The VLDB Journal (2008) 17:117–149
DOI 10.1007/s00778-007-0075-9

SPECIAL ISSUE PAPER

Précis: from unstructured keywords as queries to structured databases as answers

Alkis Simitsis · Georgia Koutrika · Yannis Ioannidis

Received: 24 October 2006 / Revised: 20 September 2007 / Accepted: 21 September 2007 / Published online: 9 November 2007

Abstract Précis queries represent a novel way of accessing data, which combines ideas and techniques from the fields of databases and information retrieval. They are free-form, keyword-based, queries on top of relational databases that generate entire multi-relation databases, which are logical subsets of the original ones. A logical subset contains not only items directly related to the given query keywords but also items implicitly related to them in various ways, with the purpose of providing to the user much greater insight into the original data. In this paper, we lay the foundations for the concept of logical database subsets that are generated from précis queries under a generalized perspective that removes several restrictions of previous work. In particular, we extend the semantics of précis queries considering that they may contain multiple terms combined through the *AND*, *OR*, and *NOT* operators. On the basis of these extended semantics, we define the concept of a logical database subset, we identify the one that is most relevant to a given query, and we provide algorithms for its generation. Finally, we present an extensive set of experimental results that demonstrate the efficiency and benefits of our approach.

Keywords Keyword search · Free-from queries · Query processing

A. Simitsis
National Technical University of Athens,
15772 Athens, Greece
e-mail: asimi@dblab.ece.ntua.gr

Present address:
A. Simitsis
IBM Almaden Research Center,
San Jose, CA, USA

G. Koutrika (✉) · Y. Ioannidis
University of Athens, 15784 Athens, Greece
e-mail: koutrika@di.uoa.gr

Y. Ioannidis
e-mail: yannis@di.uoa.gr

Present address:
G. Koutrika
Stanford University, Stanford, CA, USA

1 Introduction

Emergence of the World Wide Web has opened up the opportunity for electronic information access to a growing number of people. This, however, has not come without problems. A large fraction of content available through the web resides in (semi-)structured databases. On the other hand, web users do not, and should not, have any knowledge about data models, schemas, query languages, or even the schema of a particular web collection to form their own queries. In addition, they often have very vague information needs or know a few buzzwords. But, at the same time, they "want to achieve their goals with a minimum of cognitive load and a maximum of enjoyment. . . . humans seek the path of least cognitive resistance and prefer recognition tasks to recall tasks" [42]. Hence, the need to bridge the gap between the average user's free-form perception of the world and the underlying systems' (semi-) structured representation of the world becomes increasingly more important.

In this direction, current commercial and research efforts have focused on *free-form queries* [25,40,59] as an alternative way for allowing users to formulate their information needs over structured data. Recently, *précis* was introduced as an approach tackling both query and answer formulation over structured data [38]. The term "précis " is defined as follows:

"**précis** /ˈpreɪsiˑ▲▼: [(of)] a shortened form of a piece of writing or of what someone has said, giving only the main points." (Longman Dictionary)

A précis is often what one expects in order to satisfy an information need expressed as a question or as a starting point toward that direction. For example, if one asks about "Woody Allen", a possible response 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)."

Likewise, returning a précis of information in response to a user query is extremely valuable in the context of web accessible databases.

Based on the above, a *précis query* is an unstructured, keyword-based, query that generates a structured answer. In particular, it generates an entire multi-relation database, instead of the typical individual relation that is output by current approaches for (un)structured queries over structured data. This database is a *logical subset* of the original one, i.e., it contains not only items directly related to the given query terms but also items implicitly related to them in various ways. It provides to the user greater insight into the original data and may be used for several purposes, beyond intelligent query answering, ranging from data extraction to information discovery. For example, enterprises often need subsets of their regular, large, databases that, nevertheless, conform to the original schemas and satisfy all constraints, so that they may perform realistic tests of new applications before deploying them to production. A logical database subset would be also useful as an answer to a structured query. However, combining unstructured queries with structured answers, the advantages of two worlds are combined: ease of query formulation based on keywords [58] and richness of information conveyed by structured answers.

Receiving a database as a query answer has many advantages. Assuming that the original schema was normalized, the database-answer remains normalized as well, thus avoiding any redundancy and capturing the original semantic and structural relationships that exist between objects/tuples. Furthermore, having a database as an answer permits multiple queries to be asked on it for further refinement, reviewing, or restructuring of the data; doing so on a single, unnormalized table would be either impossible or at least unnatural. In fact, this is just one example of a more general benefit related to the reuse of the data answer. This database can be shared by multiple users or can be integrated with other databases in a federation. The rationale of returning a subset of the initial schema as an answer can be found in other areas of computer science as well. In several languages that have been developed for semantically richer data models (object-oriented models, XML, etc.), the answer to a query may be an elaborate construct that includes different kinds of objects appropriately interlinked. This paper transfers and adapts this philosophy to the relational world.

Earlier work has been restricted to précis queries with a single keyword, which was searched for in all attributes of all database relations. In this work, we present a generalized framework for précis queries that contain multiple terms combined with the logical operators *AND*, *OR*, and *NOT*. Supporting such generalized précis queries using a relational database system is not straightforward and requires a sequence of operations to be performed. First, the possible interpretations of a précis query need to be found by taking into account (a) the database schema graph, (b) the query semantics, and (c) the database relations that contain the query terms. For example, given the query "Clint Eastwood" AND "thriller", its possible interpretations include those that refer to thrillers where Clint Eastwood is an actor, thrillers directed by Clint Eastwood, and so forth. This is achieved by finding initial subgraphs of the schema graph, each one corresponding to a different query interpretation. Second, these interpretations need to be enriched in order to identify other information that is also implicitly related to the query, e.g., actors acting in thrillers directed by Clint Eastwood. This corresponds to expanding the initial subgraphs up to a point where prespecified constraints are satisfied, so that the schema of the logical database subset desired is formed. Third, the logical subset needs to be populated. At this stage, a set of queries is dynamically constructed and these generate a whole new database, with its own schema, constraints, and contents, derived from their counterparts in the original database.

Contributions In accordance to the above stages of answer formulation for précis queries, the contributions of this paper are the following:

– We extend précis queries allowing them to (a) contain multiple terms and (b) have those terms be combined using the logical operators *AND*, *OR*, and *NOT*. For instance, a précis query on a movie database may be formed as:

(*"Clint Eastwood" AND "thriller"*) *OR*
(*"Gregory Peck" AND NOT "drama"*)

– We formally define the answer to a generalized précis query as a logical database subset. The constraints that determine the logical database subset may be purely syntactic, e.g., a bound on the number of relations in it, or may be more semantic, capturing its relevance to the query,

e.g., a bound on the strength of connection between the relations containing the query terms and other relations. Such strength is captured by weights on the database schema graph, offering great adjustability and flexibility of the overall framework.
- We provide novel and efficient algorithms for the creation of the logical database subset schema and its population. We tackle the problem of finding appropriate initial subgraphs for a given précis query under a set of constraints. We introduce an algorithm that expands initial subgraphs in order to construct the schema of the logical database subset. We prove the completeness and correctness of our methods.
- We present an extensive set of experimental results that: (a) demonstrate how the logical database subset characteristics are affected by variations in the query characteristics, the constraints and the weights used; (b) evaluate the algorithms proposed and exhibit their efficiency both in absolute terms and in comparison to earlier work; (c) provide insight into the effectiveness of our approach from the users' point of view.

Outline The paper is structured as follows. Section 2 discusses related work. Section 3 presents the general framework of our approach, including a database graph model, and the formal definition of a logical database subset. Section 4 provides the system architecture of our approach. Sections 5 and 6 present algorithms for the construction of a logical subset schema and its population, respectively. Section 7 presents experimental results. Section 8 provides a discussion on the overall evaluation of our approach. Finally, Sect. 9 concludes our work with a prospect to the future.

2 Related work

In this section, we present the state of the art concerning keyword search in the context of databases and in contrast to Information Retrieval approaches.

2.1 Keyword search in databases

The need for free-form queries has been early recognized in the context of databases. Motro described the idea of using tokens, i.e., values of data or metadata, instead of structured queries, for accessing information, and proposed an interface that understands such utterances by interpreting them in a unique way, i.e., completing them to proper queries [47]. In the same context, BAROQUE [46] uses a network representation of a database and defines several types of relationships in order to support functions that scan this network.

With the advent of the World Wide Web, the idea has been revisited. Search engines allow users to perform keyword searches. However, a great amount of information is stored in databases and cannot be indexed by search engines [18,39]. Therefore, the need for supporting keyword searching over databases as well is growing. Recent approaches extended the idea of tokens to values that may be part of attribute values [59]. Several systems have been proposed, including BANKS [5,29,32], DISCOVER [25,27], DBXplorer [2], ObjectRank [4], and RSQL/Java [43]. A different line of research focuses on search effectiveness rather than efficiency [40]. The notion of proximity in searches is also discussed in some approaches [2,20,25,27]. For instance, a query language for performing proximity searches in a database has been proposed, which supports two kinds of queries: "Find" that is used to specify the set of objects that are potentially of interest, and "Near" that is used to rank the objects found in the Find set [20].

Keyword search over XML databases has also attracted interest [17,21,23,26,28]. Recent work includes extending XML query languages to enable keyword search at the granularity of XML elements [17], ranking functions for XML result trees [23], construction and presentation of XML minimal trees [28], keyword proximity queries on labeled trees [26], and concept-aware and link-aware querying that considers the implicit structure and context of Web pages [21].

Several commercial efforts also exist. For instance, the major RDBMS vendors present solutions that create full text indexes on text attributes of relations in order to make them searchable: IBM DB2 Text Information Extender [31,41], Microsoft SQL Server 2000 [24,45], MySQL [48], and Oracle 9i Text [15,49]. However, they do not support truly free-form queries but SQL queries that specify the attribute name, in which a keyword will be searched for.

Précis queries are defined for relational databases. Therefore, subsequently, we elaborate more on the most related approaches for keyword search in relational databases [2,4,5,25,27,32] and we accent the key differences of our work.

2.1.1 Keyword search in relational databases

Keyword search approaches in relational databases fall into two broad categories: schema-level and tuple-level approaches.

Schema-level approaches Schema-level approaches model the database schema as a graph, in which nodes map to database relations and edges represent relationships, such as primary key dependencies [2,25,27]. Based on this graph, the generation of an answer involves two generic phases. The first one takes as inputs the relations that contain the query keywords and the graph and builds the schema of each possible (or permissible, if constraints are given) answer. If there

are more than one answer for a query, this phase tries to find the schema for each one of them. The second phase generates appropriate queries that retrieve the actual tuples from the database following the schema of each answer.

In DBXplorer [2], the database schema graph is undirected. The first phase of the answer generation process finds join trees (instead of subgraphs, for simplicity) that interconnect database tables containing the query keywords. The leaves of those trees should be tables containing keywords. The algorithm used to identify the join trees first prunes all sink nodes of the schema graph that do not contain any keywords. The result is a subgraph guaranteed to contain all candidate join trees. Then, qualifying sub-trees are built based on a breadth-first enumeration on this subgraph. To improve the performance, a heuristic is used: The first node of a candidate qualifying sub-tree is the one that contains a keyword that occurs in the fewest tables. The population of the qualifying join trees is straightforward. For each join tree a respective SQL query is created and executed. Results are ranked based on the number of joins executed.

DISCOVER [27] also models a relational database as an undirected schema graph. It uses the concept of a candidate network to refer to the schema of a possible answer, which is a tree interconnecting the sets of tuples (i.e., relations) that contain all the keywords, as in DBXplorer. The candidate network generation algorithm is also similar, i.e., it is based on a breadth-first traversal of the schema graph starting from the relations that contain the keywords. Also, the algorithm prunes 'dead-ends' appearing during its execution. The candidate networks are constructed in order of increasing size; smaller networks, containing fewer joins, are preferred. For the retrieval of the actual tuples, DISCOVER creates an appropriate execution plan that uses intermediate results to avoid re-executing joins that are common among candidate networks. For this purpose, a greedy algorithm is used that essentially creates a new query per each candidate network taking into consideration the possible avoidance of some joins that are used more than once in the whole scenario. Results are ranked based on the number of joins of the corresponding candidate network.

In continuation of this work, DISCOVER considered tuple retrieval strategies for IR-style answer-relevance ranking [25]. The authors adapted their earlier naïve algorithm so that it issues a top-k SQL query for each candidate network. They also proposed three more algorithms for retrieving the top-k results for a keyword query. Their Sparse algorithm is essentially an enhanced version of the naïve one that does not execute queries for non promising candidate networks, i.e., those that do not produce top-k results. The Global Pipelined algorithm progressively evaluates a small prefix of each candidate network in order to retrieve only the top-k results, and hence, it is more efficient for queries with a relatively large number of results. The Sparse algorithm is more efficient for queries with few results because it can exploit the highly optimized execution plans that the underlying RDBMS can produce when a single SQL query is issued for each candidate network. On the basis of the above, a Hybrid algorithm has been also described that tries to estimate the expected number of results for a query and chooses the best algorithm accordingly.

In comparison with our work, we note the following:

- DBXplorer and DISCOVER use undirected graphs, while we model a database schema as a directed weighted graph. Directionality is natural in many applications. To capture that, we consider that the strength of connections between two relation nodes is not necessarily symmetric. For instance, following a foreign/primary key relationship in the direction of the primary key may have different significance than following it in the opposite direction.
- Following a schema-level approach, our query answering operates in two phases. In order to find the schema of the possible answers for a query, our logical subset creation algorithm (described in Sect. 5) first extracts initial subgraphs from the schema graph, each one corresponding to a different query interpretation. Then, these interpretations are enriched in order to discover other information that may be implicitly related to the query. This corresponds to expanding the initial subgraphs based on a set of constraints. The creation of the initial subgraphs resembles candidate network generation. However, based on the précis semantics (Sect. 3.4), we are interested in generating initial subgraphs in decreasing order of significance as captured by the graph weights. Therefore, our corresponding algorithm (*FIS*) is based on a best-first traversal of the database schema graph, which is more efficient for generating subgraphs in order of weight compared with the breadth-first traversal methods for candidate network generation used in [2,25,27].
- Regarding the generation of results, our naïve algorithm (*NaiveLSP*) is an adaptation of the population algorithms used in [2,27] and the naïve of [25], i.e., it executes one query per initial subgraph, but then it has to split results among the relations contained in the logical subset. On the other hand, we do not perform tuple ranking, therefore in order to compare ourselves to the top-k ranking algorithms used in [25], we can consider that all tuples have the same weight, i.e., equal to 1, and that the algorithms return all matching tuples, not just the top-k ones. In this case, the Sparse algorithm coincides with the naïve, while the Global Pipelined algorithm is more time-consuming than the naïve, because it has to fully evaluate all candidate networks. Instead of exploiting the highly optimized execution plans that the underlying RDBMS can produce when a single SQL query is issued per candidate network, the Global Pipelined uses nested-loops joins.

– Finally, for ranking, DBXplorer and the earlier version of DISCOVER use the number of edges in a tree as a measure of its quality, preferring trees with fewer edges, while we rank answers based on the weight of the corresponding schema sub-graph. IR-style ranking techniques, such as those presented in [25], can be incorporated in our approach for ranking results of précis queries.

Tuple-level approaches Tuple-level approaches model the database as a data graph, in which nodes map to tuples and edges represent relationships between tuples, such as primary key dependencies [4,5,32]. These approaches involve one phase for the answer generation, where the answer schema extraction and the tuple retrieval tasks are interleaved: the system walks on the data graph trying to build trees of joining tuples that meet the query specification, e.g., they involve all keywords for queries with AND semantics.

BANKS [5] views the data graph as a directed weighted graph and performs a Backward Expanding search strategy. It constructs paths starting from each relation containing a query keyword and executing a Dijkstra's single source shortest path algorithm for each one of them. The idea is to find a common vertex from which a forward path exists to at least one tuple corresponding to a different keyword in the query. Such paths will define a rooted directed tree with the common vertex as the root containing the keyword nodes as leaves, which will be one possible answer for a given query.

Due to the fact that they have to traverse the data graph, which is orders of magnitude larger than the database schema graph, and that they execute a great number of joins in order to connect tuples containing the keywords, tuple-level techniques may present scalability issues. For example, the Backward Expanding search performs poorly in case a query keyword matches a very large number of tuple nodes. For these reasons, the Bi-directional Expansion strategy explores the data graph in a way that nodes that join to many other nodes or point to nodes that serve as hubs are avoided or visited later [32]. Hence, it reduces the size of the search space at the potential cost of changing slightly the answer.

ObjectRank [4] uses a different answer model, making it incomparable to all afore-mentioned approaches, including ours. ObjectRank views the database as a labeled directed graph, where every node has a label showing the type of node and a set of keywords that comprise the content of the node. Every node has a weight that represents the node's authority, i.e., its query specific ObjectRank value, determined using a biased version of the Pagerank random walk [6]. Edges carry weights that denote the portion of authority that can flow between two nodes. Also, their evaluation algorithm requires expensive computations making the whole query execution expensive.

Although the tuple-level approaches are not directly comparable with the schema-level ones, at a higher level, all approaches to keyword search on relational databases deal with the same kind of problem: Given some kind of a graph and a set of nodes on it, they try to build one or more trees connecting these nodes and containing a combination of keywords. Hence, they all deal with some variant of the Steiner tree problem [30]. On the basis of this abstraction, we could compare tuple-level and schema-level algorithms. However, this abstraction cannot be applied to Bi-directional Expansion [32], because, this algorithm, in effect, chooses a join order dynamically and incrementally on a per-tuple basis. Therefore, it works only on a data graph. Thus, in what follows, we compare our algorithm for the creation of initial subgraphs (*FIS*) only to the Backward Expanding strategy [5].

– Given a graph and a set of nodes, each method seeks a different solution, due to the different query semantics defined in each approach. We seek solutions that are directed subgraphs, in which the root and the sink nodes belong to the given set of nodes. BANKS builds directed trees, in which the leaves are given but the root could be any relation [5,32].
– Apart from the graph and the set of nodes, *FIS* takes also as input a set of constraints that describe which solutions are desired, in terms of relevance or structure.
– The *FIS* algorithm and the Backward Expanding strategy perform a best-first traversal of the graph starting from every node in this set. However, due to the fact that they assume a different answer model as explained above, they proceed in a different way. *FIS* considers each of the given nodes as a root of a possible solution and traverses the graph to find other nodes that contain keywords, while the Backward Expanding strategy considers each given node as a leaf and traverses the graph edges in reverse direction in order to find a node that connects all given nodes, and this node is the root of the solution tree. Hence, the former progressively builds subgraphs in decreasing order of weight, whereas the latter builds solutions that are not ordered.

Furthermore, a high level comparison of our approach to both tuple-level and schema-level approaches to keyword search in relational databases reveals the following differences:

– *Query language*. A keyword query is a set of keywords with *AND* [2,4,5,27,32] or *AND*/*OR* semantics [25]. In contrast, our framework provides a richer query language allowing the formulation of a précis query as a combination of keyword terms with the use of the logical operators *AND* ($\bigwedge$), *OR* ($\bigvee$), and *NOT* ($\neg$).
– *Answer model*. Existing approaches to keyword searching focus on finding and possibly interconnecting, depending on the query semantics, tuples in relations that contain

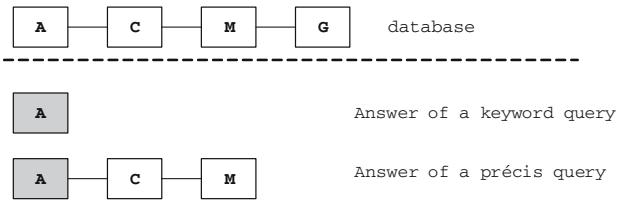


Fig. 1 Comparison of précis answers to typical keyword answers

the query terms. On the other hand, précis queries additionally consider information found in other parts of the database that may be related to the query. For example, consider the database depicted in Fig. 1, which stores information about {A}ctors, {M}ovies, {C}ast and movie {G}enres, and a query about "Julia Roberts." The answer provided by existing approaches would essentially contain a tuple from the {A}ctors relation for the actress Julia Roberts, whereas the answer returned based on the précis framework may also return movies starring this actress. This difference in the two possible answers is depicted in Fig. 1: the upper case illustrates the schema of the first answer whereas the lower case depicts the schema of the précis answer. In particular, in the précis framework, we use a generalized answer model, which considers that a possible answer to a précis query can be viewed as a graph of joining tuples, where leaves do not necessarily contain query keywords, in contrast to all other approaches that consider joining trees of tuples, where leaves must contain query keywords.

– *Answer format*. A précis query generates an entire multi-relation database, instead of the typical individual relation containing flattened out results returned by other keyword search approaches.
– *Customization of answers*. Given a précis query, a set of constraints can be used to shape and customize the final answer in terms of its schema and tuples. These constraints may be purely syntactic, e.g., a bound on the number of relations in it, or more semantic, capturing its relevance to the query, e.g., a bound on the strength of connections between the relations containing the query terms and the remaining ones captured by weights on the database schema graph. Thus, the précis framework offers great adjustability and flexibility, because it allows tailoring answers based on their relevance or structure. Only DISCOVER considers the maximum size of an answer in terms of joins, which is a structural constraint.

Earlier work on précis queries [38] assumes that they contain only one term possibly found in many different relations. If the query contains more than one term, these are treated as one (phrase). In this article, we extend query semantics in two directions by considering (a) multi-term queries and (b) that terms in a query may be words or phrases that may be combined using the logical operators *AND*, *OR*, and *NOT*. A brief presentation of these ideas exists in a poster paper [55], where we discuss the issue of the generalization of precis queries using multiple keywords. In this article, we formally present the problem to its full extent, define the answer to a précis query (under the extended semantics) as a logical database subset, and describe new algorithms for creation of a logical database subset schema and for its population. An extensive evaluation of the algorithms with respect to several parameters, such as the query characteristics and the given constraints, and in comparison to earlier work shows their overall effectiveness and efficiency.

2.1.2 Ranking of query answers

One advantage of tuple-based techniques for keyword search over schema-level ones is the possibility of fine-grained tuple ranking. In the directed graph model of BANKS [5,32], an answer to a keyword query is a minimal rooted directed tree, embedded in the data graph, and containing at least one node from each initial relation. Its overall score is defined by (a) a specification of the overall edge score of the tree based on individual edge weights, (b) a specification of the overall node score of the tree, obtained by combining individual node scores, and finally, (c) a specification for combining the tree edge score with the tree node score. In our case, answers are ranked according to their relevance, which is determined based on the weights at the schema level, but it can be extended to support finer-grained ranking. For instance, it could support the IR-style answer-relevance ranking described in [25]. There is a large number of work in ranking answers to database queries, apart from the work that has been done for keyword queries, that could potentially be adapted to our framework.

For instance, a ranking method based on query workload analysis has been presented [3]. It proposes query processing techniques for supporting ranking and discusses issues such as how the relational engine and indexes/materialized views can be leveraged for query performance. In the same context, two efforts have been proposed: the ranked join index that efficiently ranks the results of joining multiple tables [57] and a technique for the processing of ranked queries based on multidimensional access methods and branch-and-bound search [56]. In general, a number of research approaches investigate the problem of ranking answers to a database query from several points of view: relational link-based ranking [19], context-sensitive ranking [1], probabilistic ranking [9]; recently, the ordering of attributes in query results has been also proposed [14]. Although, in general, this is a problem with high computational complexity, recent approaches overcome the exponential runtime required for the computation of top-k answers and prove that the answer of keyword

proximity search can be enumerated in ranked order with polynomial delay [33–35].

2.2 Information retrieval and databases

Information retrieval systems rank documents in a collection so that the ones that are more relevant to a query appear first. Models that have been used in IR for this purpose include the Boolean, Vector Processing, and Probabilistic models, drawing upon disciplines such as logical inference, statistics, and set theory. The Boolean model is well understood. However, even though it is very appropriate for searching in the precise world of databases, it has inherent limitations that lessen its attractiveness for text searching [11]. The essential idea of the vector processing model is that terms in documents are regarded as the coordinates of a multi-dimensional information space. Documents and queries are represented by vectors in which the ith element denotes the value of the ith term, with the precise value of each such element being determined by the particular term weighting scheme that is being employed. The complete set of values in a vector hence describes the position of the document or query in this space, and the similarity between a document and a query is then calculated by comparing their vectors using a similarity measure such as the cosine [51].

Probabilistic IR formally ranks documents for a query in decreasing order of the probability that a document is useful to the user who submitted the request, where the probabilities are estimated as accurately as possible, on the basis of whatever data has been made available to the system for this purpose [7,12,22,50]. For instance, based on a set of assumptions regarding the relevance of a document and the usefulness of a relevant document, the application of this principle optimizes performance parameters that are very closely related to traditional measures of retrieval effectiveness, such as recall and precision [50]. Adapting techniques of exploratory data analysis to the problem of document ranking based on observed statistical regularities of retrieval situations has been also proposed [22].

Keyword search approaches in databases, including ours, assume the Boolean model, which is a well understood model for structured queries in databases. Hence, assuming the same model for unstructured queries as well is natural. On the other hand, keyword search in databases is a relatively new research area. Exploring other models inspired from IR, is an open research issue. As our work matures, we are looking into such models for précis queries.

3 Précis framework

In this section, we introduce our framework for the study of précis queries. We present the data model we adopt, we provide a language for précis query formulation and we prescribe the answer to such a query as a logical database subset.

3.1 Data model

We focus on databases that follow the relational model, which is enriched with some additional features that are important to the problem of concern. The main concepts of the data model assumed are defined below.

A *relation schema* is denoted as $\mathbf{R_i}(A^i_1, A^i_2, \ldots, A^i_{k^i})$ and consists of a relation name $\mathbf{R_i}$ and a set of attributes $\mathbf{A^i} = \{A^i_j : 1 \leq j \leq k^i\}$. A *database schema* $\mathbf{D}$ is a set of relation schemas $\{\mathbf{R_i} : 1 \leq i \leq m\}$. When populated with data, relation and database schemas generate *relations* and *databases*, respectively. We use R_i to denote a relation following relation schema $\mathbf{R_i}$ and D to denote a database following database schema $\mathbf{D}$. The members of a relation are tuples.

Given a relational database D, a *logical database subset* L of D has the following properties:

- The set of relation names in the schema $\mathbf{L}$ of the logical database subset is a subset of that in the original database schema $\mathbf{D}$.
- For each relation $\mathbf{R_i}$ in $\mathbf{L}$, its set of attributes $\mathbf{A^{i'}} = \{A^{i\,'}_j : 1 \leq j \leq l_i\}$ in $\mathbf{L}$ is a subset of its set of attributes $\mathbf{A^i} = \{A^i_j : 1 \leq j \leq k_i\}$ in $\mathbf{D}$ ($l_i \leq k_i$). In other words, $\mathbf{L}$ involves some of the attributes of each relation schema present.
- For each relation $\mathbf{R_i}$ in $\mathbf{L}$, its set of tuples is a subset, R'_i, of the set of tuples in the original relation R_i (projected on the set of attributes $\mathbf{A^{i'}}$ that are present in the result).

A *Database schema graph* $\mathbf{G}(\mathbf{V}, \mathbf{E})$ is a directed graph corresponding to a database schema $\mathbf{D}$. 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: (a) projection edges, $\mathbf{\Pi}$, one for each attribute node, emanating from its container relation node and ending at the attribute node, representing the possible projection of the attribute in a query answer; and (b) join edges, $\mathbf{J}$, emanating from a relation node and ending at another relation node, representing a potential join between these relations. Therefore, a database schema graph is a directed graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$, where $\mathbf{V} = \mathbf{R} \cup \mathbf{A}$ and $\mathbf{E} = \mathbf{\Pi} \cup \mathbf{J}$. Its pictorial representation uses the notation in Fig. 2. Since projection edges are always from relation nodes to attribute nodes, they are typically indicated without their direction, as this is easily inferred by the types of the nodes on their two ends.

Given two nodes $v_s, v_e \in \mathbf{V}$, an edge from v_s to v_e is denoted by $e(v_s, v_e)$ or $e_j(v_s, v_e)$, if a set of edges is discussed, enumerated by some integer index j. Nodes v_s, v_e

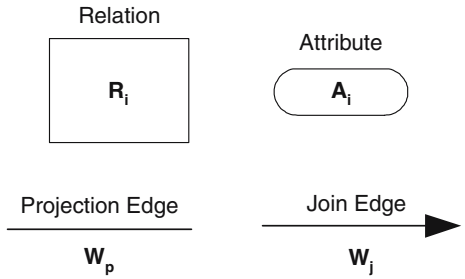


Fig. 2 Representation of graph elements

may be omitted if they are easily understood or are not critical. A *path* from a relation node v_s to an arbitrary node v_e, denoted by $p(v_s, v_e)$ or simply p, consists of a sequence of edges, the first one of which emanates from v_s and the last one of which, possibly a projection edge, ends at v_e. Notationally, this implies that $p(v_s, v_e) = e(v_s, x_1) \cup e(x_1, x_2) \cup \cdots \cup e(x_k, v_e)$, where $v_s, v_e, x_j (1 \leq j \leq k) \in \mathbf{V}$. In that case, v_e is said to be *reachable* from v_s. Likewise, if v_s and v_e are both relation nodes, then for tuples $t_s \in v_s$ and $t_e \in v_e$, if there are tuples in relation nodes $x_j (1 \leq j \leq k)$ such that t_s joins with x_1, x_j joins with x_{j+1}, $(1 \leq j < k)$, and x_k joins with t_e, then t_e is said to be *reachable* from t_s. Each node (and each tuple) is reachable from itself.

As edges represent explicit relationships between nodes in the graph, directed paths on the graph represent "implicit" relationships. In particular, a path between two relation nodes, comprising adjacent join edges, represents the "implicit" join between these relations. Similarly, a path from a relation node to an attribute node, comprising a set of adjacent join edges and ending with a projection edge, represents the "implicit" projection of the attribute on this relation.

3.2 Schema weight annotation

Each edge e of a graph $\mathbf{G}$ is assigned a weight $w_e \in [0, 1]$. Naturally, the weights of two edges between the same nodes but in opposite directions could very well be different. For instance, following a foreign/primary key relationship in the direction of the primary key may have different significance than following it in the opposite direction. The weight w_p of a path $p = e_0 \cup e_1 \cup \cdots \cup e_k$ is a function of the weights of its constituent edges and should satisfy the following condition:

$$w_p \leq \min_{0 \leq j \leq k} \; w_{e_j} \tag{1}$$

This condition captures the intuition that the weight should decrease as the length of the path increases [10]. In our implementation, we have chosen multiplication as such function.

Syntactically, weights represent the significance of the association between the nodes connected and determine accordingly whether or not the corresponding schema construct will be inserted into the query to modify its final answer. For instance, $w_e = 1$ expresses strong relationship: if the source node appears in the query, then the end node should appear as well. If $w_e = 0$, occurrence of the source node does not imply occurrence of the end node. Semantically, weights express notions of relevance, importance, similarity, or preference. Hence, they may be captured in one of several ways. They may be automatically computed taking into account graph features, such as connectivity (e.g., in the spirit of [36]). They may be also specified by a designer or mined from query logs to capture query patterns that are relevant to different (groups of) users [54].

For example, for ad hoc queries, weights may be set by the user (e.g., a designer or an administrator) at query time using an appropriate user interface. This option enables interactive exploration of the contents of a database and the weights do not necessarily have any particular meaning as only their relative ordering is important: by emphasizing or de-emphasizing the strength of connections between database relations at will, the user essentially poses different sets of queries, thereby exploring different regions of the database and generating different results.

Query logs contain query sessions from different users, hence they can be used as a valuable resource for relevance feedback data [8,13]. To illustrate how query logs could be used for computing weights, we consider the following simple example. Given a database and a set of queries issued over it, the weight of a projection edge between a relation and an attribute could be computed as the number of times this attribute is projected in these queries versus the total number of projected attributes in all queries. Likewise, the weight of a join edge from a relation R_i to a relation R_j could be given as the number of co-occurrences of both relations in a single query divided by the total number of R_i's occurrences in the log. In this case, the weight would be a reflection of the importance of an edge for the users posing these queries, which could then be taken into account for précis queries. Clearly, more sophisticated weight computation schemes exist: for instance, since certain queries are elaborations of previous ones, we could take such dependencies into account when computing join-edge weights and capture more accurately the importance of each join as users reformulate their queries to express their information needs.

The algorithms presented in this paper work on arbitrary directed weighted graphs and are not affected by how the weights are defined or on the exact technique used for computing them. Hence, any further discussion on weight computation is beyond the scope of this paper.

Example Consider a movie database described by the following schema; primary keys are underlined.

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

In the following sections, we will interchangeably use the full names of relations and their initial letter to refer to them, e.g., in case of figures for increasing their readability.

A database schema graph corresponding to this database is shown in Fig. 3. Weights on the graph edges capture the significance of the association between the nodes connected. For instance, for *THEATRE*, attribute *NAME* is more significant than *REGION* or *PHONE*, whereas for *MOVIE*, the most significant attributes are *TITLE* and *YEAR*. Also, for movies, information about the movie genre is more important than information about the movie director. Thus, the weight of the edge from *MOVIE* to *GENRE* is greater than the weight of the edge from *MOVIE* to *DIRECTOR*. In general, two relations may be connected with two join edges on the same set of joining attributes in the two possible directions carrying different weights. For instance, the weight of the edge from *GENRE* to *MOVIE* is set to 0.7, and the weight of the edge from *MOVIE* to *GENRE* equals to 1. This implies that movies of a particular genre are not so important for the genre itself, while the genre of a particular movie is very important for the movie.

Using multiplication for composition of edge weights along the path from *MOVIE* to *ACTOR*, the weight of the path is $0.7 * 1.0 = 0.7$. Thus, for a movie, information about participating actors is less significant than information about the movie's genre, whose (explicit) weight is 1.0. Similarly, the weight of the projection of a movie's actors' birth dates, represented by the path from *MOVIE* to $BDATE$, equals $0.7 * 1 * 0.6 = 0.42$.

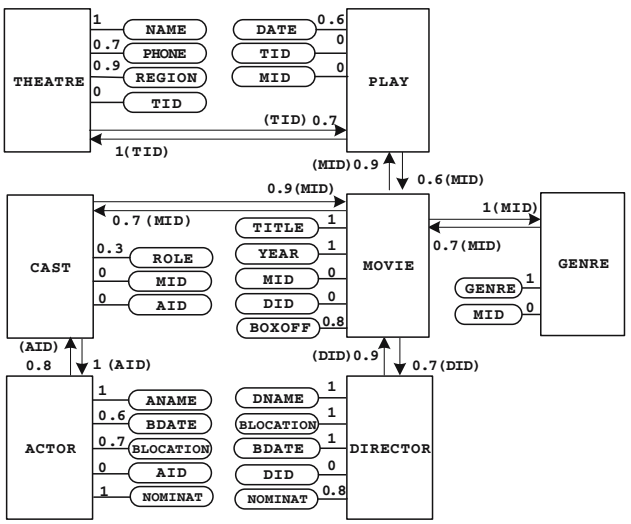


Fig. 3 An example database schema graph

3.3 Précis query language

A *précis query* is formulated as a combination of keyword terms with the use of the logical operators *AND* ($\bigwedge$), *OR* ($\bigvee$), and *NOT* ($\neg$). A term may be a single word, e.g., "Titanic", or a phrase, e.g., "Julia Roberts", enclosed in quotation marks. Given a database D and a précis query Q, an *initial tuple* is one that has at least one attribute containing a query term, whether appearing in a positive or a negative form. An *initial relation* is one that contains at least one initial tuple.

Given a *précis query* in the above syntax, its semantics, i.e., its answer, is determined by the query itself as well as by a set of constraints C that the answer is called to satisfy. Hence, before formalizing the query semantics, the form that these constraints may take is discussed next.

Constraints The set C of constraints is used to shape the logical database subset returned to a précis query in terms of its schema and tuples. Constraints may be specified by the user at query time for interactive exploration of the contents of a database or they may be stored in the system as part of user or application/service profiles. For example, a search service offered for small devices would make use of appropriate constraints for constructing small answers tailored to the display capabilities of devices targeted. The form of constraints may depend on application and user characteristics.

In our framework, we consider two generic meta-classes of constraints: *relevance* and *structural* constraints. Relevance constraints are defined based on weights on the database graph, whereas structural constraints are defined on the elements of the graph. To facilitate their usage, we provide a set of classes as a specialization (i.e., subclasses) of the generic meta-classes of constraints. Aggregate functions (e.g., min, max, avg, and sum), the usual comparison operators (e.g., $>$, $<$, and $=$), and combinations of the above can be used in order to construct a constraint instance. Figure 4 presents a taxonomy for constraints containing a non-exhaustive list of structural and relevance constraint classes and presents possible constraint instances. One can pick one of these examples or create any instance of a constraint class to express a particular information need.

For example, given a large database, a developer who needs to test software on a small subset of the database that, nevertheless, conforms to the original schema, may provide structural constraints on the database schema graph itself, such as *# of relations*, *# of attributes per relation*, and *# of joins* (length of paths in the database schema graph). On the other hand, a web user searching for information, would

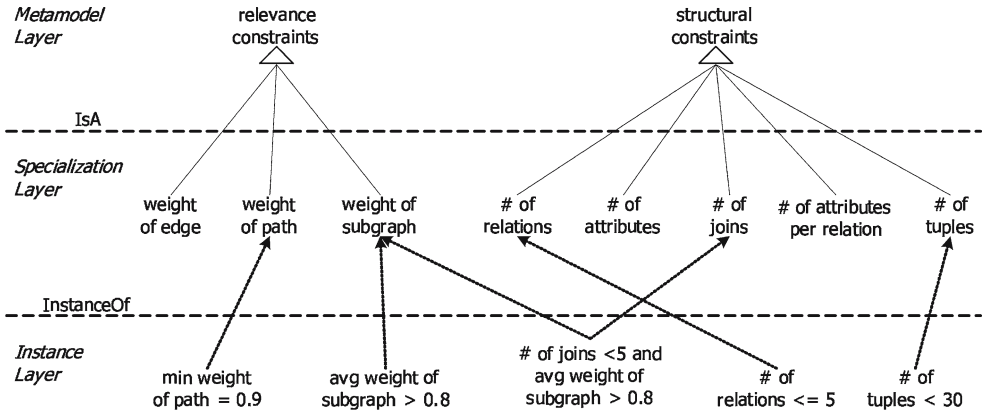


Fig. 4 A taxonomy of constraints

provide relevance constraints on the weights on the database schema graph, such as *minimum weight of path* and *minimum weight of subgraph* (described later). Additional structural constraints on the number of tuples may be specified in order to restrict the size of the subset.

Describing logical database subsets based on relevance constraints is intuitive for people. Furthermore, different sets of weights over the database schema graph and/or different constraints on them result in different answers for the same précis query, offering great adjustability and flexibility. For example, two user groups having access to the movie database could be: movie reviewers and cinema fans. 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. Likewise, different constraints on pre-specified sets of weights may also be used to facilitate different search contexts. For example, answers viewed on a cell phone would probably contain few attributes. More attributes may be projected in answers browsed using a computer. Multiple sets of weights corresponding to different user profiles may be stored in the system with the purpose of generating personalized answers [37]. For example, a user may be interested in the region where a theater is located, while another may be interested in a theater's phone.

Given a set of weights, changing constraints affects the part of database explored and essentially results in a different set of queries executed in order to obtain related tuples from this part of the database. 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.

Query semantics Given a query Q, consider its equivalent disjunctive normal form:

$$Q = \bigvee_i X_i \tag{2}$$

where $X_i = \bigwedge q_{ij}$ and q_{ij} is a keyword term or a negated keyword term ($1 \leq j \leq k_i$ with k_i the number of such terms in the ith disjunct).

The result of applying Q on database D with schema graph $\mathbf{G}$ given a set of constraints C is a logical database subset L of D that satisfies the following: a tuple t in D appears in L, if there exists a disjunct X_i in Q for which the following hold, subject to the constraints in C:

- there exists a set of initial tuples in D that collectively contain all terms combined with *AND* in X_i, such that t is reachable in $\mathbf{G}$ from all these tuples;
- tuple t is not reachable from any initial tuple containing a negated term in X_i.

Special cases of the above definition are the following:

- In the case of Q being a disjunction of terms without any negations (*OR*-semantics), L contains initial tuples for Q and any other tuple in D that is transitively reachable by *some* initial tuple, subject to the constraints in C.
- In the case of Q being a conjunction of terms without any negations (*AND*-semantics), L contains any tuple in D (including initial tuples) that is transitively reachable by initial tuples collectively representing *all* query keyword terms, subject to the constraints in C.

Example Consider the following queries issued over the movie database, without any constraints.

q_1: *"Alfred Hitchcock" OR "David Lynch"*
q_2: *"Clint Eastwood" AND "thriller"*
q_3: *"Gregory Peck" AND NOT "drama"*
q_4: (*"Clint Eastwood" AND "thriller"*) *OR* (*"Gregory Peck" AND NOT "drama"*)

The answer of q_1 would contain initial tuples that contain Alfred Hitchcock and initial tuples involving David Lynch as well as tuples joining to any initial tuple. The answer for q_2 would contain joining tuples in which all terms are found plus all tuples that are connected to these in various ways. Thus, the answer would provide information about thrillers directed by Clint Eastwood, thrillers acting Clint Eastwood, actors playing in such thrillers, and so forth. The answer of q_3 would be any tuples referring to Gregory Peck and any joining tuples except for initial tuples for drama and any joining to these; e.g., dramas with Gregory Peck and any information related to these will not be included in the result. q_4 is a more complex query being the disjunction of q_2 and q_3. Therefore, its answer would contain any tuples satisfying each disjunct, i.e., tuples in the results of q_2 plus tuples returned for q_3.

3.4 Query interpretation

Consider a database D and a query Q in disjunctive normal form, i.e., $Q = \bigvee_i X_i$. Then, for each X_i, the part of the database that may contain information related to X_i needs to be identified. For this purpose, we first have to interpret X_i based on the database schema graph **G**. As we have already seen through the query examples of the previous subsection, more than one interpretation may be possible. For instance, one interpretation for query q_2 could be that it is about thrillers directed by Clint Eastwood and another could be that it concerns thrillers with Clint Eastwood as one of the actors. We observe that each interpretation refers to a different part of the database schema graph. In what follows, we formalize this by introducing the notion of initial subgraph.

Definition (Initial Subgraph) We assume that the graph **G**(**V**, **E**) corresponding to the schema of a database D is always connected, which holds for all but artificial databases. Given a query Q over D, an *initial subgraph* (*IS*) corresponding to a disjunct X_i in Q is a rooted DAG $S_G(V_G, E_G)$ on **G** such that:

- V_G contains *at least* one initial relation per query term in X_i, along with other relations that interconnect those,
- E_G is a subset of E interconnecting the nodes in V_G,
- the root and *all* sinks of S_G are initial relations.

For each disjunct X_i in Q, there may exist more than one initial subgraph, each one corresponding to a different interpretation of X_i over the database schema graph based on the initial relations found. Consequently, for a query Q, there may be more than one initial subgraph. If multiple initial subgraphs map to the same interpretation, then the most significant one is selected for consideration, as we will see later.

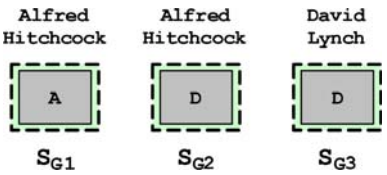


Fig. 5 Initial subgraphs for query q_1

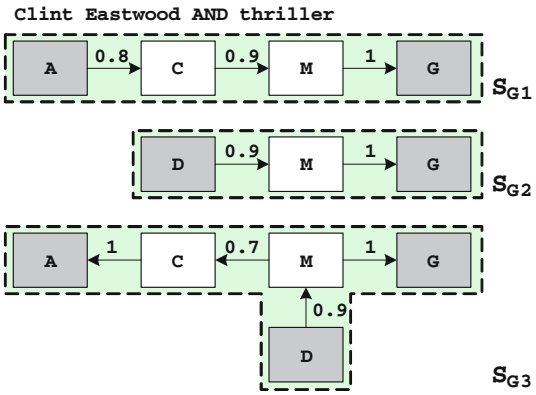


Fig. 6 Initial subgraphs for query q_2

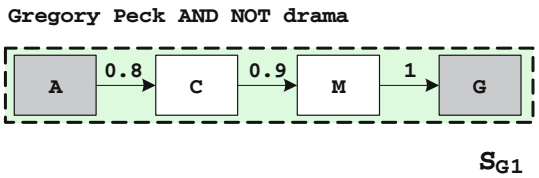


Fig. 7 Initial subgraphs for query q_3

Example We consider the queries of the previous example. Figures 5, 6, 7 depict the initial subgraphs for each query based on initial relations that may be found in a specific instance of the movie database. Each initial subgraph is denoted with S_G. Initial relations are depicted in grey. For simplicity, attributes of component relations are omitted and relation names are indicated by their initial letter only.

For query q_1, initial relations for Alfred Hitchcock are *DIRECTOR* and *ACTOR* and for David Lynch is *DIRECTOR*. Figure 5 depicts all the initial subgraphs for q_1, referring to the following interpretations: the director Alfred Hitchcock or the actor Alfred Hitchcock or the director David Lynch. Clearly, being a disjunction of single terms,

each initial subgraph has an initial relation as its sole node. For q_2, initial relations for Clint Eastwood are *DIRECTOR* and *ACTOR*, while thriller is found in *GENRE*. The initial subgraphs of the database schema graph are shown in Fig. 6 and may be interpreted as referring to the following: (a) S_{G1}: the actor Clint Eastwood acting in thrillers; (b) S_{G2}: thrillers directed by the director Clint Eastwood; or (c) S_{G3}: the director Clint Eastwood has directed thrillers in which he has also played as an actor. For query q_3, *ACTOR* and *GENRE* are the initial relations for Gregory Peck and drama, respectively. Figure 7 shows the initial subgraph corresponding to q_3, which is interpreted as referring to movies that are not dramas and have Gregory Peck as an actor. Finally, the set of initial subgraphs for q_4 is the union of the sets of initial subgraphs of q_2 and q_3.

The weight w_{S_G} of a subgraph $S_G(V_G, E_G)$ is a real number in [0, 1] and is a function f_g on the weights of the join edges in E_G:

$$w_{S_G} = f_g(W_{S_G}) \tag{3}$$

where $W_{S_G} = \{w_{e_j} \mid w_{e_j}\ weight\ of\ e_j,\ \forall e_j \in E_G\}$. This function should satisfy the following condition:

$$f_g(W_{S_G} \cup \{w_{e_i}\}) \geq f_g(W_{S_G} \cup \{w_{e_j}\}) \Leftrightarrow w_{e_i} \geq w_{e_j} \tag{4}$$

This condition captures the intuition that, when building a subgraph, the most significant edges should be favored, i.e., edges leading to the most relevant information.

The weight of a subgraph represents the significance of the associations between its nodes. For instance, if $w_{S_G} = 1$, then all nodes in the graph are very strongly connected, i.e., presence of one node in the results makes all its neighbor nodes in the graph appear as well. As the weight of a subgraph goes to 0, the elements of the subgraph are more loosely connected.

In order for f_g to be incrementally computable as edges are added to its input set, it should either be *distributive* (computable from its value before the new edge has been inserted, e.g., count, min, max) or *algebraic* (computable from a finite number of values maintained with its input set before the new edge has been inserted, e.g., average). The so called *holistic* functions (e.g., median) require full recomputation and are excluded. In our implementation, we have chosen the following (algebraic) function for the computation of the initial subgraph weight:

$$f_g(W_{S_G}) = \sum_{p \in S_G} (w_p) \ / \ N \tag{5}$$

where p is any path in S_G connecting the root relation to a sink relation. This function expresses the average significance of the join of an initial relation to the initial subgraph, and in particular, to its root, and it captures the intuition that, as an initial subgraph grows bigger and associations between initial relation nodes become weaker, its weight is decreasing. By default, an initial subgraph containing only one relation has a weight of 1.

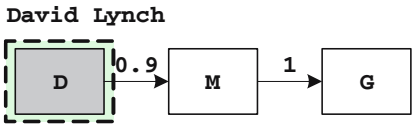


Fig. 8 Example expanded subgraph for query q_1

Example Naturally, not all of the possible initial subgraphs of a query are equally significant. For example, using the function above, we obtain the following weights for the initial subgraphs of query q_2 shown in Fig. 6 (from top to bottom): 0.72, 0.9, 0.765. Based on them, the interpretation of the query as "thrillers directed by the director Clint Eastwood" is the most significant.

Initial subgraphs for a query Q are considered in decreasing order of weight. If two subgraphs have the same weight, the one that contains more initial relations precedes. This order is taken into account during logical subset population (see Fig. 9) so that results are presented in order of relevance to the query.

Example Consider the following initial subgraphs over the movie database schema graph that could correspond to the query "John Malkovich" *AND* "Drama"; initial relations are underlined:

S_{G1} : $\underline{ACTOR} \rightarrow CAST \rightarrow \underline{MOVIE}$
S_{G2} : $\underline{ACTOR} \rightarrow \underline{CAST} \rightarrow \underline{MOVIE}$

Both of them have the same weight. However, S_{G2} precedes S_{G1} in order of significance, because it captures a stronger connection between query terms, e.g., actor "John Malkovich" has played himself in the movie "Being John Malkovich".

Definition (Expanded Subgraph) Given a query Q, a database D, constraints C, and an initial subgraph S_G, an *expanded subgraph* is a connected subgraph on the database schema graph **G** that includes S_G and satisfies C.

Example Consider the query q_1 and the initial subgraph corresponding to the query term David Lynch, which is shown in Fig. 5. Then, a candidate expanded subgraph is shown in Fig. 8.

Definition (Query Interpretation) Given a query Q, a database D, and constraints C, the set of all possible expanded

subgraphs of Q comprises the schema of the logical database subset $\mathbf{G}'$ that contains the most relevant information for Q based on C.

4 System architecture

In this section, we describe the system architecture for generating logical subsets of databases. This is depicted pictorially in Fig. 9.

Each time a précis query Q is submitted on top of a database D with schema graph $\mathbf{G}$, the following steps are performed in order to generate an answer.

Query parsing Given a query Q over a database D with schema graph $\mathbf{G}$, this phase performs two tasks. First, it transforms Q into a disjunctive normal form. As there are well-known algorithms for DNF transformations [44], we will not discuss this transformation any further. Then, it consults an inverted index over the contents of the database and, for each disjunct X_i of Q, a set of initial relations $\mathbf{IR_i}$ is retrieved (but not the corresponding initial tuples). If no initial relations are found, subsequent steps are not executed.

Logical subset schema creation This phase creates the schema of the logical database subset comprising initial relations together with relations on paths that connect them in $\mathbf{G}$, as well as a subset of their attributes that should be present in the query answer, according to the constraints.

Logical subset population This phase populates schema $\mathbf{G}'$ to create the logical database subset L. This contains initial tuples as well as tuples on the remaining relations of $\mathbf{G}'$, based on the query semantics and the constraints provided, all projected onto their attributes that appear as projection edges on $\mathbf{G}'$.

In the following sections, we elaborate on the main modules of the system architecture: the logical subset schema creation and the logical subset population.

5 Logical subset schema creation

Given a query Q over a database D with schema graph $\mathbf{G}$, this phase is responsible for finding which part of the database may contain relevant information with respect to a set of constraints C on the schema of the desired logical subset. In other words, this phase identifies the schema $\mathbf{G}'$ of the logical subset for the given query and constraints.

Example A user is interested in query q_2 and, particularly, in exploring answers in which the query terms are quite strongly interconnected and any additional information included is fairly important to all terms. The following relevance constraints have been specified for describing the desired

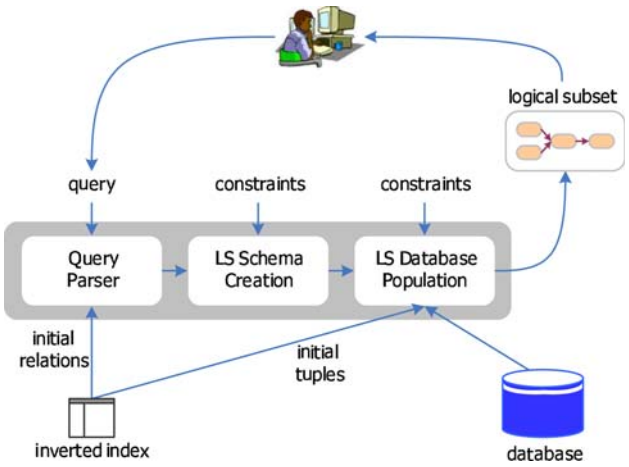


Fig. 9 System architecture

answers. In order to capture the first requirement, only graphs with weight over 0.7 are considered significant. The second requirement is expressed as a constraint on the expansion of subgraphs: a new relation is significant for a subgraph iff the weight of the path that connects this relation to every initial relation of the subgraph is equal to or greater than 0.8. We assume that the weight of a path is calculated by multiplying the weights of its constituent edges.

5.1 Problem formulation

The logical subset schema creation process is decomposed into two subproblems: initial subgraph creation and expansion. The first one is formulated as the problem of finding the *most significant* initial subgraphs, i.e., the most significant interpretations of Q, over the database schema graph based on the initial relations found for Q and the constraints. The second one refers to the expansion of the initial subgraphs on the database schema graph to include relation nodes containing most relevant information for the given query based on the constraints provided. These *expanded* subgraphs constitute the logical subset schema $\mathbf{G}'$. More formally, these subproblems are formulated as follows.

Initial subgraph creation Consider a database D with schema graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ and a query $Q = \bigvee_i X_i$ with constraints C. For each disjunct X_i, a set of initial relations $\mathbf{IR_i}$ is found in D. Then, the set of *most significant* initial subgraphs corresponding to Q is the set $\mathbf{S_G}$ such that:

$$\mathbf{S_G} = \{S_G \mid S_G(V_G, E_G),\ V_G \subseteq \mathbf{V},\ E_G \subseteq \mathbf{E},\ with\ S_G\ most\ significant\ IS\ w.r.t.\ C\ for\ \xi,\ \forall \xi \subseteq \mathbf{IR_i}\ containing\ all\ terms\ in\ X_i\ and\ \forall X_i\ in\ Q\ with\ \mathbf{IR_i} \neq \varnothing\}$$

In words, for each disjunct X_i in Q that has a non-empty set of initial relations, $\mathbf{IR_i}$, and for each combination of initial

relations ξ containing all terms in X_i, the most significant initial subgraph S_G on $\mathbf{G}$ is selected w.r.t. the constraints C.

Example The set of initial relations for q_2 is {*ACTOR*, *DIRECTOR*, *GENRE*}. All valid combinations of them are: $\xi_1 = \{ACTOR, GENRE\}$, $\xi_2 = \{DIRECTOR, GENRE\}$, $\xi_3 = \{ACTOR, DIRECTOR, GENRE\}$. ({*ACTOR*, *DIRECTOR*} is not a valid combination because it does not contain all terms of q_2.) For each one of them, the most significant initial subgraph is found, if it exists w.r.t. the constraints given. For instance, two initial subgraphs are candidate for ξ_1 (see also Fig. 3):

$\underline{ACTOR} \rightarrow CAST \rightarrow MOVIE \rightarrow \underline{GENRE}$

$\underline{ACTOR} \leftarrow CAST \leftarrow MOVIE \leftarrow \underline{GENRE}$

Using Formula (5), the weight for the first one is equal to 0.72 and for the second one is equal to 0.49. Both of them map to the same interpretation, thus, only the first one is considered, and since it satisfies the constraint, i.e., its weight is greater than 0.7, it is kept.

Expansion Consider the set $\mathbf{S_G}$ of initial subgraphs found over the database schema graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ for query Q based on constraints C. Each initial subgraph S_G is expanded as follows: a new edge is added provided that the target relation is significant for *all* initial relations in S_G w.r.t. the constraints. Then, for each initial subgraph S_G in $\mathbf{S_G}$, the set of all possible subgraphs of $\mathbf{G}$ that contain S_G ordered in decreasing weight is:

$$\{\mathbf{G_i} \mid \mathbf{G_i}(\mathbf{V_i}, \mathbf{E_i}),\ \mathbf{V_i} \supseteq \mathbf{V_1},\ \mathbf{E_i} \supseteq \mathbf{E_1},\ w_{i-1} \geq w_i, i \in [2, n]\},$$

and $\mathbf{G_1} = S_G$.

Thus, the *expanded* subgraph produced from S_G w.r.t. constraints C is $\mathbf{G_k}$ such that:

$k = max\{\, t \mid t \in [1, n] :\ \mathbf{G_t}\ satisfies\ constraints\}$.

Hence, the schema graph $\mathbf{G}'$ of the resulting logical subset L is determined by the set of all expanded subgraphs produced from $\mathbf{S_G}$.

The creation of the logical database subset schema $\mathbf{G}'$ is realized by the algorithm Logical Subset Schema Creation, *LSSC*, which is presented in Fig. 10 and is described in the following subsection.

5.2 Algorithm LSSC

The algorithm *LSSC* has the following inputs: (a) the database schema graph $\mathbf{G}$ corresponding to a database D, (b) a query $Q = \bigvee_i X_i$, (c) for each disjunct X_i, the set $\mathbf{IR_i}$ of initial relations, found during query parsing, and (d) the constraints C for shaping the logical subset. Its operation is summarized as follows:

First, for each disjunct X_i in Q, the algorithm finds the initial subgraphs that can be defined on the database schema graph based on the set $\mathbf{IR_i}$ of initial relations for X_i. If X_i is just a single (not negated) term, then each initial relation containing this term comprises an initial subgraph. For example, consider the initial subgraphs for query q_1 depicted in Fig. 5. If X_i contains more than one term, the creation of initial subgraphs is more complicated. For this purpose, algorithm *FIS* is used. Then, the set $\mathbf{S_G}$ of initial subgraphs produced for the whole query is enriched with the appropriate attributes and projection edges (algorithm *CIS*). Finally, each initial subgraph S_G in $\mathbf{S_G}$ is expanded to include additional relations that contain relevant information. This is performed by algorithm *EIS*, described later in this section.

Algorithm Logical Subset Schema Creation (*LSSC*)

Input: a database schema graph $\mathbf{G}(\mathbf{E}, \mathbf{V})$,
query $Q = \bigvee_i X_i$, constraints C,
$\{\mathbf{IR_i} \mid \mathbf{IR_i}\ a\ set\ of\ initial\ relations\ for\ X_i,\ \forall X_i\ in\ Q\}$
Output: a logical subset schema graph $\mathbf{G}'(\mathbf{E}', \mathbf{V}')$
Begin
0. $QP := \{\}$, $\mathbf{G}' := \{\}$, $\mathbf{S_G} := \{\}$
1. Foreach disjunct X_i in Q with $\mathbf{IR_i} \neq \varnothing$ {
   1.1 If X_i is a conjunction of literals {
      1.1.1 $\mathbf{S_G} \leftarrow FIS(\mathbf{G}, X_i, \mathbf{IR_i}, C, \mathbf{S_G})$ }
   else if X_i contains only one term {
   1.1.2 Foreach initial relation R_j in $\mathbf{IR_i}$ {
         mark R_j with different *s-id*
         add R_j to $\mathbf{S_G}$ }
   }}
2. $\{\mathbf{S_G}, \mathbf{G}, QP, \mathbf{G}'\} \leftarrow CIS(\mathbf{G}, C, \mathbf{S_G}, QP)$
3. $\{\mathbf{G}'\} \leftarrow EIS(\mathbf{G}, C, \mathbf{S_G}, QP, \mathbf{G}')$
4. return $\mathbf{G}'$
End

Fig. 10 Algorithm LSSC

All the aforementioned operations are driven by the constraints C and the database schema graph. The final output is the schema graph $\mathbf{G}'$ of the resulting logical subset L, which is determined by the set of expanded subgraphs produced over $\mathbf{G}$. Each expanded subgraph is assigned an id $s\text{-}id$. In this way, $\mathbf{G}'$ is represented as a subgraph of $\mathbf{G}$, whose edges and nodes are marked with the id's of the constituent expanded subgraphs.

Algorithm FIS The algorithm *FIS* (Fig. 11) has the following inputs: (a) the database schema graph $\mathbf{G}$ corresponding to database D, (b) a conjunction of literals X, (c) a set $\mathbf{IR}$ of initial relations for X, (d) constraints C and (e) a set of initial subgraphs $\mathbf{S_G}$. On the basis of these inputs, its objective is to find how X can be translated into one or more initial subgraphs on $\mathbf{G}$. These subgraphs are then added into $\mathbf{S_G}$. The latter may be initially empty or contain initial subgraphs from other invocations of *FIS*.

For this purpose, for each combination ξ of initial relations from $\mathbf{IR}$ containing all query terms in X, *FIS* has to find the most important initial subgraph that interconnects these initial relations, subject to the constraints C, if such subgraph exists. In doing so, there is a number of challenges to cope with. First, for a given combination ξ of initial

Algorithm Find Initial Subgraphs (*FIS*)

Input: a database schema graph $\mathbf{G}(\mathbf{E},\mathbf{V})$, a conjunction of literals X,
a set of initial relations **IR**, constraints C,
a set of initial subgraphs $\mathbf{S_G}$
Output: $\mathbf{S_G}$
Begin
0. QP := {}, $\mathbf{G_C}$:= **G**
1. Foreach $R_i \in \mathbf{IR}$ {
 1.1 mark relation R_i in $\mathbf{G_C}$ with different *s-id*
 1.2 Foreach $e(R_i,x) \in \mathbf{E}$, $x \in \mathbf{V}$ {
 1.2.1 If e satisfies constraints in C {
 $w_{sid} := f_G(w_e)$
 mark respective e in $\mathbf{G_C}$ with *s-id*
 add(QP, <e, *s-id*>)
 }}}
2. While QP not empty and constraints in C hold {
 2.1 get head <$e(R_i,R_j)$, *s-id*> from QP
 2.2 If destination relation R_j is not marked in $\mathbf{G_C}$
 and subgraph *s-id* is retained acyclic {
 2.2.1 mark respective R_j in $\mathbf{G_C}$ with *s-id*
 2.2.2 Foreach $e'(R_j,x) \in \mathbf{E}$, $x \in \mathbf{V}$ {
 If e' satisfies constraints in C {
 $w_{sid \cup e'} := f_G(W_G \cup w_{e'})$
 mark respective e' in $\mathbf{G_C}$ with *s-id*
 add(QP, <e, *s-id*>)
 }}}
 2.3 If subgraph with *s-id* contains new combination ξ from X
 and constraints in C hold {
 2.3.1 drop from subgraph all sink nodes n, s.t. $n \notin \xi$
 2.3.2 add subgraph in $\mathbf{S_G}$
 2.3.3 add any constituent initial subgraph in $\mathbf{S_G}$ with new *s-id*
 if it contains new combination ξ from X
 }}
3. return $\mathbf{S_G}$
End

Fig. 11 Algorithm *FIS*

relations, there may be more than one initial subgraphs satisfying the constraints. Ideally, we would like to build the most significant one and avoid building the others that will be ultimately rejected. Second, if the number of relations in **IR** is equal to the number of terms in X, then there is only one combination ξ, for which the most significant subgraph is desired. However, when there are more initial relations in **IR** than terms in X, then the number of possible combinations grows exponentially. For instance, if a query contains 2 terms, the first one found in relations R_1, R_2 and the second one found in relations R_3, R_4, then valid combinations would be: $\{R_1, R_3\}, \{R_1, R_4\}, \{R_2, R_3\}, \{R_2, R_4\}, \{R_1, R_2, R_3\}$, and so forth. Finding the most significant subgraph for each one of them independently is time consuming.

Recall that an initial subgraph is a rooted DAG with root an initial relation. In order to tackle the first challenge, *FIS* performs a best-first traversal of the database graph starting not from a single relation but from all initial relations that belong to **IR**. As we will show later in this subsection (Theorem 1), this ensures that subgraphs are generated in order of decreasing weight and that the most significant one is generated for each valid combination of initial relations. Furthermore, *FIS* essentially transforms the problem of "finding the most significant subgraph for each combination ξ (if it exists)", which is exponential of the number of initial relations, into the problem of "finding the most significant subgraphs considering each initial relation in **IR** as a possible root", which is linear of the number of initial relations.

The algorithm progressively builds multiple subgraphs. Each time an initial subgraph is identified that interconnects a different combination of initial relations, $\xi \in \mathbf{IR}$, containing all query terms, it is assigned an id, $s\text{-}id$, and it is placed in $\mathbf{S_G}$. We note that since **G** is considered connected, an initial subgraph corresponding to a combination of terms involving the *NOT* operator, should include initial relations that contain the negated terms. Hence, in this phase, initial relations containing negated terms are confronted in the same way as those containing simple terms.

More concretely, the algorithm considers a copy graph $\mathbf{G_C}$ of **G** in order to store the status of subgraphs progressively generated (Ln: 0). It starts from each $R_i \in \mathbf{IR}$ and constructs as many subgraphs as the number of relations in **IR** (Ln: 1). A list QP of possible next moves is kept in order of decreasing weight of subgraphs; a move, stored as $< e, s\text{-}id >$, identifies an edge e that adds a new relation to the subgraph with id $s\text{-}id$, and it is determined by the weight of this subgraph if extended to edge e (Ln: 1.2.1, QP's initialization;

step	(< $QP(e, s\text{-}id)$>, W_{s-id})	vis	A	C	M	D	P	T	G	$\mathbf{S_G} \leftarrow S_{id}$	$W_{S_{id}}$
1	(< D-M,2> , 0.9), (< A-C,1> , 0.8), (< G-M,3> , 0.7)	-	1			2			3		
2	(< M-G,2> , 0.9), (< M-P,2> , 0.81), (< A-C,1> , 0.8), (< G-M,3> , 0.7), (< M-C,2> , 0.63)	M,2	1		2	2			3		
3	(< M-P,2> , 0.855), (< A-C,1> , 0.8), (< M-C,2> , 0.765), (< G-M,3> , 0.7)	G,2	1		2	2			3,2	$\mathbf{D} \rightarrow M \rightarrow \mathbf{G}$	0.9
4	(< P-T,2> , 0.855), (< M-C,2> , 0.84), (< A-C,1> , 0.8), (< G-M,3> , 0.7)	P,2	1		2	2	2		3,2		
5	(< M-C,2> , 0.84), (< A-C,1> , 0.8), (< G-M,3> , 0.7)	T,2	1		2	2	2	2	3,2		
6	(< C-A,2> , 0.84), (< A-C,1> , 0.8), (< G-M,3> , 0.7)	C,2	1	2	2	2	2	2	3,2		
7	(< A-C,1> , 0.8), (< G-M,3> , 0.7)	A,2	1,2	2	2	2	2	2	3,2	$\mathbf{D} \rightarrow M \rightarrow \{\mathbf{G}, \{C \rightarrow \mathbf{A}\}\}$	0.765
8	(< C-M,1> , 0.72), (< G-M,3> , 0.7)	C,1	1,2	2,1	2	2	2	2	3,2		
9	(< M-G,1> , 0.72), (< G-M,3> , 0.7), (< M-P,1> , 0.648), (< M-D,1> , 0.648)	M,1	1,2	2,1	2,1	2	2	2	3,2		
10	(< G-M,3> , 0.7), (< M-P,1> , 0.684), (< M-D,1> , 0.684)	G,1	1,2	2,1	2,1	2	2	2	3,2,1	$\mathbf{A} \rightarrow C \rightarrow M \rightarrow \mathbf{G}$	0.72

Fig. 12 An example of the execution of *FIS* for query q_2

and Ln: 2.2.2, QP's incremental update). In each round, the algorithm follows the next best move on the graph (Ln: 2.1). As a result, the respective subgraph is enriched with a new join edge and relation. A relation may possibly belong to more than one subgraph; however, a subgraph should not contain cycles (Ln: 2.2).

Each time a subgraph is identified that interconnects a combination ξ of initial relations containing all query terms in X and ξ has not been encountered in initial subgraphs constructed earlier, then all sink nodes that are not initial relations are removed from this subgraph (Ln: 2.3). The resulting initial subgraph is placed in $\mathbf{S_G}$. An initial subgraph may also contain other initial subgraphs, all having the same root. Any of these subgraphs interconnecting a different combination, ξ, of initial relations containing all query terms in X is also placed in $\mathbf{S_G}$ (Ln: 2.3.3). *FIS* stops when constraints C do not hold or QP is empty, i.e., no possible moves on $\mathbf{G}$ exist.

Example The functionality of the algorithm FIS for the query q_2 is presented in Fig. 12. Recall that the relevance constraint specifies that only graphs with weight greater than 0.7 are significant for query q_2. For each step of the algorithm, the figure depicts: (a) the content of the list QP, i.e., the candidate edges for a subgraph $s\text{-}id$ in decreasing order of subgraph weight, along with the weight w_{s-id} that the subgraph will have if a respective edge is picked (the weight is calculated using Formula (5)); (b) the relation visited by a subgraph $s\text{-}id$ in this step (column vis in the figure); (c) the id's of the subgraphs that have visited each relation of the database graph so far; and finally, (d) the initial subgraphs found (if any) along with their weights.

FIS starts from the initial relations, *ACTOR*, *DIRECTOR*, *GENRE*, and progressively constructs three candidate initial subgraphs (step 1 of the figure and line 1 of the algorithm). The remaining steps depicted in the figure represent the generic functionality of line 2 of FIS. All candidate next moves are stored in QP in decreasing order of subgraph weight. Each time, the algorithm follows the next best move, thus, in the second step the subgraph with $s\text{-}id = 2$ visits the relation *MOVIE*. Observe that each time a new relation is added in a subgraph, then the weights associated with all the candidate edges in QP for this specific subgraph are updated: e.g., in step 3, if the edge M-P is picked, the weight of subgraph with $s\text{-}id = 2$ would be 0.855 instead of 0.81 that was in step 2, before the relation *MOVIE* was visited. Also, in step 3, after visiting the relation *GENRE*, the subgraph with $s\text{-}id = 2$ contains both relations *DIRECTOR* and *GENRE*, which is one of the valid combinations ξ of the initial relations. Thus, this subgraph is qualified as an initial subgraph and it is added in $\mathbf{S_G}$. The procedure goes on until QP is emptied or the top edge in QP can not produce a subgraph with a weight over 0.7.

Finally, we notice that in step 7, the subgraph produced is the $\mathbf{D} \rightarrow M \rightarrow \{\mathbf{G}, \{P \rightarrow \mathbf{T}\}, \{C \rightarrow \mathbf{A}\}\}$. However, it is not qualified as an initial subgraph because it contains a sink node that it does not belong to any ξ, so the pruning described in line 2.3.3 of FIS should take place and the resulting initial subgraph is then added in $\mathbf{S_G}$.

Theorem 1 (Completeness) *In the absence of constraints ($C = \emptyset$),* FIS *finds the set* $\mathbf{S_G^*}$ *of all possible initial subgraphs for a query Q. (Correctness) In the presence of constraints ($C \neq \emptyset$),* FIS *finds the set* $\mathbf{S_G}$ *of the most significant initial subgraphs for Q ($\mathbf{S_G} \subseteq \mathbf{S_G^*}$) that satisfy the given constraints.*

Sketch of Proof The main idea of the proof is divided into the following parts:

(a) All possible solutions, i.e., all possible initial subgraphs corresponding to ξ, are DAGs with root an initial relation from ξ. Since the algorithm starts searching from all initial relations in ξ and the schema graph is connected, assuming no constraints, all initial subgraphs are constructed. Consequently, *FIS* is complete.

(b) If $< e_o, S_{G_o} >$ is QP's head, then the following holds:

$$w_{S_{G_o} \cup e_o} \geq w_{S_{G_i} \cup e_i}, \forall < e_i, S_{G_i} > \in QP$$

Thus, S_{G_o} is the most significant subgraph from *all subgraphs currently* in QP.

Furthermore, for any subgraph S_{G_i} in QP (including S_{G_o}), upcoming moves are derived from edges that are already in QP. Consider such an edge $< e, S_{G_i} >$ as well as another edge $e' \notin QP$ that is also a potential upcoming move. Due to condition (1), $w_e \geq w_{e'}$ holds. By combining this with condition (4), we conclude that $w_{S_{G_i} \cup e} \geq w_{S_{G_i} \cup e'}$. Thus, S_{G_i} is the most significant subgraph from *all upcoming subgraphs* not currently in QP that will contain S_{G_i}. Consequently, *FIS* builds subgraphs in decreasing order of weight.

If a subgraph is found that does not satisfy the constraints, then the algorithm stops. By combining (a) and (b), we conclude that, no other larger subgraph would have satisfied the constraints either. Hence, when it stops, *FIS* has found the most significant initial subgraph (if any, with respect to the constraints) for ξ. Consequently, *FIS* is correct. □

We have described how the most important initial subgraphs are built. Next, we prepare the initial subgraphs for the expansion through a procedure described by the algorithm *CIS*, and then, we expand them using the algorithm *EIS*.

Algorithm CIS The algorithm *CIS* (Fig. 13) has the following inputs: (a) the database schema graph $\mathbf{G}$ corresponding to database D, (b) constraints C, (c) a set of initial subgraphs $\mathbf{S_G}$, and (d) a list QP that is initially empty. It performs three tasks. First, it enriches all $S_G \in \mathbf{S_G}$ with attributes and corresponding projection edges. In particular, for each relation $R_i \in V_G$, only attributes that satisfy the given criteria are kept

Algorithm Create Initial Subgraphs (*CIS*)

Input: a database schema graph $\mathbf{G}(\mathbf{E},\mathbf{V})$, constraints C,
a set of initial subgraphs $\mathbf{S_G}(\mathbf{E_G},\mathbf{V_G})$, a list QP
Output: $\mathbf{S_G}$, $\mathbf{G}$, QP, a logical subset schema graph $\mathbf{G}'(\mathbf{E}',\mathbf{V}')$
Begin
0. $\mathbf{G}' := \{\}$
1. Foreach initial subgraph $S_G \in \mathbf{S_G}$ satisfying constraints in C {
  1.1. Foreach relation R_i in S_G {
    1.1.1 mark corresponding node in $\mathbf{G}$
    1.1.2 create attribute nodes and projection edges
      in S_G for attributes of R_i satisfying constraints in C
    1.1.3 Foreach join edge $e(R_i,x) \in (\mathbf{E}-\mathbf{E_G})$,
      $x \in (\mathbf{V}-\mathbf{V_G})$, that retains S_G acyclic and
      satisfies constraints in C {
        add(QP, <e, s-id>)
  }}
  1.2. update $\mathbf{G}'$ with edges and nodes of S_G
  }
2. return $\mathbf{S_G}$, $\mathbf{G}$, QP, $\mathbf{G}'$
End

Fig. 13 Algorithm *CIS*

Algorithm Extend Initial Subgraphs (*EIS*)

Input: a database schema graph $\mathbf{G}(\mathbf{E},\mathbf{V})$, constraints C,
a set of initial subgraphs $\mathbf{S_G}(\mathbf{E_G},\mathbf{V_G})$, a list QP,
a logical subset schema graph $\mathbf{G}'(\mathbf{E}',\mathbf{V}')$
Output: $\mathbf{G}'(\mathbf{E}',\mathbf{V}')$
Begin
1. While QP not empty and constraints in C hold {
  1.1. get head <$e(R_i,R_j)$, s-id> from QP
  1.2. $S_G \in \mathbf{S_G}$ is the subgraph identified by s-id
  1.3. If destination R_j is not marked on $\mathbf{G}$ (i.e. $R_j \notin V'$),
    is significant for S_G, and satisfies constraints in C {
      1.3.1 mark corresponding node on $\mathbf{G}$
      1.3.2 create a new node in $\mathbf{V}'$ for R_j
      1.3.3 create attribute nodes and projection edges
        in $\mathbf{G}'$ for attributes of R_j satisfying constraints in C
    }
  1.4. If $R_j \in \mathbf{V}'$, $e(R_i,R_j) \notin \mathbf{E}'$ and e satisfies constraints in C
    { 1.4.1 insert e in $\mathbf{E}'$ }
  1.5. annotate $e \in \mathbf{E}'$ with s-id
  1.6. update S_G
  1.7. Foreach join edge $e'(R_j,x) \in \mathbf{E}$, $x \in \mathbf{V}$, that retains
    S_G acyclic and satisfies constraints in C {
    1.7.1 add(QP, <e', s-id>)
  }}
2. return $\mathbf{G}'$
End

Fig. 14 Algorithm *EIS*

plus any attributes required for joins, in which this relation participates in the initial subgraph S_G (Ln: 1.1.2). Then, it checks every out-going join edge of S_G and if it meets the constraints then it is added along with their respective $s\text{-}id$ to the list QP that is ordered in decreasing weight of edges (Ln: 1.1.3). So, QP contains candidate edges for expansion of the initial subgraphs, which is implemented by algorithm *EIS*. Finally, it updates the schema $\mathbf{G}'$ of the logical subset with the edges and nodes of S_G. To deal with the fact that subgraphs may share relations and edges on $\mathbf{G}'$, each relation and edge is annotated with the set of ids of the subgraphs using it. In this way, $\mathbf{G}'$ is represented as a subgraph of $\mathbf{G}$, whose edges and (relation) nodes are marked with the id's of the constituent expanded subgraphs.

The tasks above are performed only for initial subgraphs that meet the constraints provided. In this way, algorithm *EIS* will expand only these.

Algorithm EIS The algorithm *EIS* (Fig. 14) has the following inputs: (a) the database schema graph $\mathbf{G}$ corresponding to database D, (b) constraints C, (c) a set $\mathbf{S_G}$ of initial subgraphs, (d) a list QP that contains candidate edges for expansion of the initial subgraphs, i.e., departing edges from each S_G in decreasing weight and (e) a logical subset schema graph $\mathbf{G}'$ as defined by the initial subgraphs found. QP has been initialized by *CIS*. The objective of *EIS* is to extend initial subgraphs toward relations that may contain information that is also significant for the query. This is performed by gradually adding edges in a subgraph $S_G \in \mathbf{S_G}$ in order of weight, as long as the target relation is significant for *all* initial relations in S_G w.r.t. C. The set of expanded subgraphs built from the initial subgraphs in $\mathbf{S_G}$ comprises the logical subset schema graph $\mathbf{G}'$.

The algorithm proceeds as follows. While QP is not empty and the constraints hold, it picks from QP the candidate edge e with the highest weight belonging to a subgraph S_G with id $s\text{-}id$, and it checks whether its target relation R_j is significant for *all* initial relations in S_G w.r.t. C. For this purpose, it is sufficient to check whether it is important for the most "distant" initial relation in the subgraph; i.e., the one whose path to R_j has the minimum weight among all paths from other initial relations to R_j. Toward this aim and for optimization reasons, each relation in S_G keeps the weight of the path to the most distant initial relation from it. For each R_j accepted, the algorithm updates $\mathbf{G}'$ with this relation and the edge e. For each relation and join edge in $\mathbf{G}'$, it keeps track of the set of subgraphs to which they belong. This information is used during the logical subset population in order to keep track of the subgraphs that need to be enumerated in order to populate each relation. Also, *EIS* inserts into QP the edges that depart from R_j for possibly expanding the subgraph further away from this relation. These steps are explained below.

If R_j is accepted, then the following happen. If the node in $\mathbf{G}$ mapping to R_j is not marked (Ln: 1.3), then a new node in $\mathbf{G}'$ is created along with projection edges and attribute nodes in the way described before for relations of the initial subgraphs. The join edge e is inserted into $\mathbf{G}'$ provided that it is not already there, it satisfies the constraints, and its destination relation R_j exists in $\mathbf{G}'$ already. (Note that insertion of a relation may have failed due to the constraints.) Afterward, the edge in $\mathbf{G}'$ is annotated with the corresponding subgraph id (Ln: 1.5). Moreover, we add the node R_j and the respective edge e into this subgraph. Finally, all join edges starting from the node R_j are added in QP provided that they retain the

subgraph with id *s-id* acyclic and they satisfy the constraints (Ln: 1.7). The algorithm stops when no other relations satisfying the constraints can be added in graph $\mathbf{G}'$. Hence, the logical subset schema graph $\mathbf{G}'$ produced consists of a set of expanded subgraphs. Nodes in $\mathbf{G}'$ correspond to relations included in these subgraphs. Edges in $\mathbf{G}'$ correspond to the constituent edges of the subgraphs. Each relation and edge is annotated with the set of ids of the subgraphs using it.

Example Consider the initial subgraphs found for query q_2 (see also Fig. 12). Recall that the constraint for their expansion specifies that a new relation is significant for a subgraph iff the weight of the path that connects this relation to every initial relation of the subgraph is equal to or greater than 0.8. Observe that only the relation *MOVIE* may connect with a new relation, i.e., *PLAY*, with respect to the database graph depicted in Fig. 3. (Expansion toward other directions is not possible with respect to the specific constraint considered; e.g., $w_{M-D} = w_{M-C} = 0.7$, or it has already been done; e.g., toward *GENRE*.) As we have mentioned, it is sufficient to check if the weight of the path between the relation *PLAY* and the most distant initial relation of each subgraph satisfies the constraint. The relation *PLAY* is connected to *MOVIE* for all three subgraphs. Therefore, the most distant initial relations from *MOVIE*, hence from *PLAY* too, per subgraph are as follows:

Initial subgraph	Most distant I.R. from *M*	Distance of I.R. from *M*
D-M-G	D	0.9
D-M-{G,{C,A}}	A	0.7
A-C-M-G	A	0.72

As the weight of the edge M-P equals to 0.9 (and further, the same holds for the subsequent relation *THEATER* where $w_{P-T} = 1.0$), only the expansion of the first initial subgraph satisfies the constraint. Thus, the first subgraph is expanded as follows: D-M-{G,{P,T}}, while the other two initial subgraphs remain intact.

Theorem 2 (Correctness) *Given a query Q and constraints C,* EIS *builds the set of expanded subgraphs for Q that satisfy C.*

Sketch of Proof All possible solutions, i.e., all possible expanded subgraphs contains an initial subgraph. Similarly to *FIS*, the algorithm expands subgraphs by selecting in each round the most significant move. So, following the same philosophy as in the proof for Theorem 1, i.e., essentially combining conditions (1) and (4), one can prove that the algorithm constructs candidate expanded subgraphs in decreasing order of weight as long as all constraints are satisfied. In addition, to expand a subgraph by including a relation R_j, the algorithm checks whether or not R_j is important (i.e., satisfies the constraints in C) for the most "distant" initial relation in the subgraph C, i.e., the one whose path to R_j has the minimum weight among the paths from all other initial relations. If this is the case, based on simple arithmetic, one concludes that R_j is significant for *all* initial relations in this subgraph with respect to C. Consequently, for each initial subgraph of Q, *EIS* builds the corresponding expanded subgraph subject to the constraints C. □

6 Logical subset population

Given a database D and the schema graph $\mathbf{G}'$ of the logical database subset L for a query Q and constraints C, this phase is responsible for the population of L with tuples from D following the query semantics. For this purpose, we first describe a straightforward approach to logical subset population (*NaiveLSP*). Then, we will elaborate on the particular requirements and characteristics of the problem, and we will present two algorithms: *PLSP* and *LSP*.

6.1 Naïve approach

Consider a query $Q = \bigvee_i X_i$. A logical subset schema graph $\mathbf{G}'$ consists of a set of expanded subgraphs, each one of them corresponding to some X_i. A straightforward approach to logical subset population is to build for each subgraph an appropriate query that retrieves tuples taking into account the initial relations contained in this subgraph and the query semantics. In particular, for each expanded subgraph corresponding to some X_i in Q, a query is built that retrieves a subset of tuples from D such that: $\forall t_j \in R_k, \forall R_k$ belonging to this subgraph, the following hold (based on the subgraph's join edges): if X_i contains only one term, then t_j is an initial tuple or transitively joins to an initial tuple; otherwise, t_j transitively joins to initial tuples containing all query terms not contained in itself and it neither contains any negated terms nor transitively joins to any tuple containing such terms.

This approach, called *NaiveLSP*, proceeds in the following way. First, for each expanded subgraph on $\mathbf{G}'$, an appropriate query is built. The inverted index is used for retrieval of the id's of initial tuples. When these queries are executed, the results generated are used to populate each relation in the logical database subset L. At this point, special care is required so that duplicate tuples and tuples not satisfying constraints are not inserted in the relations of the result.

While being straightforward and simple to implement, this approach has several intricacies. First, we recall that for a query $Q = \bigvee_i X_i$, there may be more than one expanded subgraph corresponding to each X_i; hence, overall, for a query there may exist multiple subgraphs. As a result, multiple queries need to be assembled and executed. Furthermore, for X_i involving negated terms, the corresponding query is quite complex and expensive. In addition, relational queries

produce a single relation whose tuples are forced to contain attributes from several relations. Additional effort is thus required in order to transform these results into a logical database subset. A side effect of that is the generation of duplicate tuples, which need to be removed from the final output. Duplicate tuples are generated both as result of SQL semantics and also as a result of different queries.

The approach followed by *NaiveLSP* resembles those used for the population of join trees in DBXplorer [2] and candidate networks in DISCOVER [27]. It is actually an adaptation of them to our problem properly augmented with the additional functionality of splitting tuples returned from each query to smaller tuples for populating the relations of the logical subset. (For further details see Sect. 2.1.1.)

6.2 Algorithm PLSP

In [38], an approach is used for the population of logical subsets corresponding to single-term queries, which implements two simple yet effective ideas. The first one is generating the logical subset by a series of simple selection queries without joins, each one populating a single relation in the logical subset. In particular, initial tuples are retrieved first; then, tuples from any other relation in L are retrieved based on a list of values for the attribute that joins this relation to the logical subset. In this way, this method overcomes the problems of splitting tuples among multiple relations and removing duplicates, which both arise when executing a single query involving multiple relations, i.e., following the *NaiveLSP* approach. Furthermore, if a relation in the logical subset collects tuples that transitively join to more than one initial relation, then the algorithm tries to collect them all, before joining another relation to this one. This heuristic is used in order to reduce the number of queries executed.

This approach works for single-term queries. We extend the ideas described above for the population of logical subsets corresponding to précis queries of the form described in this paper. The algorithm *PLSP* (Fig. 15) has the following inputs: (a) the schema graph $\mathbf{G}'$ of the logical database subset L, (b) the set $\mathbf{S_G}$ of initial subgraphs corresponding to a query and (c) constraints C for shaping the desired logical subset. The algorithm proceeds in two steps in order to produce the logical subset L. First, it populates initial subgraphs (Ln: 1). Then, more tuples are retrieved and inserted into L by executing join queries starting from relations in the initial subgraphs and transitively expanding on $\mathbf{G}'$ (Ln: 2). Since a logical subset schema graph $\mathbf{G}'$ consists of a set of expanded subgraphs, these subgraphs may share joins, which may be executed multiple times. In order to reduce the number of joins executed and to avoid creating duplicate tuples, a join from R_i to R_j is not executed until all subgraphs to which this join belongs have populated R_i.

Algorithm Progressive Logical Subset Population (*PLSP*)

```
Input:  a logical subset schema graph G'(E',V'), constraints,
        a set of initial subgraphs S_G
Output: a logical subset L
Begin
0.  QP := {}
1.  Foreach initial subgraph S_G∈S_G satisfying constraints {
      1.1 execute query corresponding to S_G
      1.2 Foreach relation R_j in S_G {
            1.2.1 populate relation R_j with result tuples
            1.2.2 annotate tuples in R_j with matching id's from s-id's
            1.2.3 {QP,G'} ← addinQP(R_j,G',QP,constraints)
      }}
2.  While ( (QP not empty or ∃ joins in G' not fully executed)
        and (constraints hold) ) {
      2.1 If QP is not empty {
            2.1.1 get head <e(x,R_j),s-id's> from QP, x∈V'
            2.1.2 populate R_j with ExeJoin(e,s-id's,constraints)
            2.1.3 annotate tuples in R_j with matching id's from s-id's
            2.1.4 {QP,G'} ← addinQP(R_j,G',QP,constraints)
            Else
            2.1.5 populate R_j with ExeJoin(most important
                  pending join e in G' with destination R_j,
                  s-id's,constraints)
            2.1.6 annotate tuples in R_j with matching id's from s-id's
            2.1.7 {QP,G'} ← addinQP(R_j,G',QP,constraints)
      }}
3.  Return L as the G' populated with tuples
End

addinQP(relation R_j, graph G', list QP, constraints) {
  Foreach join edge e(R_j,x)∈E', x∈V' {
    mark those s-id's of e that have already populated R_j
    If e has all s-id's marked or due to constraints
      { add(QP,<e(R_j,x),s-id's>) }
}}
```

Fig. 15 Algorithm *PLSP*

A logical subset schema graph $\mathbf{G}'$ consists of a set of expanded subgraphs, each one of them corresponding to a disjunct X_i in the query Q. Each relation and edge on $\mathbf{G}'$ is annotated with the set of id's of the subgraphs using it. First, initial subgraphs on $\mathbf{G}'$ are populated in order of decreasing weight, so that important subgraphs are populated first (Ln: 1). In the case that an initial subgraph comprises a single relation, it is populated by simply retrieving initial tuples using their tuple id's retrieved from the inverted index. If an initial subgraph corresponds to some X_i that contains a combination of terms using *AND* and/or *NOT*, then an appropriate query is built and executed as in the *NaiveLSP* approach.

Each tuple retrieved, in this or subsequent steps, is annotated with the set of subgraph id's that produce it. This is a subset of the set of subgraph id's (**s-id's**) assigned to the relation to which a tuple belongs and it is used in order to know in which joins a tuple will participate. Additionally (Ln: 1.2.3), for each outgoing join edge from a relation R_j of an initial subgraph, the associated id's of subgraphs that have already populated R_j are marked. The algorithm keeps a list QP of joins that can be executed next in order of decreasing weight, along with the respective set of subgraphs id's for each join edge. An edge is inserted into QP provided that all its id's

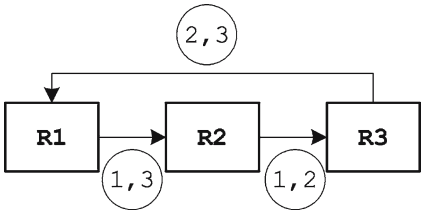


Fig. 16 Example of a deadlock

relation	tuple cardinality	"Clint Eastwood"	"thriller"
D	t_i^D, i=1,...,5	t_3^D	-
M	t_i^M, i=1,...,10	-	-
G	t_i^G, i=1,...,15	-	$t_1^G, t_2^G, t_3^G, t_8^G$
P	t_i^P, i=1,...,5	-	-
T	t_i^T, i=1,...,4	-	-

Fig. 17 Example of PLSP execution

are marked. This means that its source relation contains all possible tuples based on the subgraphs it belongs to; thus, this join may be executed.

Subsequently, a best-first traversal of the graph is performed. In each round, the algorithm picks from QP the candidate join e with the highest weight and executes it (Ln: 2.1). Thus, the target relation R_j of e is populated with new tuples, which are annotated with the appropriate subgraphs id's. Also, R_j's outgoing join edges are marked in the way described above. Edges with all their subgraph id's marked are inserted in QP and become candidate for execution.

A join from relation R_i to relation R_j is executed as follows (function $ExeJoin$, Ln: 2.1.5). From R_i, only tuples annotated with id's of subgraphs containing the respective join edge are processed. We consider the set of all distinct joining attribute values in these tuples. Then, tuples from R_j containing any of these values in the corresponding joining attribute are retrieved by executing an appropriate selection condition on R_j. Thus, we observe that a selection of tuples from R_j is performed instead of a real join of the two relations.

If QP gets empty and there are still joins in $\mathbf{G}'$ that have not been executed, then a deadlock has been identified: two (or more) joins cannot be executed because they wait for each other to be executed. In order to resolve this situation, the algorithm selects the join edge with the highest weight and performs a partial join (Ln: 2.1.5). A partial join from R_i to R_j is performed considering tuples currently contained in R_i. In this way, some new edges may be added to QP and the algorithm continues its normal execution. More tuples may be added to R_i as a result of executing subgraphs that traverse this relation and have not been marked on the edge yet. At any point, if a relation contains tuples for all subgraphs involving it, it is output, allowing for a progressive construction of the logical database subset. The algorithm stops if the constraints are violated or there are no join edges to be executed.

Example Figure 16 displays a deadlock. Assume that relations R_1, R_2, R_3 are initials. Edge $R_1 \rightarrow R_2$ is used by subgraphs $R_1 \rightarrow R_2 \rightarrow R_3$ and $R_3 \rightarrow R_1 \rightarrow R_2$. Subgraph $R_1 \rightarrow R_2 \rightarrow R_3$ cannot use edge $R_1 \rightarrow R_2$ since $R_3 \rightarrow R_1$ is not executed yet. The latter cannot be executed since subgraph $R_2 \rightarrow R_3 \rightarrow R_1$ waits for $R_1 \rightarrow R_2$ to be executed. In this case, the algorithm solves the deadlock by performing the partial join having the highest weight, let's say $R_1 \rightarrow R_2$. Then, the join $R_2 \rightarrow R_3$ is executed normally, since all subgraphs using it have produced tuples in R_2. Join $R_3 \rightarrow R_1$ is executed next. Finally, the partial join $R_1 \rightarrow R_2$ is executed for the tuples belonging to subgraph $R_3 \rightarrow R_1 \rightarrow R_2$.

Overall, algorithm *PLSP* involves two steps. Its advantages lie in the second step, where it proceeds by populating one relation at a time and by exploiting commonalities among subgraphs. The first characteristic allows the progressive output of relations in the logical subset. This is useful in scenarios where the logical subset is intended for use as a separate new database. The second one aims at reducing the number of queries executed. However, its first phase bears the problems mentioned for *NaiveLSP*, since, essentially, it populates initial subgraphs in the same way. Therefore, the more an expanded subgraph and its constituent initial subgraph overlap, the more *PLSP* behaves like *NaiveLSP* and the smaller benefit can be gained by its execution.

Example In Sect. 5, we showed with a running example how the schema of the logical subset concerning the motivating question q_2: "$Clint\ Eastwood''\ AND$ "$thriller''$ is found w.r.t. the database graph depicted in Fig. 3. That logical subset contained three expanded subgraphs: S_{G1}: A-C-M-G, S_{G2}: D-M-{G,{P,T}}, and S_{G3}: D-M-{G,{C,A}}. The subgraph S_{G2} has been expanded toward *PLAY* and *THEATRE*, while the other two are essentially the initial subgraphs because expansion has not taken place for them. We build upon that example to demonstrate the functionality of the population algorithms. Consider the instance of the movie database depicted in Fig. 17. Each relation contains a number of tuples; e.g., *MOVIE* contains 10 tuples. Also, the two keywords contained in q_2: "Clint Eastwood" and "thriller", can be found in two relations: the first in one tuple in relation *DIRECTOR* and the second in four tuples in relation *GENRE*, respectively.

For the population of the three subgraphs, first, the algorithm *PLSP* populates the initial subgraphs. Assume that for the subgraph S_{G2} the population is realized as depicted in Fig. 18a. Each outgoing edge from a relation belonging to an initial subgraph is stored in the list QP; in our example, the only such edge is the M-P for the subgraph S_{G2} (step 1 in the

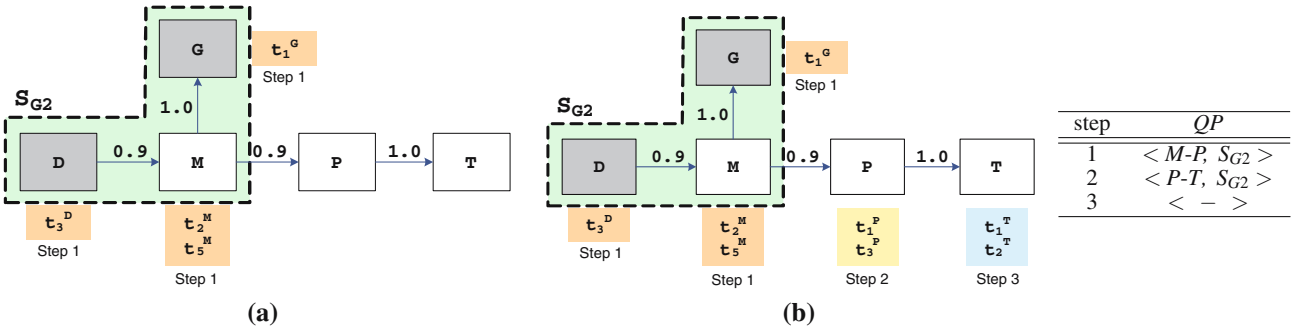


Fig. 18 An example instance of the movie database shown in Fig. 3

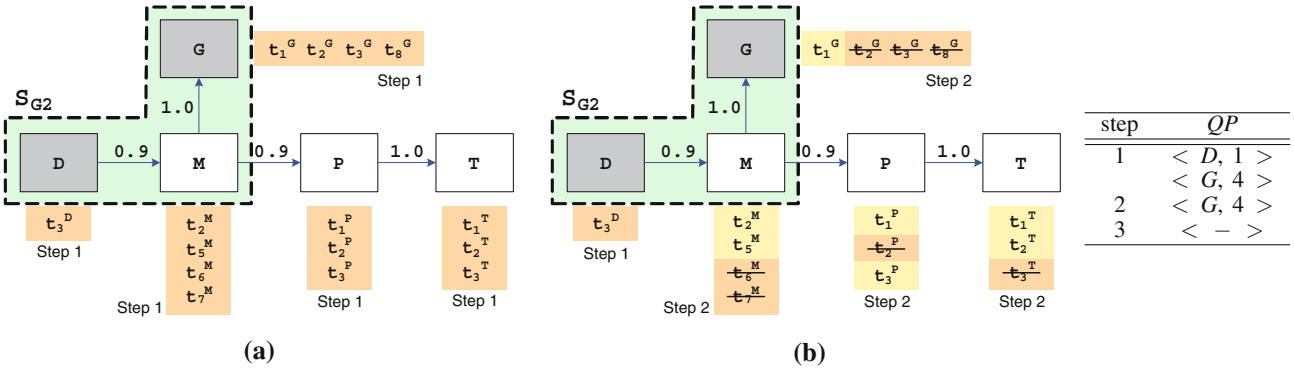


Fig. 19 Example of LSP execution

figure, line 1 of *PLSP*). Then, the population of the expanded subgraph S_{G2} follows a best-first approach by picking the edge with the highest weight; i.e., the edge M-P. The relation *PLAY* is populated and the edge P-T is added in the list QP (Fig. 18b, step 2). Finally, the relation *THEATRE* is populated too (Fig. 18b, step 3). If another subgraph contained the relation *PLAY*, then in the step 3, the relation *THEATRE* would not have been populated until *PLAY* was populated by all the subgraphs that contain it. As we have discussed and we will demonstrate in Sect. 7, this heuristic improves significantly the performance of *PLSP*.

6.3 Algorithm LSP

The algorithm *LSP* is driven by two requirements. First, the need to essentially overcome the relational query effect of generating results of joining tuples that need to be decomposed in order to populate relations in the logical subset. Second, the need to handle précis queries, and in particular those containing the *NOT* operator, more efficiently.

The algorithm (Fig. 20) takes as input the schema graph $\mathbf{G}'$ of a subset L and processes expanded subgraphs in decreasing order of weight. Thus, most important information will be generated first. The basic idea for populating an expanded subgraph on $\mathbf{G}'$ is the following. The algorithm considers initial relations in order of their estimated number of tuples in the result. It retrieves the initial tuples of the smallest initial relation, and it further populates other relations with tuples

Algorithm Logical Subset Population (*LSP*)

Input: a logical subset schema graph $\mathbf{G}'(\mathbf{E}', \mathbf{V}')$, constraints
Output: a logical subset L
Begin
0. $QP := \{\}$
1. Foreach expanded subgraph S of $\mathbf{G}'$ satisfying constraints{
   1.1 add in QP all $R_i \in \mathbf{IR}_S$ in increasing order of their estimated number of tuples in L
   1.2 pick head of QP, i.e., the smallest R_i
   1.3 populate $\mathbf{G}'$ with the initial tuples of R_i
   1.4 populate $\mathbf{G}'$ with tuples that join to the initial tuples in R_i
   1.5 mark R_i in $\mathbf{G}'$
   1.6 mark all tuples in $\mathbf{G}'$ with R_i-*id*
   1.7 re-estimate numbers of tuples in L and sort QP
   1.8 While QP not empty {
      1.8.1 pick head R_i of QP
      1.8.2 mark with R_i-*id* the initial tuples of R_i already in $\mathbf{G}'$
      1.8.3 mark joining tuples in $\mathbf{G}'$ with R_i-*id*
      1.8.4 re-estimate numbers of tuples in L and sort QP
   }}
2. unmark all tuples in $\mathbf{G}'$ with a negated term or joining to such tuples
3. Remove from $\mathbf{G}'$ the tuples that are not marked with the *id*'s of all $R_i \in \mathbf{IR}_S$ of a specific expanded subgraph S
4. Return L as the $\mathbf{G}'$ populated with tuples
End

Fig. 20 Algorithm *LSP*

that transitively join to these tuples. At the end of this step, the logical subset contains a superset of its final tuples. If the expanded subgraph corresponds to a single query term, then these tuples are the ones contained in the final logical subset. Otherwise, *LSP* examines the remaining relations in order of their estimated size in the final logical subset. For each initial relation, it marks all tuples in the logical subset that are initial tuples of this relation or join to them. At the end, the final logical subset consists only of tuples that have been marked by all initial relations.

In order to estimate the number of tuples for a relation in the result, standard statistics kept in the database are not appropriate, since they take into account attribute values and not words, which these values are composed of. For this purpose, we keep information regarding the frequencies of the words found in the database relations. The estimation of the number of tuples for a relation in the result is performed by taking into account (*a*) the fact that a query term may be found in more than one attribute in a relation, (*b*) the fact that more than one query term may be found in a relation, and (*c*) the frequencies of word occurrences in a relation.

LSP proceeds as follows. Consider an expanded subgraph S of $\mathbf{G}'$. A list QP stores initial relations in S that contain terms that are not negated in increasing order of the estimated number of their tuples in the logical subset. In order to populate an expanded subgraph S, the algorithm first populates $\mathbf{G}'$ with the initial tuples of the smallest initial relation in QP (Ln: 1.3) along with all tuples that transitively join to these initial tuples (Ln: 1.4). All these tuples are marked with the id of the initial relation considered (Ln: 1.6). Then, *LSP* proceeds with the rest of the initial relations in QP: For each one of them, it marks its initial tuples found in $\mathbf{G}'$ (Ln: 1.8.2) as well as all other tuples joining to them (Ln: 1.8.3). Consequently, each tuple stored in L is marked with a set of initial relations for the subgraph S examined. The final tuples are only those that have been marked by all initial relations of the subgraph they belong to, apart from those containing negated terms. Also, in this way, duplicates are not created.

Negated terms in a query, i.e., terms preceded with *NOT*, are treated as follows: when a tuple is found that contains such a term, its set of initial relations is emptied and the same happens to any tuple joining to it (Ln: 2). As a result, these tuples will not be finally output. In this way, *LSP* avoids executing expensive queries in order to make sure that tuples in L do not join to any tuples containing these terms. One can anticipate that it is sufficient to not consider the respective initial relations for the negated terms, in the first place. However, due to the fact that the database graph is connected, there is a necessity to include these relations in the search to ensure that no tuple transitively related to a tuple containing a negated term will be included in the logical subset. Finally, when all initial relations of S have been visited, *LSP* examines the next expanded subgraph.

Example Consider again the example presented for the algorithm *PLSP*. The instance of the movie database is depicted in Fig. 17. The population of S_{G2} using the algorithm *LSP* is realized as follows (Fig. 19). First, all the initial relations are added in QP in increasing order of their estimated number of tuples in the logical subset. Using the inverted index, we know that "Clint Eastwood" appears only once in the relation *DIRECTOR* in the tuple t_3^D and "thriller" appears four times in the relation *GENRE* in the tuples t_1^G, t_2^G, t_3^G, and t_8^G. Therefore, an entry in QP is created for each one of these two relations: $< D, 1 >$ and $< G, 4 >$, respectively. According to *LSP*, starting from the relation *DIRECTOR*, we find all tuples from all relations that join with the tuple t_3^D (step 1 in Fig. 19a). Afterward, we continue with the next initial relation *GENRE* which is now the head of the list QP. The tuples of *GENRE* that join with those already in the logical subset are marked, i.e., are kept, while the rest tuples are filtered out (step 2 in Fig. 19b). Then, the list QP empties and the algorithm terminates.

6.4 Cardinality of logical database subsets

Our framework provides a mechanism for data exploration and information discovery for all types of users either they expect a logical subset with all relevant information or a certain portion of it. Our algorithms are in fact independent of the weights and constraints used. If constraints are general enough or even if they do not exist at all, then the logical subset contains all relevant information to the initial query. For example, in this case, the answer to the question: "Clint Eastwood" AND "thriller", would contain the intersection of all information associated with these two query terms. On the other hand, if the constraints are more restrictive to the expansion of the logical subset, then the logical subset contains only a fraction of relevant information. In the latter case, a challenge arises: how many and which tuples from each relation to include in the logical database subset.

Although we left this issue for future work, a few suggestions can be made. The first suggests random population of the logical subset. Consider the following policy: populate relations in decreasing order of weight; i.e., the relation with higher weight is populated first, then the next one, and so on, until the constraints are satisfied. However, this may result in a database schema with some empty relations. Instead of overspending all resources to relations with high weight, a more conservative approach could be: populate relations with high weights with some tuples, continue with next relations, then populate the former relations again, and so forth. This procedure may be realized in a random fashion, but it should be biased in favor of relations with higher weight and relations that are expected to have greater cardinality due to their participation in $1{:}N$ or $N{:}M$ relationships, i.e., these should contain more tuples than the others. We note that the problem

of assigning an appropriate weight formula to the relations of a database has already been studied in literature (see Sect. 2).

Another approach could be to include the most relevant information from each relation in the logical subset. One may consider that each relation has a dominant attribute that characterizes its importance and allows the ranking of the tuples contained in the relation. For instance, in Fig. 3, relation *MOVIE* may have $BOXOFF$ as such an attribute, in the sense that movies that sold more tickets are more important than the others; while *ACTORS* and *DIRECTORS* may have the $NOMINAT$ attribute that keeps track of the nominations of an artist. Related literature has already presented results in this field: the Split–Pane approach could be used to return the top attributes for each relation [14]. Another discrimination criterion among relations is their weight. Thus, a combination of these two: (a) ordering of tuples in a relation, and (b) the weight of the relation, gives a global ranking criterion for tuples.

Whichever policy is selected, from either the aforementioned suggestions or the related work (see Sect. 2), it may be incorporated to our approach.

7 Experiments

7.1 Implementation

We have developed a prototype system in C++ on top of Oracle 9i release 2. Experiments were performed on a Pentium IV 3GHz machine with 512MB of main memory. Our database is stored on a local (7200RPM) disk, and the machine has been otherwise unloaded during each experiment.

The mechanism of inverted index is implemented in Oracle PLSQL. There are some details on its implementation we would like to emphasize. We consider attribute values in the database that can be decomposed into separate words. A single word is an alphanumeric expression, which may represent age, name, date and so forth. A word may be found in more than one tuple or attribute of a single relation and in more than one relation. For this reason, the system uses an inverted index that stores for each word k in D, a list of all its occurrences. A single word occurrence is a tuple $\langle R_j, A_l^j, Tid_l^j \rangle$, where Tid_l^j is the id of a tuple in relation R_j that contains word k as part of the value of attribute A_l^j.

Instead of storing all these tuples in a single structure, we use two structures for storing such information in a compact form. In particular, for each word found in an attribute value in the database, we store a bit vector, in which the jth element corresponds to the jth relation of the database. An element in a bit vector may be set to 0, showing that the corresponding relation does not contain any occurrence of this word, or to 1, showing that it contains at least one occurrence. This structure is used during the Query Parsing phase for efficiently retrieving the set of initial relations for a query. For each pair $\langle k, R_j \rangle$ in this structure, properly identified based on a hashing scheme, a separate structure keeps information about the initial tuples of R_j that involve k, in the form of a list (A_l^j, Tid_l^j's), where Tid_l^j's are the id's of the tuples in which attribute A_l^j contains the word k. This structure is used during the logical subset population phase for retrieving initial tuples.

Example Consider the query q_2 provided in the motivating example of Sect. 3.3: "Clint Eastwood" AND "thriller". For this query, the first structure of the inverted index returns the following information indicating the initial relations. (For simplicity, relations in the database are represented by their initial letters.)

		A	*C*	*D*	*G*	*M*	*P*	*T*
"clint"	=	1	1	1	0	0	0	0
"eastwood"	=	1	0	1	0	0	0	0
"thriller"	=	0	0	0	1	0	0	0

Observe that the term "Clint Eastwood" is a phrase and it can be split into two words that may be found in different relations. In order to determine the appropriate initial relations for this query term, the query parsing phase first combines the two index values into their logical summary (1010000). This summary shows which initial relations contain both words. Then, the second index structure is consulted in order to determine if the relations indicated from the first structure (*ACTOR* and *DIRECTOR*) contain at least one tuple that involves both words in the same attribute value; i.e., the whole phrase "Clint Eastwood". (Index values are case-insensitive.) The second structure shows for each word in an initial relation in which attribute and row id is located. (Multiple index values exist for multiple occurrences of the same word.)

Word Relation Attribute RowID
↓ ↓ ↓ ↓
<thriller, Genre> = <Genre, $AAAFd1...BSAA$ >

7.2 Experimental framework

For the experiments, we have used synthetic data and real-world data: movie data from IMDB (http://www.imdb.com) and restaurant data (http://www.gastronomia.gr). The movie database consists of a small number of relations with a large number of tuples per relation, and the restaurant database stores a large number of relations, each one of them having relatively few tuples. We conducted experiments to evaluate the efficiency of our algorithms for the creation of logical database subsets under several modifications of the parameters involved. These parameters are described below.

Table 1 Notation for the experiments

Query characteristics and constraints	
$width(Q)$	Number of disjuncts in Q
$span(Q)$	Maximum number of conjuncts in a disjunct of Q
C	Minimum path weight in the logical subset
Logical subset characteristics	
#S	Number of subgraphs
#RS	Number of relations per subgraph
#IRS	Number of initial relations per subgraph
#TLS	Number of tuples in logical subset
Database characteristics	
#DB	Database size

A logical subset is determined based on a précis query Q and constraints C. We consider that a query $Q = \bigvee_i X_i$ is characterized by two parameters:

- $width(Q)$ that is the number of disjuncts in Q, and
- $span(Q)$ that is the maximum number of conjuncts in a disjunct of Q.

Namely, if: $Q = X_1 \bigvee \cdots \bigvee X_m$ and $X_i = q_{i,1} \bigwedge \cdots \bigwedge q_{i,k_i}$, then $width(Q) = m$ and $span(Q) = max\{k_1, \ldots k_m\}$.

In order to illustrate the effect of constraints, we consider a relevance constraint C that represents the minimum path weight allowed in the logical subset. Furthermore, we consider the following parameters that denote the characteristics of the logical subset:

- *the number of subgraphs* (*#S*) that comprise a logical database subset
- *the number of relations per subgraph* (*#RS*)
- *the number of initial relations per subgraph* (*#IRS*)
- *the number of tuples in the logical subset* (*#TLS*).

The notation used for these parameters is shown in Table 1. In the following subsections, we describe the experimental results of our evaluation. First, we study the effect of the query characteristics, weights, and constraints on the logical database subset (Sect. 7.3.) This study enables us to understand better how the algorithms for the generation of logical database subsets are affected by these parameters (Sect. 7.4.) In particular, we can address questions, such as: Why algorithms display some particular behavior? Which changes in the logical subset characteristics affect most each algorithm? How well algorithms behave and adapt to such changes?

7.3 Logical subset characteristics

Effect of query characteristics Figure 21 illustrates the effect of $width(Q)$ and $span(Q)$ on the logical subset produced. Each result shown represents the average of 100 different experiment runs (100 queries with constraints) with the same configuration. For these experiments, we generated a random set of weights for the edges of the database schema graph. Based on these weights, we chose the constraint C to be equal to 0.3, so that a large part of the database schema graph could be explored and logical subsets with different characteristics could be generated. Figure 21 shows the impact of $width(Q)$ and $span(Q)$ on *#IRS*, *#S*, and *#TLS*. For the selected constraints, the number *#RS* of relations per subgraph is not determined by the query characteristics, therefore, we do not plot the corresponding results, as these are not interesting.

Figure 21a presents the number *#IRS* of initial relations per subgraph as a function of $span(Q)$, if $width(Q) = 1$. *#IRS* increases with the number of query terms but, interestingly, it is not equal to it. It is possible that a query term is found in more than one initial relation. In this case, the algorithms may build subgraphs that contain more than one initial relation per query term. For example, for a query that is a conjunction of three terms, i.e., $span(Q) = 3$, there may be subgraphs that contain 4 initial relations. On the other hand, *#IRS* does not depend on $width(Q)$, and that is also confirmed by the results in Fig. 21b.

Figure 21c, which shows the effect of varying $span(Q)$ on *#S* for $width(Q) = 1$, can be analyzed based on the observations we made above. Since more than one initial relation may be found for the same query term, this gives the opportunity to search for an increasingly larger number of subgraphs on the database schema graph based on the possible combinations of initial relations that contain all query keywords at least once. Consequently, the number *#S* of subgraphs that are actually found increases with $span(Q)$. Furthermore, Fig. 21d shows that *#S* is a multiple of $width(Q)$, which is expected.

Figure 21e shows that the number of tuples in the logical subset decreases for varying $span(Q)$ and $width(Q) = 1$, as expected due to query semantics. On the other hand, Fig. 21f shows that *#TLS* increases for varying $width(Q)$ and $span(Q) = 1$. One might expect that for $width(Q) > 1$, *#TLS* would be a multiple of *#TLS* for $width(Q) = 1$. However, this does not hold because, as the number of subgraphs

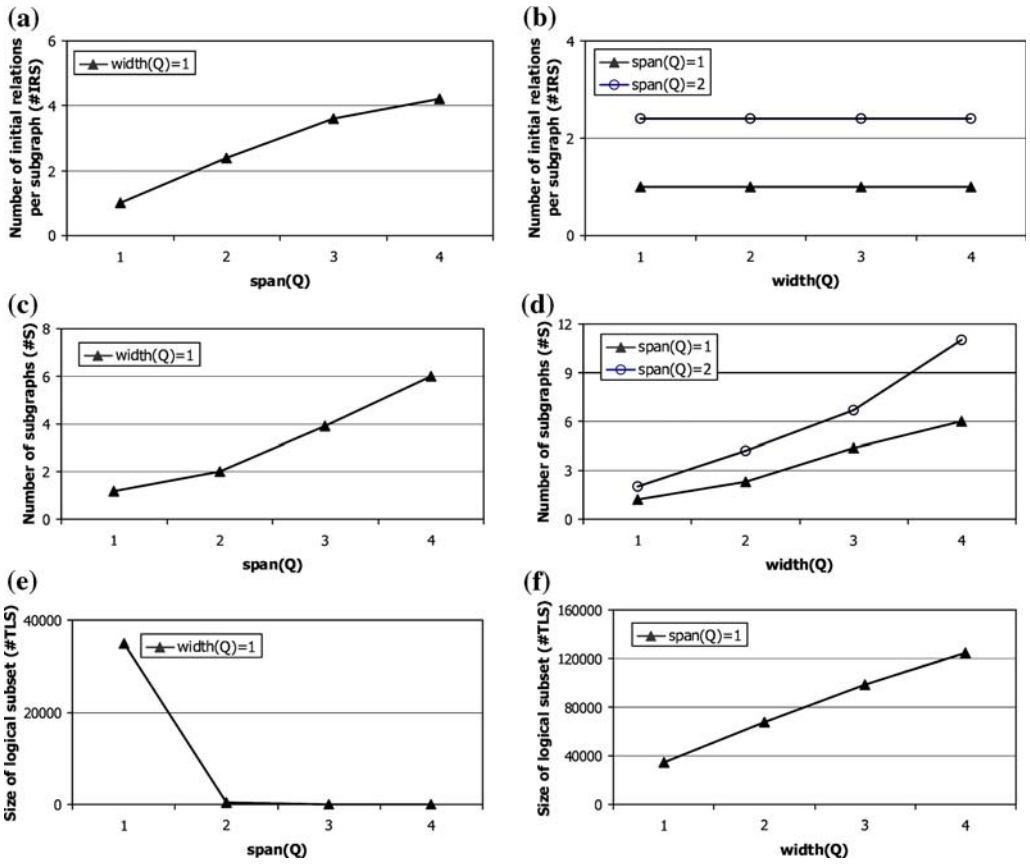


Fig. 21 Impact of query characteristics on the logical database subset

increases with $width(Q)$, more duplicate tuples may be found in different subgraphs; these are inserted only once in the logical subset.

Effect of constraints and weights Figure 22 illustrates how constraints and different sets of weights on the database graph shape the logical database subset. In order to evaluate the effect of the latter, we considered two scenarios: one where the database relations are strongly connected, and one where they are relatively loosely connected. For simulating the first scenario, we generated a random set of weights (*Set 1*) with the constraint that the minimum edge weight should be 0.85, while for the second scenario, we generated a set of weights (*Set 2*) with the requirement that the minimum edge weight should be 0.6. Note that we could have used other constraints, such as the maximum or the average edge weight, for generating sets of weights with the properties desired, i.e., one that would make database relations more strongly connected and one that would more loosely connect relations. However, our observations would not be significantly different.

Figure 22 shows the impact of the constraint C on *#S*, *#RS*, and *#IRS* for C ranging from 0.1 to 1.0 and for the two sets of weights. Each result shown represents the average of 100 queries with $width(Q) = 1$ and $span(Q) = 4$.

A general observation based on this figure is that *different constraints enable the construction of different logical subsets for the same queries*. In order to understand better the figure, we have to read the values of x-axis from the higher to the lower ones, that is, start with the stricter constraints and move down to the looser ones. We observe that all curves are similar and can be divided into three parts.

A very strict constraint, w.r.t. the weights considered, generates no logical subset and it is considered *cold*. As we make the constraint less strict, we find an increasing number of subgraphs (Fig. 22a), of increasing size, i.e., they include more relations (Fig. 22b), and of increasing complexity, i.e., they may contain more than one initial relation per query keyword (Fig. 22c). Therefore, the constraint has become *active*. The discovery of new subgraphs continues up to a certain point. Then, all subgraphs have been found for every valid combination of initial relations based on the database schema and they have been fully expanded. Relaxing even more the constraint has no effect on the logical subset. It is an *inactive* constraint.

For which interval of values a constraint is cold, how quickly it gets inactive, and in what degree it affects the logical subset, these are issues determined by the weights on the

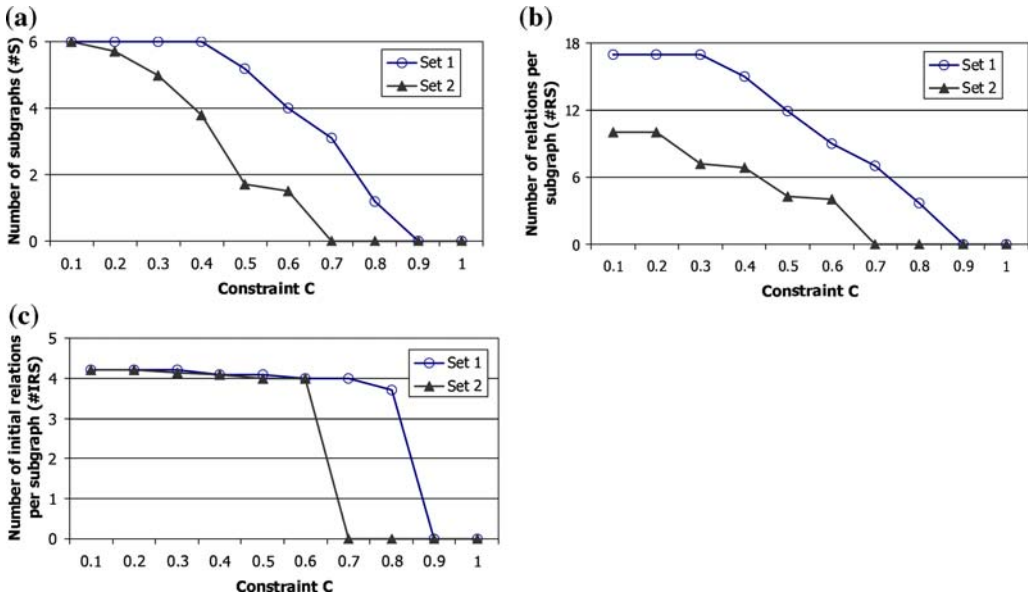


Fig. 22 Impact of constraints and weights on the logical subset

database schema graph. For instance, in Fig. 22a, we observe that, in the case of a database graph with high weights (e.g., *Set 1*), relatively strict constraints (in this case, with a threshold value close to 0.9) can be active. Moreover, since all join edges have large weights, relaxing the constraints quickly leads to generating all possible subgraphs extended to their full size (17 relations based on the database used in this series of experiments). Hence, constraints on such graphs become very soon inactive. On the other hand, strict constraints on graphs with lower weights, as in the case of *Set 2*, may be cold, and small changes in the constraint may not produce radical changes in the shape of the logical subset. Finally, we observe that using a set of weights, such as *Set 2*, allows exploring the database to a smaller extent than when using a set of high weights, such as *Set 1*. For instance, in Fig. 22b, we observe that different subgraphs are built based on the same constraints but different weights.

Overall, different sets of weights determine at which depth, in terms of relations examined, and with how much effort in terms of the constraints that need to be crafted, a user can look into the contents of a database with a single précis query.

7.4 Evaluation of algorithms

In this section, we discuss experimental results on the execution times of the algorithms for the creation and population of logical subsets. Our algorithms are comparable with some extent to DISCOVER [25,27] and DBXplorer [2], as we have seen in Sect. 2. Considering the answer generation problem at a more abstract level, our approach can be also compared with the Backward Expanding strategy used in BANKS [5] to a certain extent. In the subsequent discussion on algorithms' performance, we only refer to these comparable approaches. In the experiments, no constraints are assumed on the number of tuples in the logical subset, and each result shown represents the average of 100 different experiment runs (100 queries with constraints) with the same configuration. Times are shown in seconds.

***LS* schema creation** The first phase of the logical subset schema generation algorithm, *FIS*, is responsible for the initial subgraph creation, and corresponds to the candidate network generation in DISCOVER and DBXplorer. However, *FIS* is based on a best-first traversal of the database schema graph, which is more efficient and generates subgraphs in order of weight compared to the breadth-first traversal performed in candidate network generation. The *FIS* algorithm and the Backward Expanding strategy used in BANKS assume a different answer model. Overall, *FIS* fits our answer model better, and, in addition, it progressively builds subgraphs in decreasing order of weight, whereas the Backward Expanding strategy builds only trees in random order.

Figure 23 shows the impact of query characteristics and the database size on the performance of the algorithm *LSSC*. For these experiments, we used a random set of weights for the edges of the database schema graph and we set the constraint C to be equal to 0.3, as described in the previous subsection for the corresponding experiments regarding the effect of query characteristics on the logical database subset.

Figure 23a shows execution times for varying $width(Q)$ and $span(Q)=2$. Figure 23b presents execution times for varying $span(Q)$ and $width(Q)=1$. We observe that *LSSC*

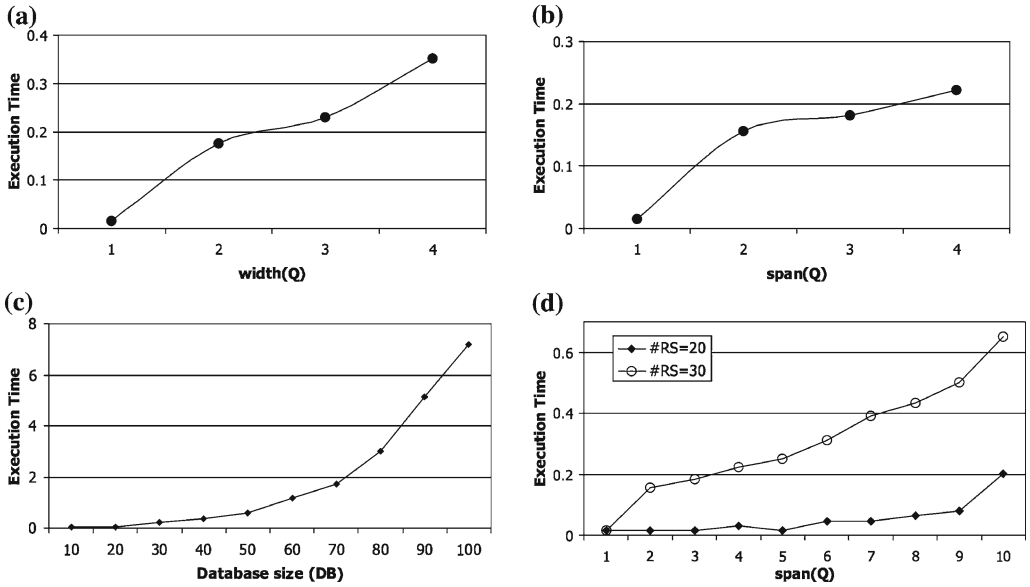


Fig. 23 Impact of query characteristics and database size on the efficiency of logical subset creation

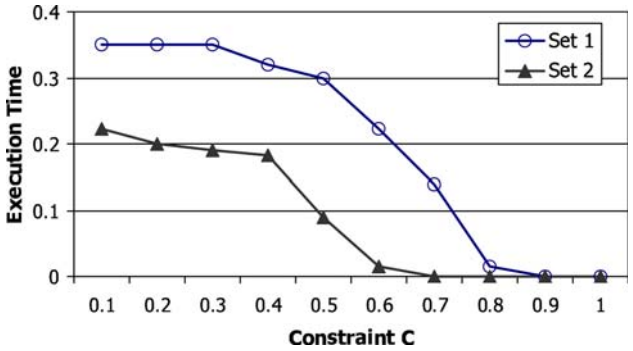


Fig. 24 Impact of constraints and weights on the efficiency of logical subset creation

is affected by changes in query characteristics but, overall, it exhibits good performance. Its behavior can be explained taking into account how different query characteristics lead to the generation of different logical subsets, as shown in Sect. 7.3. In particular, Fig. 21 shows that larger and more complex queries, i.e., of increasing $span(Q)$ and $width(Q)$, mostly affect the number *#S* of subgraphs in a logical subset. Consequently, the performance of LSSC is mainly determined by *#S*.

LSSC's performance has been also tested when the algorithm is used for constructing larger subgraphs. Experimental results are shown in Figs. 23c, d. For these experiments, we generated random synthetic graphs representing database schema graphs with varying number *#DB* of nodes mapping to relations. Figure 23c shows the behavior of algorithm *LSSC* for *#DB* ranging from 10 to 100 relations. As defaults, we have used $width(Q)=1$ and $span(Q)=4$. We observe that *LSSC* is quite efficient even for large database graphs. Figure 23d depicts execution times for $span(Q)$ ranging from 1 to 10, $width(Q)=1$, and for two databases comprising 20 and 30 relations, respectively. In both cases, the algorithm is very efficient.

Finally, Fig. 24 illustrates how constraints and different sets of weights on the database graph affect the execution times of *LSSC*. We use the same sets, *Set 1* and *Set 2*, as in the previous subsection. Each result shown represents the average of 100 queries with $width(Q) = 1$ and $span(Q) = 4$. In the case of modifying the constraints for determining the shape of the logical subset desired, execution times are not affected only by *#S*. As Fig. 22b depicts, changing constraints has a great impact on the number *#RS* of relations per subgraph. This explains the shapes of the curves in Fig. 24 and also the gap between them, which is due to the fact that using *Set 1* enables expanding subgraphs more and, in effect, examining more database relations. Consequently, when changing the weights and constraints that determine the logical database schema, LSSC is mostly affected by the change (increase or decrease) in the number of relations comprising the logical database schema.

***LS* population** The population algorithms proposed in DISCOVER [27] and DBXplorer [2] as well as the naïve of [25] follow essentially the same strategy: They execute one query per candidate network. We have adopted the same strategy into our naïve algorithm (*NaiveLSP*) and we compare it to the algorithms *PLSP* and *LSP*. The only difference of *NaiveLSP* from the naïve algorithms used in the aforementioned systems is that it needs to perform one additional step for splitting results among the relations contained in the logical subset. The additional algorithms proposed in [25], if adapted

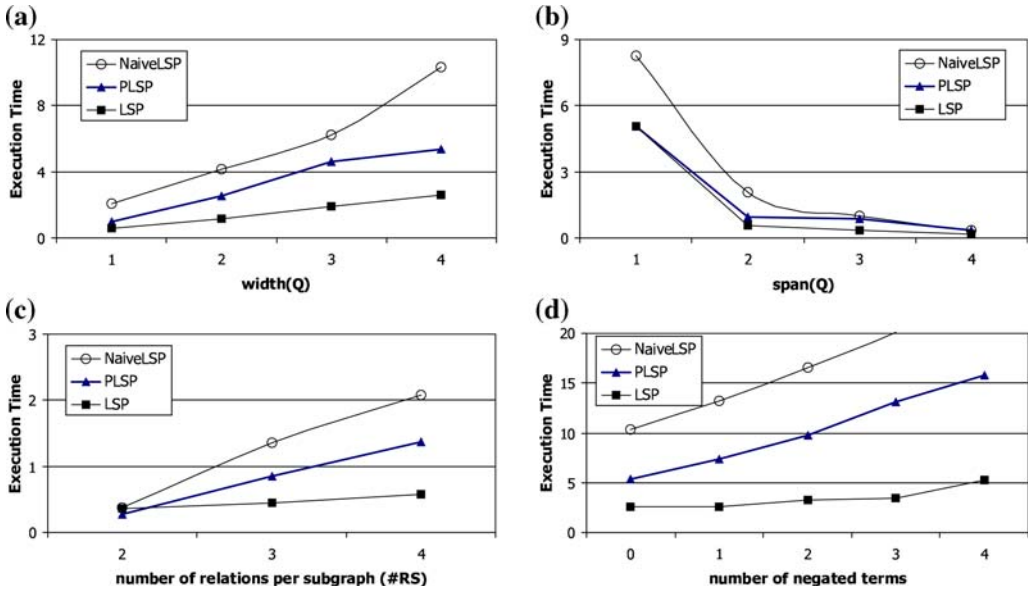


Fig. 25 Impact of query characteristics on the efficiency of logical subset population

to our problem, they can only be as good as the naïve one. The Sparse turns into the naïve algorithm (see Sect. 2). The Global Pipelined algorithm uses nested-loops joins, and when used for fully evaluating all candidate networks (or initial subgraphs in our case) is more expensive than the naïve approach [25]. Finally, the Hybrid uses one of the other two depending on the estimated answer size.

Figure 25 shows the impact of query characteristics on the performance of the algorithms *NaiveLSP*, *PLSP* and *LSP* for the population of logical subsets. For these experiments, we used a random set of weights for the edges of the database schema graph and we set the constraint C to be equal to 0.3 (if not stated otherwise). Figures 25a, b, c show execution times for queries that do not contain operator *NOT*, while Fig. 25d refers to queries with negated terms.

Figure 25a shows times as a function of $width(Q)$ with $span(Q)=2$. In order to explain the algorithms' behavior, recall from the previous subsection that increasing $width(Q)$ results in increasing *#S* and *#TLS*. This has a negative impact on the performance of all algorithms. *NaiveLSP*'s unsatisfactory performance is primarily due to the fact that *#S* queries are executed: each one of them returns a single relation, which contains tuples and attributes from multiple relations. Its tuples need to be decomposed into smaller ones for populating each relation in the logical subset. Hence, *NaiveLSP* also spends a considerable amount of time for sharing tuples across relations in the logical subset. Moreover, due to the existence of joins and SQL semantics, each query creates duplicates, which must be removed from the final relations. Also, as $width(Q)$ increases, the number of tuples processed by the algorithm *NaiveLSP* increases; thus, more duplicates may be created as the result of execution of different queries. On the other hand, *PLSP* involves two steps. In the first step, it uses the same approach as *NaiveLSP* in order to populate relations belonging to initial subgraphs. So, its first step may be time consuming depending on the size of initial subgraphs. In the second step, the algorithm executes a set of selection queries, each one directly populating one relation in the logical subset. Also, by exploiting commonalities among subgraphs, it manages to reduce the number of queries executed. The above explain why this algorithm demonstrates a better behavior compared to *NaiveLSP*. Moreover, as $width(Q)$ increases, *#S* inevitably increases, hence, *PLSP* processes more subgraphs with an increased probability that more commonalities between them are found. This is the reason why, in Fig. 25a, we observe that the gradient of the curve corresponding to *PLSP*'s times, slightly decreases as $width(Q)$ increases. Finally, *LSP* behaves well with respect to changes in $width(Q)$ because it first populates the smallest, in terms of tuples that match a query term, initial relation and then each one of the remaining relations in the logical subset. Then, it progressively marks the tuples that should remain in the final logical subset. In this way, it completely avoids the time-consuming steps of composing and decomposing complex tuples. It also avoids many duplicates that may be created from complex queries due to the SQL semantics.

Figure 25b depicts execution times for varying $span(Q)$ and $width(Q)=1$. Increasing $span(Q)$ affects *#S*, *#IRS*, and *#TLS*. Execution times decrease due to the rapid decrease in *#TLS*, as shown in Fig. 21e. Initially, *PLSP* exhibits a better behavior than *NaiveLSP*, which is explained by the fact that it executes a set of selection queries each one directly populating the respective relation in the logical subset. However, with $span(Q)$ growing, each initial subgraph tends to grow

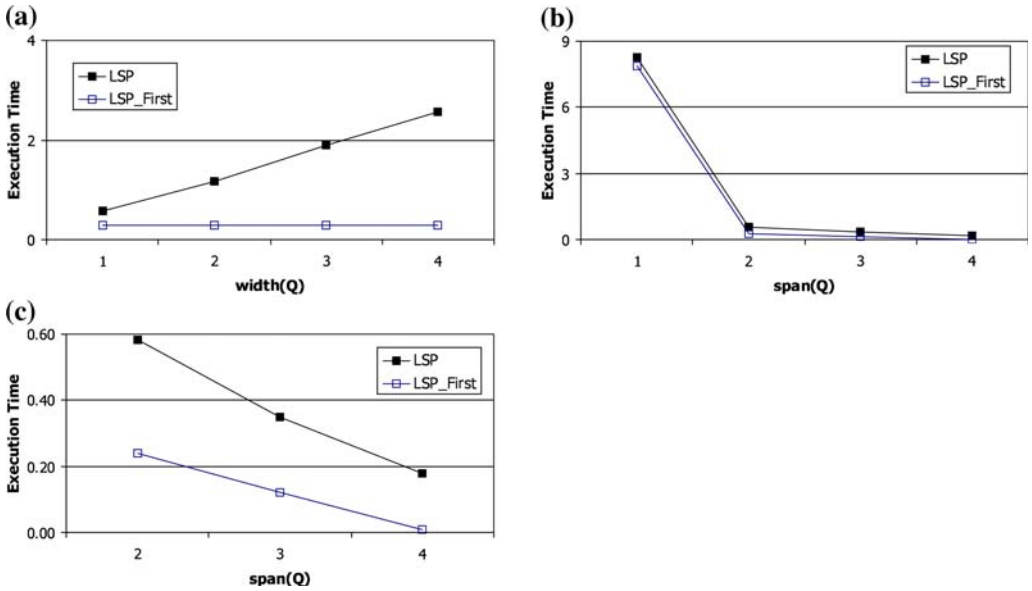


Fig. 26 Progressive behavior of LSP

to the size of the respective expanded one and *#IRS* tends to reach *#RS*. This has a great impact on *PLSP*'s performance, because it means that its first step, which is the time consuming one, dominates, while the second one shrinks. For this reason, execution times of *PLSP* and *NaiveLSP* converge as shown in the figure. On the other hand, *LSP*'s execution times are shaped by the cost of retrieving initial tuples for the initial relation populated first. Then, it progressively processes tuples that will remain in the final logical subset. Duplicates and tuples not satisfying query semantics are efficiently discarded. In particular, for $span(Q)=1$, i.e., for a query that is a disjunction of terms, we observe that *LSP* and *PLSP* coincide, which explains why they exhibit the same execution times at this point in the figure. Also, with increasing $span(Q)$, *#TLS* falls radically making the differences in execution times of the algorithms less pronounced.

Figure 25c presents results of a slightly different experiment. For this experiment, we used the number *#RS* of relations per subgraph as the constraint C, and we measured execution times as a function of *#RS* for queries that are characterized by $width(Q)=1$ and $span(Q)=2$. Increasing *#RS* means that more tuples will be processed. This explains why the performance of all algorithms deteriorates. *NaiveLSP* executes more complex queries, which tend to generate more duplicate tuples, due to the SQL semantics, which the algorithm has to remove from the logical subset. Moreover, it spends more time for sharing tuples when there are more relations in the logical subset. *PLSP* performs better, because it executes simple selection queries. *LSP*'s times increase smoothly with respect to *#RS*, because *LSP* tuple processing is more lightweight as we have already explained in previous paragraphs.

Figure 25d shows execution times for précis queries that contain zero up to 4 negated terms (operator *NOT*) and are characterized by $width(Q)=4$ and $span(Q)=2$. Thus, we consider queries that contain four disjuncts, each one of them containing two terms, with one of them possibly negated. We observe that *NaiveLSP* and *PLSP* are greatly affected by the number of negated terms in the query. They both execute a set of queries for logical subset population, each one of them generating results that satisfy one of the disjuncts. If a disjunct contains a negated term, then the respective queries are quite complex and time consuming to execute. Also, experiments have shown that such queries typically produce large result sets. All the above contribute to the unsatisfactory performance of the two algorithms. On the other hand, *LSP*'s times increase smoothly with respect to the number of negated terms in a query. The reason is that it does not execute complex queries. Instead, undesirable tuples, such as tuples that were initially brought in the logical subset but contain negated terms, are marked along with their joining tuples, so that they are not presented in the final answer.

Overall, *LSP* is the most efficient among the three logical subset population algorithms. *NaiveLSP* presents the worst performance due to the number of complex queries that executes in order to retrieve tuples and due to the post-processing of the results returned from these queries in order to populate the relations of the logical subset. *PLSP* can avoid some fraction of this post-processing and it can also take advantage of commonalities among different subgraphs to decrease the number of queries executed. *LSP* behaves well mainly because it completely avoids the time-consuming steps of composing and decomposing complex tuples as well as many duplicates that may be generated by complex

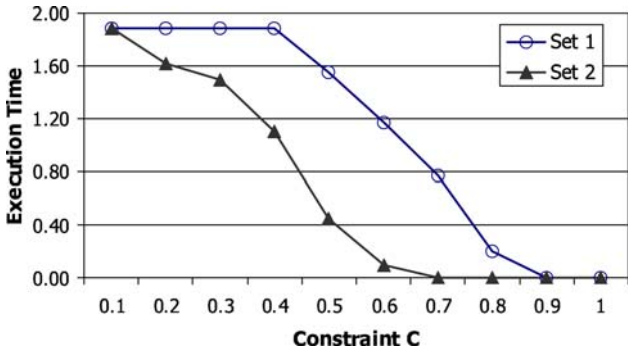


Fig. 27 Impact of constraints and weights on the efficiency of LSP

queries due to the SQL semantics. *LSP* is characterized by another nice feature: it can progressively generate results corresponding to different interpretations of a query. Recall that an interpretation maps to a subgraph constructed for the query. More than one subgraph may be built for the same query, as Figs. 23a, d depict. Figure 26 shows the times for populating the first subgraph (LSP_{First}) as a function of $width(Q)$ with $span(Q)=2$ (Fig. 26a) and as a function of $span(Q)$ and $width(Q)=1$ (Fig. 26b).

Figure 27 illustrates how constraints and different sets of weights on the database graph affect the execution times of *LSP*. We use again the sets *Set 1* and *Set 2*. Each result shown represents the average of 100 queries with $width(Q) = 1$ and $span(Q) = 4$. As we have seen in Figs. 22a, b, changing the constraints affects the number *#S* of subgraphs and the number *#RS* of relations per subgraph in the logical subset. This explains the shapes of the curves in Fig. 27. An implicit consequence of the effect on *#S* and *#RS* is that the number *#TLS* of tuples processed also changes. The impact of processing a different number of tuples is also visible in the distance between the two curves that correspond to the different sets of weights. *Set 1* enables expanding subgraphs more and, in effect, examining more database relations, thus more tuples.

Overall performance In this subsection, we present some representative experiments regarding the overall performance of our approach. Figure 28a shows times as a function of $width(Q)$ and $span(Q)=2$. We observe that the logical subset population phase, i.e., algorithm *LSP*, shapes total execution times. Figure 28b presents execution times for varying $span(Q)$ and $width(Q)=1$. We observe that total execution time decreases with $span(Q)$ increasing.

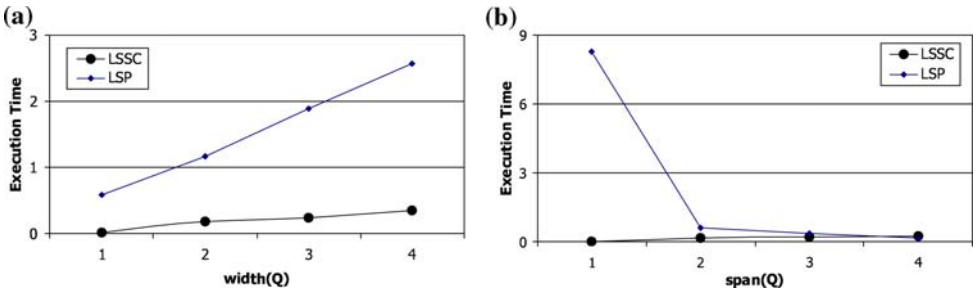
Thus, $width(Q)$ has a negative effect on overall performance. However, we recall that no constraints have been assumed on the number of tuples in the logical subset in any experiment. This number determines the performance of algorithm *LSP*. Consulting Fig. 21c, we see that as $width(Q)$ increases, *LSP* processes an increasingly larger number of tuples. Based on that, its performance is satisfactory. In web applications, constraints on the number of tuples will be probably used resulting in faster execution of *LSP*. Furthermore, another advantage of *LSP* is that it can progressively output results for different interpretations of a précis query as Fig. 26 illustrates.

7.5 User evaluation

The effectiveness of our approach was evaluated from 14 users with a good knowledge of SQL language and databases. The dataset used was the IMDB database. Users were provided with a list of 30 topics containing keywords from the inverted index, with the goal of finding information related to these topics. They chose 10 out of these topics to use in their search. Experiments consisted of two phases. In the first phase, users formulated and executed their own SQL queries. In the second phase, they repeated their search for the same topics using précis queries.

As expected, the total time they spent to think, write, and execute the query in the first phase is substantially greater than the respective time spent using our approach. However, it is of great interest to illustrate the difference between the times in each phase, that users spent to evaluate the results of the queries until they considered that they learned adequate information about each topic. Figure 29a depicts this difference in average times: using our approach (curve $T2$) they gained time in an approximate factor of 2.3. In both phases, users rated their effort and satisfaction in an increasing scale of satisfaction between [0,10]. Figure 29b presents the average results of the users' evaluation for: SQL before ($S1$) and after ($S2$) they used précis queries, and for précis queries ($S3$). Interestingly, not only did they prefer our

Fig. 28 Overall performance



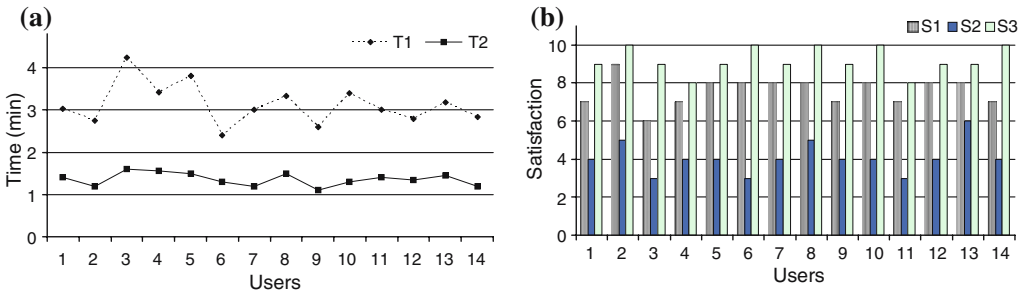


Fig. 29 Users' evaluation

approach better, but they lowered their grades for the usual practice. The dominant reasons for this behavior as reported by the users are: the easiness of query formulation offered by the précis framework in combination with the richness and improved insight offered by the answers, as well as serendipity of results.

8 Discussion

In this section, we briefly discuss two issues that are related to the overall evaluation of our approach: how cycles in query subgraphs may affect the creation of a logical subset and how homonyms and synonyms may affect the semantics of a précis query.

Cycles Currently, we consider subgraphs that contain no cycles. In general, cycles have termination problems and may arise under different circumstances. For instance:

- The edges between two relations corresponding to the same join are both integrated into a path, e.g., the two join edges that connect relations *MOVIE* and *DIRECTOR* in Fig. 3.
- Two relations are connected through more than one join, e.g., the relations *EMPLOYEE* and *DEPT* depicted in the example of Fig. 30.

Clearly, generation of logical database subsets is not limited by the existence of cycles. As far as the creation of the schema of a logical subset is concerned, a cycle may be considered only once in the schema with the use of a mechanism for cycles detection, e.g., an appropriate flag. The challenge is in the population of the logical subset. In this case, there is a termination problem that can be resolved either by using an appropriate threshold according to a respective constraint or user need, or by taking into consideration the value of weights, such as the weight of the path that connects a certain tuple with an initial tuple (or tuples).

For example, Fig. 31 represents a population scenario for the logical subset depicted in Fig. 30 that contains a cycle

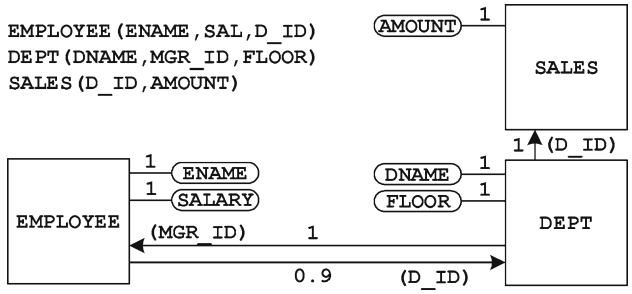


Fig. 30 Parallel join edges

between relations *EMPLOYEE* and *DEPT*. Assume that the population of the logical subset is performed based on a constraint that a tuple should be included into the logical subset if the weight of the path that connects this tuple with an initial one is no less than 0.9. Consider that multiplication is used to estimate the weight of a path between two tuples, in a sense similar to the weight of a path connecting two nodes presented in Sect. 3.1. If we consider that tuple t_1^E belonging to the *EMPLOYEE* relation is an initial tuple, then it is valid to include tuples from the *DEPT* relation too, e.g., t_1^D, because the weight of the path connecting it with the initial relation *EMPLOYEE* is 0.9. Next, there are two options: (i) due to the cycle, to include new tuples from the *EMPLOYEE*, e.g., t_2^E, connected with t_1^D (weight $= 0.9 * 1$) or (ii) to include tuples from relation *SALES*, e.g., t_1^S, connected with t_1^D (weight $= 0.9 * 1$). Thus, the situation depicted in Fig. 31b is valid. The termination problem has been resolved because it is not possible to include any more tuples from relation *DEPT* for the second time, because then the respective weight equals to $0.9 * 1 * 0.9 < 0.9$ (Fig. 31a).

The constraints used in our approach can be extended to cover the existence of cycles in the logical subsets, for example specifying a constraint on the number of cycles in a path or on the number of tuples contained in relations comprising a cycle, that are included in the logical subset. For instance, in Fig. 30, if the weight of the join edge from *EMPLOYEE* to *DEPT* equals to 1, instead of 0.9, then there is a need for a constraint to determine the number of tuples contained in this cycle that should be included in the logical subset.

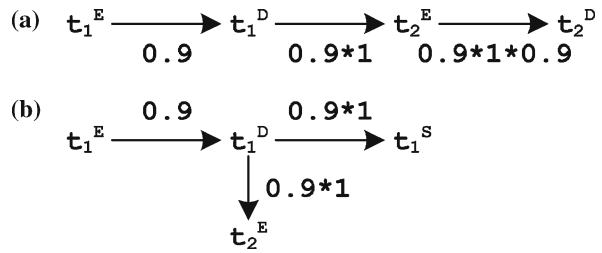


Fig. 31 Example population of LS containing cycles

Nevertheless, cycles do not limit the applicability of our approach presented before. It is an orthogonal technical problem that requires more detailed constraints.

Homonyms and synonyms In the movie database example used throughout the document, "Clint Eastwood" can be found in two different relations: *ACTOR* and *DIRECTOR*. It is straightforward to assume that the value "Clint Eastwood" refers to the same physical entity. In general, there exist several semantic problems that complicate the whole process.

It is possible that a single value may be used to represent different objects (homonyms); e.g., "Clint Eastwood" could correspond to two different persons, or different values may be used for the same object (synonyms); e.g., "C. Eastwood" and "Clint Eastwood" corresponding to the same person. To tackle the former problem, in the absence of any additional knowledge stored in the system, we may return multiple answers, one for each homonym, or obtain additional information through interaction with the user. For the latter problem, there exist approaches for cleaning and homogenizing string data, such as addresses, names, acronyms, and so forth [16,52]. Both these problems, however, are orthogonal to answering précis queries and, hence, are out of the scope of this paper.

9 Conclusions

The need for facilitating access to information stored in databases has not been a new one for the research community, but it has been amplified by the emergence of the World Wide Web. Existing approaches have either focused on facilitating users to formulate their information needs as queries or deriving meaningful answers from the underlying data.

In our earlier work, we have proposed précis queries, a uniform query answering approach to tackle both issues [38,53]. These are free-form queries that have structured answers. The answer to a précis query is an entire multi-relation database, a logical subset of an existing one. In this article, building upon our earlier work, we have described a generalized framework for précis queries with multiple terms combined with the logical operators *AND*, *OR*, and *NOT*, and we have formally defined the answer to such queries as a logical database subset. We presented algorithms that efficiently and correctly generate logical subsets. Their performance was evaluated and comparison with prior work exhibited their superiority. Finally, we presented experimental results providing insight so as to the effectiveness of the approach.

One interesting direction for future work is the selection of tuples to include in a logical subset under constraints on its size. Incorporation of notions of tuple ranking in our approach and appropriate coupling techniques are currently investigated. Another interesting problem is the incorporation of the logical subset into the internals of an open source RDBMS such as POSTGRES, and the possible extension of SQL language to support it. Finally, an open and challenging research direction is the application of the notion of logical subsets in different contexts from relational databases.

Acknowledgments We would like to thank Andrey Balmin for several helpful discussions about related work, his clarification of the inner mechanisms of ObjectRank and DISCOVER, and his verification that ObjectRank and our approach to précis queries are largely incomparable. Also, we thank the anonymous referees for their careful reading of the paper and their valuable comments that significantly improved its quality. 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. 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).

References

1. Agrawal, R., Rantzau, R., Terzi, E.: Context-sensitive ranking. In: SIGMOD Conference, pp. 383–394 (2006)
2. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A system for keyword-based search over relational databases. In: ICDE, pp. 5–16 (2002)
3. Agrawal, S., Chaudhuri, S., Das, G., Gionis, A.: Automated ranking of database query results. In: CIDR (2003)
4. Balmin, A., Hristidis, V., Papakonstantinou, Y.: ObjectRank: Authority-based keyword search in databases. In: VLDB, pp. 564–575 (2004)
5. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: ICDE, pp. 431–440 (2002)
6. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: WWW (1998)
7. Callan, J., Croft, W.B., Harding, S.M.: The INQUERY retrieval system. In: 3rd International Conference on Database and Expert Systems Applications, pp. 78–83 (1992)
8. Chakrabarti, K., Chaudhuri, S., won Hwang, S.: Automatic categorization of query resutls. In: SIGMOD Conference, pp. 755–766 (2004)
9. Chaudhuri, S., Das, G., Hristidis, V., Weikum, G.: Probabilistic ranking of database query results. In: VLDB, pp. 888–899 (2004)
10. Collins, A., Quillian, M.: Retrieval time from semantic memory. J. Verbal Learn. Verbal Behav. **8**, 240–247 (1969)
11. Cooper, W.S.: Getting beyond boole. Inf. Process. Manage. **24**(3), 243–248 (1988)

12. Croft, W.B., Harper, D.J.: Using probabilistic models of document retrieval without relevance information. J. Doc. **35**, 285–295 (1979)
13. Cui, H., Wen, J.R., Nie, J.Y., Ma, W.Y.: Probabilistic query expansion using query logs. In: WWW (2002)
14. Das, G., Hristidis, V., Kapoor, N., Sudarshan, S.: Ordering the attributes of query results. In: SIGMOD Conference, pp. 395–406 (2006)
15. Dixon, P.: Basics of oracle text retrieval. IEEE Data Eng. Bull. **24**(4), 11–14 (2001)
16. Dong, X., Halevy, A.Y., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD Conference, pp. 85–96 (2005)
17. Florescu, D., Kossmann, D., Manolescu, I.: Integrating keyword search into XML query processing. Comput. Netw. **33**, 1–6 (2000)
18. Florescu, D., Levy, A.Y., Mendelzon, A.O.: Database techniques for the World-Wide Web: a survey. SIGMOD Rec. **27**(3), 59–74 (1998)
19. Geerts, F., Mannila, H., Terzi, E.: Relational link-based ranking. In: VLDB, pp. 552–563 (2004)
20. Goldman, R., Shivakumar, N., Venkatasubramanian, S., Garcia-Molina, H.: Proximity search in databases. In: VLDB, pp. 26–37 (1998)
21. Graupmann, J., Schenkel, R., Weikum, G.: The SphereSearch engine for unified ranked retrieval of heterogeneous xml and web documents. In: VLDB, pp. 529–540 (2005)
22. Greiff, W.: Advances in information retrieval: recent research from the center for intelligent information retrieval. In: Bruce Croft, W. (ed.), The Use of Exploratory Data Analysis in Information Retrieval Research, pp. 37–70. Kluwer, Boston (2002)
23. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over xml documents. In: SIGMOD Conference, pp. 16–27 (2003)
24. Hamilton, J.R., Nayak, T.K.: Microsoft SQL server full-text search. IEEE Data Eng. Bull. **24**(4), 7–10 (2001)
25. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB, pp. 850–861 (2003)
26. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in XML trees. IEEE Trans. Knowl. Data Eng. **18**(4), 525–539 (2006)
27. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
28. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on XML graphs. In: ICDE, pp. 367–378 (2003)
29. Hulgeri, A., Bhalotia, G., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword search in databases. IEEE Data Eng. Bull. **24**(3), 22–32 (2001)
30. Hwang, F., Winter, P., Richards, D.: Steiner Tree Problem. Elsevier, Amsterdam (1992)
31. IBM: DB2 Text Information Extender. http://www-306.ibm.com/software/data/db2/extenders/text/
32. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)
33. Kimelfeld, B., Sagiv, Y.: Efficient engines for keyword proximity search. In: WebDB, pp. 67–72 (2005)
34. Kimelfeld, B., Sagiv, Y.: Efficiently enumerating results of keyword search. In: DBPL, pp. 58–73 (2005)
35. Kimelfeld, B., Sagiv, Y.: Finding and approximating top-*k* answers in keyword proximity search. In: PODS, pp. 173–182 (2006)
36. Kleinberg, J.: Authoritative sources in a hyperlinked environment. ACM J. **46**(5), 604–632 (1999)
37. Koutrika, G., Ioannidis, Y.: Personalized queries under a generalized preference model. In: ICDE, pp. 841–852 (2005)
38. Koutrika, G., Simitsis, A., Ioannidis, Y.: Précis: The essence of a query answer. In: ICDE, pp. 69–78 (2006)
39. Lawrence, S., Giles, C.L.: Searching the World Wide Web. Science **280**, 98–100 (1998)
40. Liu, F., Yu, C.T., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: SIGMOD Conference, pp. 563–574 (2006)
41. Maier, A., Simmen, D.E.: DB2 optimization in support of full text search. IEEE Data Eng. Bull. **24**(4), 3–6 (2001)
42. Marchionini, G.: Interfaces for end-user information seeking. J. Am. Soc. Inf. Sci. **43**(2), 156–163 (1992)
43. Masermann, U., Vossen, G.: Design and implementation of a novel approach to keyword searching in relational databases. In: ADBIS-DASFAA, pp. 171–184 (2000)
44. McCluskey, E.J.: Logic Design Principles. Prentice-Hall, Englewood Cliffs (1986)
45. Microsoft: SQL Server 2000. http://msdn.microsoft.com/library/
46. Motro, A.: BAROQUE: A browser for relational databases. ACM Trans. Inf. Syst. **4**(2), 164–181 (1986)
47. Motro, A.: Constructing queries from tokens. In: SIGMOD, pp. 120–131 (1986)
48. MySQL: MySQL. http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html
49. Oracle: Oracle 9i Text. http://www.oracle.com/technology/products/text/index.html
50. Robertson, S.E.: Readings in information retrieval. The probability ranking principle in IR, pp. 281 – 286. Morgan Kaufmann, San Mate (1997)
51. Salton, G.: The SMART System—Experiments in automatic document processing. Prentice-Hall, Englewood Cliffs (1971)
52. Sarawagi, S.: Special issue on data cleaning. IEEE Data Eng. Bull. **23**(4) (2000)
53. Simitsis, A., Koutrika, G.: Comprehensible answers to précis queries. In: CAiSE, pp. 142–156 (2006)
54. Simitsis, A., Koutrika, G.: Pattern-based query answering. In: PaRMa, pp. 41–50 (2006)
55. Simitsis, A., Koutrika, G., Ioannidis, Y.: Generalized précis queries for logical database subset creation. In: ICDE (2007)
56. Tao, Y., Hristidis, V., Papadias, D., Papakonstantinou, Y.: Branch-and-bound processing of ranked queries. Inf. Syst. **32**(3), 424–445 (2007)
57. Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., Srivastava, D.: Ranked join indices. In: ICDE, pp. 277–290 (2003)
58. Wang, Q., Nass, C., Hu, J.: Natural language query vs. keyword search: Effects of task complexity on search performance, participant perceptions, and preferences. In: INTERACT, pp. 106–116 (2005)
59. Wang, S., Zhang, K.: Searching databases with keywords. J. Comput. Sci. Technol. **20**(1), 55–62 (2005)