 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. Moreover, it is further propagated to the Database Generator through the Précis Manager module. The whole procedure is formally described by the algorithm EP depicted in Fig. 6.

Algorithm Extraction of a Précis Pattern (EP)
Input: a set of initial relations **R**, a group of users U, a set of stored patterns **P**
Output: a set of logical database subsets **D′**

```
Begin
  D′={};
  For each initial relation R∈R
    If (Index(R,U)!=null) {
      P = Index(R,U);
    else
      P = Schema_Generator(R,U);
      P = P ∪ P;
    }
    D′ = D′ ∪ P;
  End for
  Return D′;
End.
```

Fig. 6. The algorithm EP

For instance, a user belonging to the group of filmgoers of Fig. 3 issues the query "1960". Keyword Locator returns two initial relations, DIRECTOR and MOVIE, because this token is found in the field BDATE of the former and in the field YEAR of the latter. Then, Précis Manager identifies two applicable patterns, $\mathcal{P}_1$ and $\mathcal{P}_2$.

Database Generator. Subsequently, *Database Generator* enriches patterns with tuples extracted from the database. On each pattern, it starts from the initial relation where tokens in Q appear. Then, more tuples from other relations are retrieved by (foreign-key) join queries starting from the initial relation 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 this way, relations that are most related to a query are populated first. Any relations that may not be eventually populated due to constraints in C would be the ones most weakly connected to the query. 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.

Formally, given are a database D, a pattern $\mathcal{P}$, and optionally a set of constraints C (e.g., maximum total number of tuples, maximum number of tuples per relation and so forth). For the initial relation of $\mathcal{P}$, the list of tuples containing query tokens is considered. This is an initial logical database subset $D_\circ$ corresponding to pattern $\mathcal{P}$.

The set of possible logical database subsets corresponding to $\mathcal{P}$ in order of increasing cardinality is defined as follows:

$$D_1 \leftarrow D_o \bowtie R_1 \ , \ D_2 \leftarrow D_1 \bowtie R_2 \ , \ \ldots \ , \ D_{n_j} \leftarrow D_{n_{j-1}} \bowtie R_{n_j}$$

At any point, a relation R_i is joined to D_{i-1} if there is a join edge in $\mathcal{P}$ between this relation and a relation already populated in D_{i-1}. If more than one join may be executed, these are considered in order of decreasing weight. In this way, relations in D that are most related to the query are populated first. Any relations that may not be eventually populated due to the constraints would be the ones most weakly connected to the query. A logical database subset D_i contains all tuples also contained in D_{i-1} plus any tuples from D that join to those through the corresponding join. According to the constraints, the result database D' is a database D_c, such that:

$$c = \max(\{ t \mid t \in [0, n_J] : \text{constraints in } D_t \text{ hold} \})$$

For each relation R_i, a subset of its tuples, R_i', is found in the result D', projected on the set of attributes that are present in the result.

Optimization issues. Apart from the benefit of getting a pre-stored schema for a logical subset of a database instead of creating from scratch a new one, we further exploit the presence of précis patterns in our framework in a two-fold manner: (a) incremental population of a logical database subset, and (b) pre-stored answers to the most frequent précis queries.

Consider the following scenario: a user submits a query, and the system returns an answer, similar to the one presented in the introduction, in which certain keywords are hyperlinks. Clicking one of them fires a new query involving the corresponding keywords. The latter query is executed by the system and returns a new subset of information. The interesting problem is that this new query may specify results that have already been part of the initial system answer. We discriminate two possible cases: these results may be either presented to the user or not. However, in both cases, it would be desired to avoid re-computing them again.

For instance, assume that the initial query contained a keyword that identifies a director. Then, a possible system answer would contain, among others, a set of several movies, along with the names of their star actors, which could be transformed to hyperlinks. If the search continues with one of the actors, then the movies that he/she has participated in are a superset of the movies presented in the precedent answer. In such case our system incrementally populates the respective pattern for the new query.

Moreover, as practice shows, several keywords are more often posed than some others. According to this, we can keep track of the search history and maintain in the inverted index an extra attribute that stores for each keyword the frequency of its occurrences in queries submitted in the past. In our approach, we take into account the most frequently used keywords along with other parameters, such as the complexity of a pattern, in order to decide which patterns should be populated in advance. The threshold that determines which logical subsets should be populated is subject of further experimentation and tuning, inasmuch as the extent to which each database differs from another.

At this point, it is noteworthy to underline the difference between the notion of a précis pattern and the classical definition of a view. A view returns a single relation,

whereas a précis pattern represents the schema of a full-fledged database, which is the logical subset of another database, thus, containing multiple relations along with their relationships and constraints.

5. Conclusions

In this paper, we revisit the idea of a logical database subset generated by a précis query by recognizing the existence of précis patterns, i.e. patterns of logical database subsets that capture semantics of different précis queries or preferences of different user groups and improve the efficiency of a précis query answering system. In this context, each time a user poses a question, the system searches in a repository of précis patterns to extract an appropriate one. Then, this précis pattern is enriched with tuples extracted from the database according to the query keywords, in order to produce the logical database subset. Further optimization techniques are discussed.

Future work includes extension of the aforementioned methods toward the efficient capture and maintenance of précis patterns, the treatment of précis queries with complex semantics, e.g., involving multiple keywords as input combined with several operators, and the tuning of Database Generator. Another challenging issue is the extension of précis queries to provide ranked or top-k results.

6. 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.

7. 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] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pp. 564-575, 2004.

[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] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into xml query processing. *Computer Networks*, 33(1-6), 2000.

[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, pp. 16-27, 2003.

[6] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Effcient IR-style keyword search over relational databases. In *VLDB*, pp. 850-861, 2003.

[7] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pp. 670-681, 2002.

[8] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pp. 367-378, 2003.
[9] IBM. *DB2 Text Information Extender. url: www.ibm.com/software/data/db2/extenders/textinformation/*.
[10] G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. In *ICDE*, 2006.
[11] 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.
[12] Microsoft. *SQL Server 2000. url:msdn.microsoft.com/library/*.
[13] A. Motro. Baroque: A browser for relational databases. *ACM Trans. Inf. Syst.*, 4(2):164-181, 1986.
[14] A. Motro. Constructing queries from tokens. In *SIGMOD*, pp. 120-131, 1986.
[15] Oracle. *Oracle 9i Text. url: www.oracle.com/technology/products/text/index.html.*
[16] IMDB. Internet Movies DataBase. *url: www.imdb.com.*
[17] A. Collins and M. Quillian. Retrieval time from semantic memory. J. of Verbal Learning and Verbal Behaviour, 8:240-247, 1969.
[18] G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. TR-2006-1. *url: www.dblab.ntua.gr*.
[19] A. Simitsis and G. Koutrika. Comprehensible Answers to Précis Queries. In *CAiSE*, 2006.
[20] D. Shasha, J. Tsong-Li Wang, R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In PODS, pp. 39-52, 2002.