ABSTRACT

SEMANTIC, INTEGRATED KEYWORD SEARCH OVER STRUCTURED AND LOOSELY STRUCTURED DATABASES

by
Xinge Lu

Keyword search has been seen in recent years as an attractive way for querying data with some form of structure. Indeed, it allows simple users to extract information from databases without mastering a complex structured query language and without having knowledge of the schema of the data. It also allows for integrated search of heterogeneous data sources. However, as keyword queries are ambiguous and not expressive enough, keyword search cannot scale satisfactorily on big datasets and the answers are, in general, of low accuracy. Therefore, flat keyword search alone cannot efficiently return high quality results on large data with structure. In this dissertation, the algorithm improve keyword search over databases by exploiting semantic information of the data and by extending flat keyword queries with semantic information.

First, it develop an algorithm for keyword search over graph databases which exploits tree canonical forms and techniques developed for mining tree patterns. The algorithm substantially reduces the number of redundant intermediate results generated, which is the bottleneck of query evaluation algorithms. Our experiments show that it outperforms previous algorithms by one to two orders of magnitude in terms of efficiency and memory consumption and displays smooth scalability.

Furthermore, the algorithm leverages semantic information of the data to address the aforementioned problems. The method follows a schema-based approach for evaluating keyword queries on relational databases, which computes patterns mapped onto the schema graph of the database. Pattern graphs are representatives for clusters of query results. As such, they are much less numerous than the actual query results and can be translated into SQL queries on the relational database which can produce the results in the cluster. Our pattern graphs allow keywords to match schema elements and capture key-foreign key relationships and inclusion relationships. The the system employ information-retrieval-based and semantics-based techniques for scoring query pattern graphs and design an efficient top-k algorithm for computing the patterns graphs of a keyword query.

Finally, the dissertation study employing keyword queries enhanced with cohesiveness constraints (cohesive keyword queries) to query relational databases. Cohesive keyword queries bridge the gap between flat keyword queries and structured queries. The dissertation formally define semantics for cohesive queries on relational databases and design an efficient evaluation algorithm. The experimental results show that cohesive keyword queries substantially improve the quality of the results of flat keyword queries and the performance of their evaluation. Most importantly, these improvements are attained without compromising the simplicity and convenience of traditional keyword search.

SEMANTIC, INTEGRATED KEYWORD SEARCH OVER STRUCTURED AND LOOSELY STRUCTURED DATABASES

by
Xinge Lu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology,
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science, NJIT

December 2020

APPROVAL PAGE

SEMANTIC, INTEGRATED KEYWORD SEARCH OVER STRUCTURED AND LOOSELY STRUCTURED DATABASES

Xinge Lu

BIOGRAPHICAL SKETCH

Author:	Xinge Lu
Degree:	Doctor of Philosophy
Date:	December 2020

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2020
- Master of Science in Computer Science,
  Stevens Institute of Technology, Newark, NJ, 2015
- Bachelor of Science in Computer Science,
  Shanghai University, Shanghai, China, 2013

Major: Computer Science

Presentations and Publications:

Personalized Keyword Search on Large RDF Graphs based on Pattern Graph Similarity, Souvik Brata Sinha, Xinge Lu, and Dimitri Theodoratos, Proceedings of the 22nd International Database Engineering Applications Symposium, ACM, June 2018, Pages 12 - 21.

Leveraging Pattern Mining Techniques for Efficient Keyword Search on Data Graphs, Lu Xinge, Theodoratos Dimitri, Dimitriou Aggeliki, International Conference on Web Information Systems Engineering (WISE) 2019 Workshop, Hong Kong, China, Revised Selected Papers, Springer, Communications in Computer and Information Science, Volume 1155, Pages 98-114.

Leveraging Schema Information for Semantic Keyword Search over Structured Databases, Xinge Lu, Dimitri Theodoratos, Xiaoying Wu, Michael Lan, Submitted.

Semantics and Evaluation of Cohesive Keyword Queries on Structured Databases, Xinge Lu, Dimitri Theodoratos, Xiaoying Wu, Michael Lan, To be submitted

ACKNOWLEDGMENT

Throughout the writing of this dissertation I have received a great deal of support and assistance.

I would first like to thank my supervisor, Professor Dimitri Theodoratos, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

Second, I would like to first thank my dissertation committee who have each provided helpful feedback and have been great teachers who have prepared me to get to this place in my academic life. This project would not be nearly as good without their help.

Finally, I would like to thank my parents, and all my friends for their unwavering support and encouragement throughout the years.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

CHAPTER 1

INTRODUCTION

1.1 Motivation

A significant amount of the world's data is stored or exported in a form that exhibits some kind of (lose or tight) structure. For instance, a large part of corporate data resides in relational databases, popular NoSQL databases are organized as key-value stores, publicly available datasets—including datasets from the US government—can be downloaded in XML or JSON format, DBPedia is exported in RDF format, and social media connections are represented as graphs.

The tremendous success of the internet search engines over flat documents has triggered the last fifteen years intense research activity on searching with keywords databases and datasets with some form of structure. The ambition is to allow users to extract information deeply hidden in these data sources with the simplicity offered by keyword search. The target datasets of this research involve different data models including fully structured relational databases [86, 13, 45], tree structured databases [59, 5, 31, 47], graph databases [50, 79], key-value stores [76], RDF data graphs [78, 33, 25], entity databases [82] and spatial databases [34] among others. Having foreseen this interest on keyword queries, vendors of DBMSs have long ago added not only full text indexing capabilities to their products but also tools to support keyword search [3].

Benefits of keyword search on databases with structure. There at least three major reasons for this strong interest of the database and IR communities on keyword search over data with some structure. First, the users can retrieve information without mastering a complex structured query language (e.g., SQL, XQuery [51], SPARQL). We call this benefit simple user emancipation. Second, they can issue queries against the data without having full or even partial knowledge of the structure (schema) of the data source. We call this second benefit data structure independence. Third, they can query different data sources in an integrated way: the same query can be issued against and extract information from multiple data sources which might not all comply with the same data model (if at all). This is particularly important in web and big data environments where the data sources are heterogeneous and even if some of them adopt the same data model, they do not necessarily have the same structure schema. We refer to this third benefit as data model independence.

The problems. There is a price to pay for the simplicity, convenience and flexibility of keyword search. Keyword queries are imprecise and ambiguous in specifying the query answer. They lack expressive power compared to structured query languages [19]. Consequently, they generate a very large number of candidate results. This is a typical problem in IR. However, it is exacerbated in the context of data with some structure. Indeed, in this context the result to a keyword query is not a whole document but a data fragment (e.g., a subtree, or a subgraph) and this exponentially increases the number of results. This weakness incurs two major problems. The first problem is that existing algorithms for keyword search are of high complexity and they cannot scale satisfactorily when the number of keywords and the size of the input dataset increase. We refer to this problem as performance scalability problem. Note that top-k processing algorithms [41, 40, 61, 78, 38, 49, 83, 18, 35, 65, 31] do not solve the performance scalability problem as they still generate, in most cases, a large number of results (before identifying the top-k) or rely on specialized indexes which cannot be assumed to be available in practice.

The second problem is that the correct identification of the relevant results among a plethora of candidates becomes a very difficult task. Indeed, it is practically impossible for a search system to "guess" the user intent from a keyword query and the structure of the data source. Although previous approaches are intuitively reasonable, they are sufficiently ad-hoc and they are frequently violated in practice resulting in low quality results. We refer to this second problem as query answer quality problem. These problems have hindered the widespread use of keyword queries over data with some structure. Without additional information from the user, flat keyword search cannot efficiently provide accurate answers on databases with structure.

Previous efforts. A plethora of previous work have attempted to mitigate these problems by introducing query languages which bridge the gap between structured query languages and keyword search [23, 52, 7, 6, 85, 51, 77, 75, 63, 69, 81, 48, 62, 56]. They proceed either by adding structural constraints and other constructs to keyword queries or by introducing keywords as primitives to structured query languages.

These approaches not only do not succeed in offering fully the first and the second benefit (simple user emancipation and independence from the data structure), but they all fail to offer the third one (data model independence): each of them is designed for a specific data model and they are not applicable to databases of different types. For instance, schema-free XQuery [51] requires from the user to have some knowledge of the schema in order to write a query which is not simply a flat keyword query. Moreover, these queries are applicable only to XML tree-structured data. Similarly, the approach in [69], is designed for knowledge bases involving entities, concepts and relations. The user can identify in a query some keywords which will be mapped to relations. In order to do so, she not only has to know the type of the underlying knowledge base, but she also has to guess which concepts are modelled as relations in the data.

Exploratory search. Another way to bridge the gap between structured query languages and flat keyword queries on datasets with structure is exploratory search [64, 80, 8, 20, 3]. This process requires interaction with the user which starts with her issuing a keyword query. In its simplest form, the system returns a—sometimes ranked—list of possible interpretations of the unstructured keyword query usually represented by a structured query [55, 78, 12, 28, 5]. In its more involved form, it directs the search of the users by, conceptually, browsing classification hierarchies. The goal is to disambiguate the meaning of the keyword query and identify the user intention. The interpretation of the keyword query is constructed in a sequence of interaction steps [1, 27, 58, 57, 28, 25, 4, 10, 45, 26].

Exploratory search is particularly useful when the user does not know what information is available in the database and, as a consequence, cannot correctly express her information needs. As one can see, there is an abundance of contributions in this direction too. Irrespectively of how successfully these interfaces can interact with simple users, they all suffer from the same deficiency: they are all designed for a specific data model. For instance, [28] guides the user through an unmaterialized query hierarchy to construct a structured query reflecting the user's intent. However, the exploration system is designed for relational databases. In a similar way, [58, 57] serve XML databases, while [25, 26] operate on RDF graph databases. Therefore, these approaches cannot offer the data model independence benefit of flat keyword search. As such, they are not appropriate for integrated search across heterogeneous data sources.

1.2 Contribution and Results

Our goal in this dissertation is the developing of techniques and algorithms for empowering efficient integrated search over databases with some form of structure.

Many keyword search problems over structured databases translate into keyword search over graphs. For instance, we saw while working with relational databases that keyword search over relational databases requires computing results of keyword queries over graph-structured data. Graphs model complex relationships among objects in a variety of web applications. Keyword search is a promising method for extraction of data from data graphs and exploration. However, keyword search faces the so called performance scalability problem which hinders its widespread use on data graphs. We observed that existing algorithms over graph data generate a large number of redundant intermediate results and this negatively affects their performance. Our first task was to address the performance scalability problem by leveraging techniques developed for mining tree patterns (keyword search results over graphs are often trees). We focus on avoiding the generation of redundant intermediate results when the keyword queries are evaluated. We define a canonical form for the isomorphic representations of the intermediate results and we show how it can be checked incrementally and efficiently. We devise rules that prune the search space without sacrificing completeness and we integrate them in a query evaluation algorithm. We implemented our algorithm and experimentally tested it and compared it with previous algorithms on real and benchmark datasets. Our experiments show that our algorithm outperforms previous algorithms by one to two orders of magnitude in terms of efficiency and memory consumption and displays smooth scalability.

As keyword queries are ambiguous and not expressive enough, keyword search cannot scale satisfactorily on big datasets and the answers are, in general, of low accuracy. Therefore, we focused on exploiting semantic information to address the problems above and improve the effectiveness and efficiency of keyword search on relational databases. We follow a schema-based approach for evaluating keyword queries on relational databases: our approach returns patterns which are graphs casted on the schema graph of the database. Pattern graphs are representatives for clusters of query results. As such they are much less numerous than the actual query results. They can be translated into SQL queries on the relational database to produce results for the query. We defined pattern graphs which include schema components. They can distinguish between keywords matching schema elements like attributes and relation names and capture key-foreign key relationships and inclusion relationships. Our concept of pattern graph can record semantic information from the relational database that previous approaches cannot extract. We employ IR-based and semantics-based techniques for scoring query pattern graphs. We further design an efficient top-k algorithm for computing the patterns graphs of a keyword query. which exploits a normal form for pattern graphs to avoid the computation of redundant intermediate results. Our algorithm exploits a canonical form for pattern graphs to avoid the redundant generation of intermediate results which is the bottleneck of query evaluation algorithms. An extensive experimental evaluation on two real relational databases demonstrates the effectiveness of our approach and the time and memory efficiency of our algorithm.

Finally, in order to cope with the problems of keyword search on structured databases, we examined a technique which enhances keyword queries with constraints. Our approach uses cohesive keyword queries for querying relational databases. Cohesive keyword queries contain cohesiveness constraints which identify cohesive sets of keywords in the query. Intuitively, the occurrences of the keywords of a cohesive set in the result of a query form a cohesive whole which is not "penetrated" by the occurrences of the rest of the keywords. Cohesive queries allow for term nesting. They bridge the gap between flat keyword queries and structures queries. Although more expressive, they are as simple and convenient for the naïve user as traditional keyword queries. We formally defined semantics for cohesive queries on relational databases and we design an efficient evaluation algorithm. We run experiments to demonstrate the effectiveness of cohesive keyword queries and the efficiency of our algorithm. Our experimental results show that cohesive keyword queries substantially improve the quality of the results of flat keyword queries and their evaluation time, memory consumption and scalability. Importantly, these improvements are attained without compromising the simplicity of traditional keyword search.

1.3 Outline of the Dissertation

The rest of the dissertation is organized as follows: The next chapter reviews related work on keyword search on structured and loosely structured databases. In Chapter 3, we present our results for efficiently computing the results of a keyword query over graph databases. In Chapter 4, we discuss exploiting semantic information for improving keyword search on relational databases. Chapter 5 elaborates on the semantics and the evaluation of cohesive queries over relational databases. We conclude in Section 6 and suggest future work.

CHAPTER 2

STATE OF THE ART

2.1 Background

With the ever expanding internet and the tens of millions of existing websites, the amount of available data has been sharply increasing. Considering the difficulty users face to find the needed information, keyword search is one of the most popular information retrieval mechanisms.

The relational and the XML databases are popular data types. Both of them have a specific query language (SQL and XQuery [51], respectively) for retrieving information. In order to use these query languages, a user needs to have a sufficient understating of the data structure and of the language through practice. In relational databases, the information is separated into multiple tables which are connected by key-foreign key relations. The user has to locate the specific tables that contain data of interest and use the key-foreign key relations to connect the tables in order to get the result. In XML datasets, the schema is often complicated. The embedded XML structures pose a lot of difficulty in expressing queries that requiring the traversal of tree structures. Besides relational and XML databases, there are many graph structured datasets which do not have an obvious schema and do not even have a useful language associated with them for search. For graph databases, traditional graph search algorithms [43, 71, 84] can be used.

Compared to other approaches, keyword search has the benefits that is easy to use even by non-expert users and that the same query can be applied to databases with different structure and data models. The users do not need to have full understanding of the structure of the dataset to locate the information and the relationships between the data objects. Both relational databases and the XML databases can be viewed as graphs. Therefore, keyword search reduces to finding structural information among the data graph components using a keyword query Q. The results are subgraphs or trees which contain the keywords in Q and show the possible relationships between the keywords. In some approaches, weights are assigned to the nodes and edges of the data graph. The ranking method uses the weights to rank the final results. The most relevant results appear in the top results for the user to check.

Keyword search over graph data is challenging. The first question is how to associate the keywords in the query with the data components in the graphs. The simplest way is to find the exact matches of the keywords and data component labels. Exact matching is widely used in document information retrieval. The graph data not only has textual information, but also contains structural information and possibly a schema. Furthermore, a keyword query might be satisfied by a huge number of the subgraphs while the user might only be interested in one or a few of them. Top-k strategies effectively return the top relevant results. To improve the search efficiency, many systems propose ways to prune the search space or change the search strategies to reduce the generation of invalid results. Approaches to keyword search over graphs which do not take into account any schema information include the backward expanding search [14], the bidirectional search [44], the dynamic programming technique DPBF [32], and BLINKS [40]. The work presented in [24] focuses on finding the keyword search results in the external memory.

When doing the keyword search on XML Data, most of the approaches [39, 23, 52, 36, 46], will constrain the data to a tree structure.

The main research [42, 14, 3, 41, 61, 55] on the relational database focuses on generating a labeled graph of the database. The relations in the database are mapped to as nodes and the edges are based on the foreign-key relationships.

2.2 Keyword Search on Graphs

Graph data has widespread use in many applications. Most of them do not have a schema to describe their data. Keyword search on XML takes advantage of the hierarchical property of trees. Keyword search over a graph finds substructures of the graph containing all or some of the input keywords. BLINKS [40] is an implementation which focuses on schema-less node-label graphs. It uses an index to identify the tuples that contain keywords, but it also exploits the indexes to provide graph connectivity information which speeds up searches. BLINKS is based on cost-balanced expansion, which is an improvement of the backward search strategy [14]. The approach combines the search process with indexing. To reduce the search time in the shortest-path list, BLINKS partitions a data graph into multiple subgraphs or blocks. It also uses a bi-level index to keep a summary of the data which guides the search. The DPBF approach [32] uses a dynamic programming algorithm. It proposes an incremental method to get the top-k answers.

2.3 Keyword Search on Tree Data

XQuery [51] is usually used to query XML data. It provides flexible query facilities to extract data from virtual documents. Even though XQuery is expressive, it requires the users to be the experts in the language. The users also need to have full knowledge of the data schema. For naive users, keyword search is a friendlier search option.

2.3.1 Answer Ranking

In XRank [39], there are two possible semantics for keyword search queries. Under the conjunctive keyword query semantics, the result of the query is a data structure which contains all the keywords. Under the disjunctive semantics, the result data structure contains at least one of the keywords in the keyword query. Most existing approaches focus on conjunctive semantics.

There are many potential results according to the query semantics. Due to the difference of the XML structure, not all the answers are equally relevant. To rank the results that contain all the keywords, many approaches assign a numerical score to the result. First, the result with the more specific match should be ranked higher. Second, the result where the keywords are closer to each other should also be ranked higher. The tighter result structure means that the nodes in the answer are probably more related to each other. Furthermore, the ranking method has to consider the hyperlinks in the XML documents.

In XRANK, $ElemRank(v)$ is a function expressing the object importance of an XML element v computed using the underlying hyperlinked structure. $ElemRank(v)$ is similar to Google's PageRank, but $ElemRank(v)$ also takes the distance between the elements and the nested structure into account. For a sequence of containment edges $(v_1, v_2), (v_2, v_3), \ldots, (v_t, v_t + 1)$ such that $v_t + 1$ is a value node that directly contains the keywords k_i. The rank of v_1 with respect to a keyword k_i is $ElemRank(v_t)$ scaled appropriately to account for the specificity of the result: $r(v_1, k_i) = ElemRank(v_t) \times decay^{t-1}$. *decay* is a parameter that can be set to a value in the range 0 to 1. This rank ensures that less specific results get lower ranks, and is still related to $ElemRank(v_1)$ due to certain properties of containment edges. If there are multiple relevant occurrences of k_i, $\widehat{r}(v_1, k_i)$ is the max rank. The overall ranking $R(v_1, Q)$ is the sum of the ranks with respect to each query keyword, multiplied by a measure of the keyword proximity $p(v_1, k_1, k_2, \ldots, k_n)$.

$$R(v_1, Q) = \big(\sum_{1 \le i \le n} \widehat{r}(v_1, k_i) \big) \times p(v_1, k_1, k_2, \ldots, k_n) \tag{2.1}$$

A function value equal to 0 means the keywords are very far apart in v_1 and a function value equal to 1 stands for the case where the keywords are next to each other in v_1.

Even if the keywords appear in a small text window of the data structure, the result can still be meaningless. To solve a problem like this, XSEarch [23] proposes a semantic-based keyword proximity measure that takes into account the nested structure of the XML documents. The standard *tfidf* formula is used to give weight to leaf nodes with keywords. XSEarch returns to the user subtrees of a document. The value that is given by the *tfidf* formula represents both the frequency of the keywords in the leaf node document under consideration and the inverse frequency of the keyword in all the documents of the leaf nodes. The final scores $w(k, n_1)$ are the normalized $tfidf$ value of the pair (k, n_1). If $w(k, n_1)$ is equal to 0, k does not appear in the node n_1. To evaluate how the results match with the keyword query, XSEarch uses the vector space model. Let L stand for the label set, and K stand for the keyword set. Every node n is associated with a vector V_n of size $|L \times K|$.

$$V_n[l,k] = \begin{cases} \sum_{n' \in N_leaf} w(k,n') & \text{if label(n) = l} \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

If the node n does not match with the label, the score will be 0. If the node n matches with the label l, the score will be the sum of the weights for keyword k in the node n'. The measure of similarity between a query Q and an answer N, denoted $sim(Q, N)$, is the sum of the distances between the vectors associated with the nodes in N and the vectors associated with Q.

2.4 Keyword Search on Relational Data

A huge amount of data has been stored in relational databases. The standard way to get data from a relational database is using SQL. Even though the SQL language is able to get all kinds of data that the user wants, it has the aforementioned problems of structured query languages. Many recent works have designed approaches which implement keyword search on relational databases.

2.4.1 Query Semantics

The traditional keyword search is on flat documents. It simply checks if the keyword exists or not in the document. Relational databases store the data in tables which are linked though key-foreign key relationships. A table is a set of tuples that have the same attributes. It usually represents to a set of objects. The keywords in the query might appear in different tuples in the tables of the database. In order to compute the answer to a keyword query, one has to find not only the matching tuple set for the keywords, but also the key-foreign key relationships between these tuple sets. The result of a keyword query is commonly defined as a graph or a tree, such that the nodes correspond to tuples that contain zero or more keywords of the query, and the edges correspond to key-foreign key relationships between these tuples.

2.4.2 Candidate Network Generation

The connections between the tuples might be very numerous. There are two approaches to keyword search on relational databases. The first one is the tuple-based approach. The tuple-based approach actually links the matched tuples for each keyword. The second approach is the schema-based approach. The schema is a high level summarization of the database and shows the structure information for the tables. In the tuple-based approach, a relational database is a large data graph. The nodes are tuples and the edges are key-foreign key relationships. Given a query $Q = k_1, \ldots, k_n$, a query result is a connected subgraph of the database graph. This subgraph contains all or some of the keywords of the query. The connections represent joins between different tables or self-joins. In order to limit the number of results, the size of the results is usually constrained to be below a given threshold. The schema-based approach does not use the data graph with the tuples, but uses instead the schema graph which contains relation schemas and key-foreign key relationships.

In the schema based keyword search, the results, called joining networks, represent a cluster of one or more result graphs. Most of the schema-based approaches require the result to be a minimal total joining network which means that it has to satisfy minimality and completeness properties.

The first algorithm to generate all minimal candidate networks (CNs) was proposed by the DISCOVER approach [42]. DISCOVER focuses on how to leverage the physical database design to build compact data structures critical for efficient keyword search over relational databases. According to DISCOVER, an association exists between two keywords if they are contained in two associated tuples. In candidate networks, this means that the two tuples are connected with each other through a sequence of key-foreign key relationships, which potentially contain several other tuples that do not contain a keyword. The minimality condition of DISCOVER postulates that the removal of a tuple that contains a keyword does not result in a graph which has all the keywords.

To generate the CNs, DISCOVER expands partial CNs to generate larger partial CNs until all CNs are produced. The CNs are expressed as a joining sequence. Join expressions are generated up to the size limit. DISCOVER prunes many of the non-valid partial CNs by exploiting the properties of the schema of the database. This reduction of the redundant intermediate results improves the efficiency of the algorithm. Unlike DISCOVER, DBXplorer [3] maintains a symbol table which only stores the list of the columns where the keywords occur. With this design, the approach can reduce the space requirement and improve search performance. The generation of CNs uses an enumeration strategy. After searching in the symbol table to match the keywords, the algorithm enumerates all the join trees. For each tree, it executes an SQL query to retrieve matching rows.

BANKS [14] is a tuple-based approach (which uses the tuple as nodes and cross-references between them as edges). It not only allows the users to use the keywords to search but also provides a rich interface to review the results. For generating the results, BANKS uses backward expanding search. Starting with the matching keyword nodes, it proceeds backwards to find a common root which connects the whole result. The backward search algorithm can explore an unnecessarily large number of the graph nodes, especially when there are frequently occurring keywords in the query. In the algorithm, each keyword has an iterator. If a node has a very large fan-in, the iterator may need to explore a large number of nodes. The algorithm of the tuple-based approach presented in [44] starts from potential roots of the results. The nodes on an iterator with small fringe would have a higher priority. Within a single iterator, those subtrees which are less bushy have a higher priority. These two methods avoid the wasteful expansion of CNs with large fringes.

2.4.3 Answer Ranking

The ranking method of DISCOVER [42] and DBXplorer [3] is pretty simple. These papers assume that if the keywords are close to each other in the result, the result is more relevant. Therefore, they rank the results in ascending order of the number of joins involved in the tuple tree T. Suppose T is a result join tree. And $size(T)$ stands for the number of joins. The Score of T if T contains all the keywords in Q as follows:

$$Score(T,Q) = \begin{cases} \frac{1}{size(T)} & \text{T contains all keywords in Q} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

In BANKS [14], the result ranking involved three kinds of scores: Node score $Nscore$, Edge score $Escore$, and combination score. The node weights depend upon the prestige of the node based on inlinks. In the implementation, it has been set to be equal to the indegree of the node. The similarity between database relations depends upon the type of the link between them. If there is no reference relation, it is set to be equal to infinity. The edge score reflects the strength of the proximity relationship between two tuples and is set to one by default. These two scores could be additively combined using the formula $(1-\lambda)Escore + \lambda Nscore$ or multiplicatively combined using the formula $Escore * Nscore^{\lambda}$. Both scores are normalized.

In DISCOVER II [41], the authors use a new IR style ranking method to rank the results. For each textual attribute which belongs to the tuple set, there is a single-attribute IR-style relevance score. This score is influenced by the keyword's appearance frequency in the attribute, and also by the number of the tuples in the attribute. This score function also can be easily extended to incorporate PageRank-style scoring. After the score for each of the attribute is obtained, all the scores are combined and divided by the size of the result. Because the size of the results shows how "tight" the whole result is, the smaller the better. Single-attribute IR-style relevance score $Score(a_i, Q)$ for each attribute a_i that belongs to T is defined as follows:

$$Score(a_i, Q) = \sum_{\omega \in Q \cap a_i} \frac{1 + ln(1 + ln(tf))}{(1-s) + s\frac{dl}{avdl}} ln \frac{N+1}{df} \tag{2.4}$$

Where, for a word w, *tf* is the frequency of w in a_i, *df* is the number of tuples in a_i's relation with word w in this attribute; *dl* is the size of a_i in characters, *avdl* is the average attribute-value size, N is the total number of tuples in a_i's relation, and s is a constant. The final score for T is the summary of all the a_i scores in T divided by the size of T.

The IR style model has an inherent problem. It might be overly rewarding contributions of the same keyword in different tuples in a join tuple tree. Spark [61], another relational database search approach, introduces a function that uses a new ranking formula by adapting existing IR techniques based on the natural notion of virtual document. It is more effective and matches better with human expectation. It is used on a relational database which has a full-text index and inverted indexes. This ranking approach models a join tuple tree as a virtual document. By adjusting the model, it naturally computes the IR-style relevance scores without using an esoteric score aggregation function.

$$score(T, Q) = score_a(T, Q) \cdot score_b(T, Q) \cdot score_c(T, Q) \tag{2.5}$$

$Score_a(T, Q)$ is the same as in Discover II [41] and it is the IR style component of the score. $Score_b(T, Q)$ is the score which reflects the completeness of the result as this is derived from the extended Boolean model [70]. $Score_c(T, Q)$ is the normalized size factor score.

2.4.4 Top-*k* Strategies

In most applications, users are more interested in the first few results in the ranked answer rather than the multitudinous totality of the results in the answer. Therefore, top-k strategies make the algorithms more efficient. Similar strategies are applied in the information retrieval domain [29] and in data mining [37].

The top-k strategies are based on a scoring function. The results are usually evaluated by multiple scoring predicates that contribute to the total object score. There are two main methods at the application level for top-k query processing. The first method assumes that the scoring function has m components $p_1, \ldots, p_m$ and all these components have a scoring range. If the score is over-estimated, which means that the result will not satisfy the top-k scoring range, the result will be cut off. If the number of results is over k in the final result, the exceeding results will also be cut off. The approach of [16] consults the database histograms to map a top-k query to a suitable multi-attribute range query. The region function $reg(q, d_q)$ contains all the possible tuples within distance d_q from point q. If there are less than k tuples in $reg(q, d_q)$, it chooses a higher value for d_q and restarts the process.

The other method uses indexes and materialized views. This approach uses more storage space but improves the response time. For example, in the onion technique [17], m dimension points are used to present the m components of scoring. The convex hull of these points is the boundary of the smallest convex region that encloses them. The onion technique will return the top results by searching the points of the out most convex hull until all of the top-k results have been found.

2.5 Do More with Keywords

2.5.1 Semantic Matching Strategies

Keyword queries usually return multiple results. These results have different meanings since the keywords in the query are associated with different nodes, or are combined into different structures. If the role (meaning) for each keyword in the query was known, it would be easier to match the user request and filter out irrelevant results.

In meaningful keyword search algorithm[45], it is observed that because of the result minimality requirement of DISCOVER [42], some relevant results might be missed. The authors address the problem by interacting with the user to identify the meaning (role) of each keyword in the query. A keyword might appear in multiple locations. In a relational database, the role of a keyword is related to the relations and attributes. By identifying the role of the keywords, the accuracy of the results is improved, and meaningful non-minimal results (which are missed by DISCOVER) are returned to the user. One way to do the keyword role selection is to use a user interface which allows the user to choose from the potential roles that the keywords might have. A short natural language description for each of the attributes in the relations is stored in the system. If the attribute contains a keyword in its values, the system will return the attribute description along with the keyword so that the user chooses the most relevant role for the specific keyword.

Another approach, introduced in metadata approach[12], assumes that the meaning/role of each keyword depends on the other keywords in the query. The authors observe that based on some known patterns of human behavior, we know that the order of keywords in the query is important and correlated keywords are typically placed closer to each other. This approach aims at finding the role of the keywords without prior access to the data instance. The keyword queries are translated to several SQL queries that show the possible semantic meaning of the keyword query. The results from these SQL queries represent the results of the initial keyword query. To give each keyword a quantitative semantic meaning, the approach uses a weight table that gives a score to each pair of keyword and database term. There are two weights: the *intrinsic*, and the *contextual* weights. The intrinsic weight is based on syntactic, semantic and structural factors. The contextual weight is used to measure the same factors albeit considering the mappings of the remaining query keywords.

Most of the approaches above require the results to contain all of the keywords or relative entities in the query. This requirement guarantees the completeness of the result, but it might also miss incomplete results which are closer to the meaning of the query than complete ones. To remedy this problem, Keyword++ [82] maps query keywords to matching predicates or ordering clauses. If one of the keywords is guaranteed to belong to one attribute, the keyword and the attribute will map to a predicate "Attribute value = ⟨value⟩". If the keyword is a number, it will be mapped to an ordering clause "order by ⟨attribute⟩ ⟨ASC|DESC⟩". This process makes the keyword query matching more flexible. In order to automatically map those keywords, the approach uses a baseline search interface which takes a query as input, and produces a list of entities as output. These entities are used to map the keywords. There are several entity search engines [2, 11, 21, 66] which return the entities relevant to the user query even when not all the query keywords have been matched.

2.5.2 A Cohesive Keyword Query Language

One of the goals of this dissertation is to use a novel keyword-based query language, called cohesive query language [30], to extract integrated information from multiple, heterogeneous, structured and semistructured data sources.

The cohesive keyword query language. In a cohesive keyword query, the user can specify cohesiveness constraints among the keywords. Cohesiveness constraints define a group of keywords and state that the instances of these keywords in the dataset should form a cohesive whole, that is, a unit in which the instances of the other keywords cannot occur. They partially relieve the system from guessing without affecting the user who can specify them naturally and effortlessly.

For example, consider the keyword query {`big data John Smith George Brown`} to be issued against a large bibliographic database. The user is looking for publications on `big data` related to the authors `John Smith` and `George Brown`. To express this request as a cohesive query she will write `((big data) (John Smith) (George Brown))` where parentheses are used to specify the cohesive groups `big data`, `John Smith` and `George Brown`. A cohesive group, say `(John Smith)`, indicates that `John` and `Smith` form a cohesive unit where the instances of the other keywords of the query `George`, `Brown`, `big` and `data` cannot penetrate to form cohesive units with either `John` or `Smith`. With a cohesive query the user will get more accurate results: the system will be able to filter out publications on `big data` by `John Brown` and `George Smith`. It will also filter out a publication which cites a paper authored by `John` Davis, a report authored by `George Brown`, a book on `big data` authored by Tom `Smith`, and an article on semistructured `data` authored by Ray `Brown`. These "results" are irrelevant, but none of the previous keyword search approaches are able to automatically exclude them from the answer of the query. Importantly, specifying cohesiveness constraints frees the system from searching for a multitude of irrelevant results and reduces, to a small fraction, the time needed to compute the query answer.

Cohesiveness constraints can be nested. For instance, the query `(Hadoop (John Smith) (citation (George Brown)))` looks for a paper on `Hadoop` by `John Smith` which cites a paper by `George Brown`. The cohesive keyword query language conveniently allows also for keyword repetition. For instance, the query `(Hadoop (John Smith) (citation (John Brown)))` looks for a paper on `Hadoop` by `John Smith` which cites a paper by `John Brown`. More generally, the syntax of a query Q is defined by the following grammar, where the non-terminal symbol G denotes a cohesive group, and the terminal symbol k denotes a keyword:

$$\begin{array}{rcl} Q & \rightarrow & (k) \mid G \\ G & \rightarrow & (S\ S) \\ S & \rightarrow & S\ S \mid G \mid k \end{array}$$

Expressivity vs. ease of use, data structure and data model indepenence. The cohesive queries have higher expressiveness compared to flat keyword queries. However, contrary to other keyword-based query languages which trade-off ease-of-use for expressiveness, the cohesive query language increases the expressive power without compromising on ease-of-use.

Cohesive semantics. The unique features of the cohesive query language arise from the particular semantics assigned to cohesiveness constraints: contrary to other query languages which extend flat keyword queries, cohesive semantics is not defined in relation to constructs of the dataset but in relation to keywords of the query. Consider, for instance, XSearch [23], which is one of the simplest keyword-based query languages, and a slight extension of keyword search. Label keywords in an XQuery query are to be mapped to internal nodes in the XML data tree, value keywords are mapped to leaf nodes, and terms involving label and value keywords are mapped to nodes related through an ancestor-descendant relationship.

Therefore, a query is interpreted in relation to dataset concepts: tree node labels, tree node values, internal and leaf nodes, ancestor-descendant relationships etc. This kills, of course, the independence of the query language from the data model, as the queries are not meaningful in the context of any other data model. Further, it requires from the user to know which keywords are expected to represent internal node labels, which are expected to be leaf node values and how these might be linked through descendant relationships. That is, the user needs to have a sense of the structure of the data.

Clearly, formulating queries even in this elementary extension of flat keyword search is not trivial and requires some understanding of tree organization—certainly not a task expected for a naive user. Thus, the benefits of keyword search mentioned above are substantially reduced or erased in older extended keyword query languages. On the other hand, a cohesiveness constraint in a cohesive query states that in any result of the query, the keywords of the corresponding cohesive keyword group form a cohesive unit which is not "penetrated" by the other keywords of the query. In other words, a keyword in a cohesive group of the query is semantically closer to the other keywords in the group than to any other query keyword outside the group. This interpretation of cohesive queries frees the user from having any knowledge of the structure and the type of the underlying data sources and allows him to effortlessly formulate his query in a natural way based on his intuition. Cohesive queries fully enjoy all the benefits of flat keyword search.

In order to evaluate cohesive queries over a dataset of a certain type, the intuitive semantics needs to be mapped to formal semantics for this type of data. This process involves providing rules based on structural and semantic information for this data type which allow the system to take decisions in the following triangular setting: given three keyword instances i_0, i_1 and i_2, in which one of i_1 and i_2 is semantically closer to i_0. It is important to note that this concept of semantic closeness is not equivalent to, and usually contradicts, structural proximity. A goal of this project is to provide semantics for the cohesive query language on relational databases.

CHAPTER 3

AN ALGORITHM FOR KEYWORD SEARCH ON GRAPHS

We present and experimentally evaluate in this chapter an algorithm for keyword search on directed graphs. As we show in the experiments, this algorithms improves substantially over previous ones by eliminating the generation of redundant intermediate results.

3.1 Data Model, Queries and Answers

Data Model. We consider a data graph $G = (V, E, l)$, where V is a set of nodes, E is a set of directed edges and l is a labeling function assigning distinct labels to all the nodes and edges in $V \cup E$. Labels on edges are useful for distinguishing two distinct edges between the same nodes. Set $labels(V)$ denotes the set of the labels of the nodes in V. Every node label l in $labels(V)$ is associated with a set $terms(l)$ of keywords from a finite set of keywords $\mathcal{K}$. The same keyword can appear in the sets $terms(l_1)$ and $terms(l_2)$ for two distinct node labels $l_1, l_2 \in labels(V)$. Figure 3.1(a) shows a data graph with six nodes. The labels and their termsets are shown by the nodes.

Queries and Query Semantics. A *query* is a set of keywords from $\mathcal{K}$. The answer of a query is defined based on the concept of query result. As is usual [41, 14, 44, 40, 45], we define the result of a query on a graph to be a tree.

Definition 3.1 (Query Result) *Given a graph G, a* result *of a query Q on G is a node and edge labeled undirected tree R such that:*

(a) Every node of R is labeled with a label from labels(V). A function keys on the nodes of R assigns to every node n in R a (possibly empty) set of keywords from $Q \cap terms(label(n))$. Function keys satisfies the following two conditions:

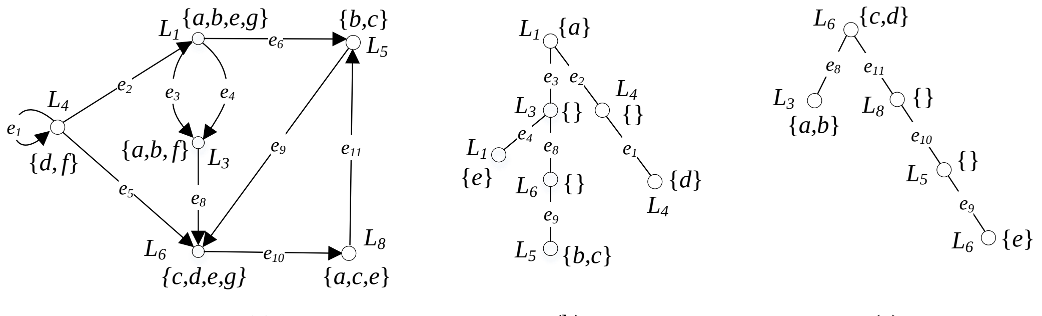


Figure 3.1 (a) Data Graph, (b) and (c) Results for the keyword query $\{a, b, c, d, e\}$.

(i) for every two nodes $n_1, n_2 \in R$,

$keywords(n_1) \cap keywords(n_2) = \emptyset$ (unambiguity), *and*

(ii) $\bigcup_{n \in R} keywords(n) = Q$ (completeness).

(b) If there is an l*-labeled edge between nodes* n_1 *and* n_2 *in* R*, then there is also an* l*-labeled edge between nodes labeled by* $label(n_1)$ *and* $label(n_1)$ *in* G*.*

(c) There is no leaf node (i.e., a node with only one incident edge) n *such that* $keywords(n) = \emptyset$ (minimality).

Figures 3.1(b) and (c) show two results for the keyword query $\{a, b, c, d, e\}$ on the data graph of Figure 3.1(a). The keyword set of every node is shown by the node. Observe that label L_4 appears twice in the result tree of Figure 3.1(b), and in one occurrence the corresponding node has an empty keyword set.

We can now define the answer of a query.

Definition 3.2 *The* answer *of a query* Q *on a data graph* G *is the set of results of* Q *on* G*.*

The results of a keyword query Q represent different interpretations of Q. Function *keywords* assigns a meaning to a keyword in Q by associating it with a result node whose label l contains this keyword in its term set $term(l)$. A keyword can appear in the keyword set of only one of a result to guarantee that only one meaning is assigned to each query keyword in a query result. This is the unambiguity property of a query result. We adopt AND semantics [42] for queries: all the keywords of Q should appear in a result of Q. This is guaranteed by the completeness property of a query result. The minimality property guarantees that a query result does not have redundant nodes.

Note that this definition of a query result is general: the result does not need to have an embedding to the data graph. That is, it does not need to be a subtree of the data graph (though it can be restricted to be). As a consequence, the same label can label multiple nodes in the result. A node n in a result which does not have any keyword associated to it (that is, $keywords(n) = \emptyset$) is called *empty* node. Based on the definition above, a query result cannot have empty nodes as leaf nodes.

A result tree can be of unrestricted size. Given a result tree R, $size(R)$ denotes the number of nodes in R. To constraint the number of results, it is common to restrict the size of the result tree. This is a meaningful constraint since results which bring the keywords closer are commonly assumed to be more relevant [42, 59, 31]. It is also required for performance reasons since the number of results can be very large. Other constraints can also be used for performance and/or results relevance reasons.

3.2 A Canonical Form for Result Trees

In this section, we leverage results obtained in the field of tree pattern mining to define a canonical form for result trees and to check whether a result tree is in canonical form.

Our algorithm for computing query answers constructs query result trees starting from a root node. Therefore, query results are generated as rooted trees. In the rest of the discussion in this section we focus on rooted trees.

Unordered and Ordered Trees. Rooted result trees are unordered trees. However, the trees that are generated by all the algorithms are ordered since a

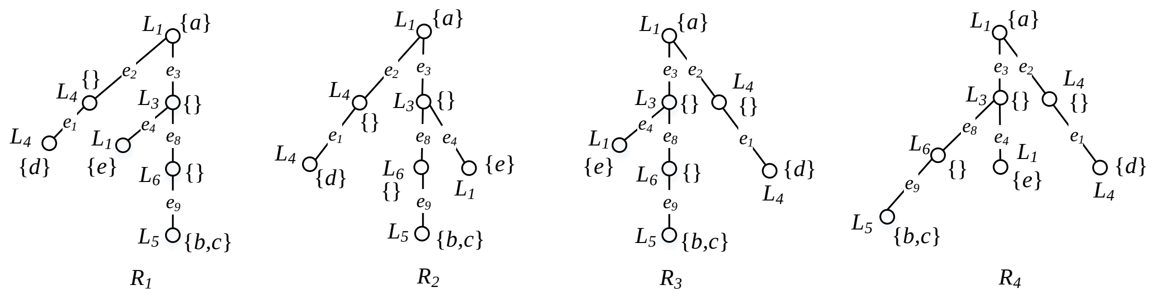


Figure 3.2 Four isomorphic result trees.

representation order is imposed to the nodes. Two labeled rooted trees R and S are *isomorphic* to each other if there is a one-to-one mapping from the nodes of R to the nodes of S that preserves node labels, edge labels, adjacency and the roots. An *automorphism* is an isomorphism that maps a tree to the same unordered tree. Figure 3.2 shows four isomorphic results trees R_1, R_2, R_3 and R_4 which form one automorphic group.

To reduce the generation of isomorphic trees (which are redundant since they represent the same unordered tree) and produce only one tree for every automorphic group we use a canonical form for result trees. In the rest of this section, we consider ordered trees. Given a node n in a tree R, let $root(R)$ denote the root of R and $subtree(n)$ denote the subtree of R rooted at n.

An Order for Labels and Trees. We first define an order for trees. Let $\leq$ be a linear order on the label set $label(V)$. Abusing notation, we also denote as $\leq$ an order on trees defined recursively as follows:

Definition 3.3 (Tree Order) *Given two trees R and S, let $r = root(R)$ and $s = root(S)$, respectively. Let also $r_1, \ldots, r_m$ and $s_1, \ldots, s_n$ denote the list of the children of r and s, respectively. Then, $R \leq S$ iff either:*

1. *$label(r) \leq label(s)$, or*
2. *$label(r) = label(s)$ and either:*

   (i) $m \geq n$ and $\forall i \in [1, n], subtree(r_i) = subtree(s_i)$, or

(ii) $\exists k \in [1, min(m,n)]$ *such that:* $\forall i \in [1, k-1], subtree(r_i) = subtree(s_i)$ *and* $subtree(r_k) \leq subtree(s_k)$.

The tree order above is essentially the same as the one introduced in [67, 88]. In the context of result trees, in order to take into account the labels of the edges in result trees, we view a labeled edge as two unlabeled edges incident to a node which has the label of the edge. Further, in order to take into account the keyword sets of the result tree nodes, we assume that the label of a result tree node n is the concatenation of the label $label(n)$ and of the keywords in $keywords(n)$ (if any) in alphabetical order. For instance, for the result trees of Figure 3.2, one can see that $R_1 \leq R_2 \leq R_3 \leq R_4$.

A Tree canonical Form. We can define now the canonical form adopted for a tree [67, 22, 88].

Definition 3.4 (Tree Canonical Form) *A tree R is in* canonical form *if for every tree S which is automorphic to R, $R \leq S$.*

As an example, consider the four trees of Figure 3.2. These are all the automorphic representations for the corresponding unordered tree. As $R \leq S$, for every tree S in the automorphic group, R is in canonical form.

To check whether a tree is in canonical form, the following proposition [67, 88] can be used.

Proposition 1 *A tree R is in canonical form iff for every node n in R, $subtee(n_i) \leq subtree(n_{i+1})$, $i \in [1, m-1]$, where $n_1, \ldots, n_m$ is the ordered list of children of n.*

Therefore, we can check whether a tree is in canonical form if we can decide about the order of a given pair of trees.

Exploiting a String Representation for Trees. Checking the order of a pair of trees can be done efficiently leveraging a string representation for trees. One such representation can be produced through a depth-first preorder traversal of the tree by adding the node label to the string at the first encounter and by adding a special symbol, say \$, whenever we backtrack from a child to a parent node [87, 22]. For instance, the string representation for the tree of Figure 3.1(c) (ignoring, for simplicity, edge labels and the node keyword sets) is $L_6L_3\$L_8L_5L_6\$\$\$$. Let's extend the linear order $\leq$ defined on labels to include \$ and let's assume that every label precedes \$ in this order.

The string representation for the trees is useful for checking the order of trees because of the following proposition [67, 88]:

Proposition 2 *Given two trees R and S, $R \leq S$ iff $string(R) \leq string(S)$.*

Therefore, well known string comparison algorithms and implementations can be employed to efficiently check the order of two trees.

3.3 The Algorithm

Our algorithm for computing keyword query answers takes as input a data graph G, a keyword query Q and a size threshold T for the query results. It starts by choosing randomly a keyword k from the input keyword query Q and by constructing intermediate result trees with a single node. For every node labeled by L_n in the data graph which contains k in its term set one node n labeled by L_n is constructed. Node n is associated with keyword sets and the resulting single node intermediate result trees are pushed into a stack. The keyword sets $keywords(n)$ associated with n: (a) contain keyword k and (b) are subsets of the set of $terms(L_n)$ of label L_n in the data graph, and of the keyword query Q. The query results are constructed by expanding these initial single node intermediate results.

Unconstrained expansion. In the absence of any expansion constraint, the expansion process can be outlined as follows: a node n labeled by L_n is expanded by adding an adjacent node m labeled by L_m if the node labeled by L_m in the data graph is adjacent to the node labeled by L_n. The set of keywords of m is selected from the set of terms of the node L_m of the data graph which: (a) appear in the keyword query and (b) do not appear in the keyword set of other nodes of the intermediate result tree. The set of keywords of a node can also be empty. All the nodes in the intermediate result can be considered for expansion and a node is expanded in all possible ways. Intermediate results which contain all the keywords of Q in the keyword sets of their nodes are characterized as final results and are returned to the user if they do not have empty leaf nodes (minimality condition of a query result). If they do not satisfy the minimality condition, they are discarded. Intermediate results whose size is equal to the given size limit and are not final are not expanded anymore and they are discarded. Note that this process will generate duplicate results (isomorphic ordered versions of unordered trees). Hence, the results will have to be compared for isomorphism with previous results before returned to the user if an answer without duplicates is sought.

The expansion process can be constrained without missing any results. We show below how this can be achieved.

Considering only intermediate result trees in canonical form. For the result trees, it is sufficient to have result trees in canonical form (the other isomorphic versions of the result are redundant). It turns out that intermediate results which are not in canonical form do not need to be considered in the expansion process. This is shown by the next proposition.

Proposition 3 *All the results of a query on a data graph can be computed by considering, during the expansion process, only intermediate result trees in canonical form.*

Proof: For every result tree, only the ordered result tree from its automorphic group which is canonical needs to be returned to the user. Based on the discussion of Section 3.2 (Propositions 1 and 2), one can see that the prefix of any result tree in canonical form is also canonical. Hence, any canonical result tree can be produced by a sequence of expansions of an initial one-node intermediate result tree where all the intermediate result trees are canonical.

Therefore, intermediate result trees which are not in canonical form can be discarded. Clearly, eliminating non-canonical intermediate result trees from the search space allows a significant reduction of the number of intermediate results generated. Note that the expansion of intermediate result trees can generate trees which are not in canonical form (even if the initial tree was in canonical form). We have presented in the previous section a technique for efficiently checking result trees for canonicity. We will show below how this technique can be further improved.

Empty node expansion. It can be further observed that if an intermediate result tree has an empty leaf node, it can be chosen for expansion without missing any results. This is shown by the next proposition.

Proposition 4 *All the results of a query Q on a data graph G can be computed by expanding only empty leaf nodes in the result trees under construction (if empty leaf nodes exist).*

Proof: Based on the minimality condition of Definition 3.1 a result tree cannot have empty leaf nodes. Therefore, only by expanding these nodes a result can be constructed.

Following this remark, if an empty node exists in an intermediate result, it should be chosen for expansion.

Rightmost path node expansion. By default, in the absence of an empty leaf node, all the nodes of intermediate result tree need to be expanded to guarantee the computation of all the results. The next proposition shows that many intermediate result tree nodes can be excluded from expansion without affecting the completeness of the result set.

Proposition 5 *All the results of a query Q on a data graph G can be computed by expanding only the nodes on the rightmost path of the result trees under construction.*

Proof: Certainly, any result tree can be constructed by adding nodes to the root node in an order that corresponds to a depth first preorder traversal of the tree. This sequence of expansions expands only nodes on the rightmost path of result tree under construction.

If this expansion strategy is followed, in most cases, most of the intermediate result tree nodes (those which are not on the rightmost path) are not expanded.

Checking the canonicity of result trees incrementally. In order to check whether a tree is in canonical form we can use Proposition 1 (and also Proposition 2 to perform the comparisons of trees efficiently). However, as mentioned above: (a) only trees which are in canonical form are expanded, and (b) only nodes in the rightmost path of a intermediate result tree are expanded. These remarks allow for an incremental checking of the canonicity of the newly generated intermediate result tree: only the trees rooted at the rightmost child of the expanded node n and the trees rooted at the rightmost child nodes of the ancestor nodes of n need to be compared to the tree rooted at their immediate left sibling. No other comparisons are needed.

Correctness of the Algorithm. The pseudo code for our algorithm is shown in Algorithm 1. This algorithm integrates in the unconstrained expansion described in the beginning of this section the three expansion constraints described subsequently. The algorithm is named CFS (for 'Canonical Form-based Search').

Theorem 1 *Algorithm CFS correctly computes all the results of a keyword query Q on a data graph G given a size threshold T.*

The correctness of the theorem results from the correctness of the expansion constraining Propositions 6, 7 and 8 and the fact that in the absence of expansion constraints the algorithm exhaustively expands all intermediate results.

Checking the canonicity of an intermediate result tree R can be done in $O(N)$, where n is the number of nodes of R. Let m be the number of nodes in the data graph G that contain the initially chosen keyword k in their term set, and d be the average degree of the nodes in G (number of incident edges). Since Algorithm CFS expands nodes in all possible ways through the incident edges of a node, its worst case complexity is $O(m(T-1)!d^{T-1})$, where T is the size threshold. The worst case complexity is of little help here: in practice, Algorithm CFS takes advantage of the canonical form and the result expansion constraints to substantially reduce the number of intermediate results and this feature allows for superior performance compared to its competitors.

3.4 Experimental Evaluation

We run experiments to evaluate the efficiency, the memory consumption and the scalability of our algorithm. We also compared with the techniques employed for computing all the results by two previous keyword search algorithms on graphs derived from relational databases. As mentioned in Chapter 2, these algorithms compute similar types of results on the data graphs. To allow for a fair comparison, given a keyword query, we generate a data graph to evaluate the query the way it is generated in the context of the previous algorithms from relational schema graphs. Our implementation was coded in Java. All the experiments reported here were performed on a workstation having an Intel(R) Core(TM) i7-7500 CPU @ 2.70GHz processor with 8GB memory.

Datasets. We used a benchmark and two relational databases to generate data graphs: the *TPC-H* benchmark database and the *IMDB* database. The *TPC-H*

Algorithm 1 ***CFS***

```
Input: data graph $G$, query $Q$, size limit $T$
Output: query results of $Q$ on $G$ of size up to $T$
1: $E := \emptyset$            /* $E$ is a queue of query results */
2: Chose a keyword $k$ in $Q$
3: for every node $n$ in $G$ s.t. $k \in terms(n)$ do
4:    for every set $K \in \mathcal{P}(terms(n) \cap Q)$ such that $k \in K$ do
5:       Construct a single-node result $R$ labeled by $label(n)$
6:       $keywords(R) = K$
7:       $enqueue(R, E)$
8: while $E \neq \emptyset$ do
9:    $R :=$ dequeue($E$)
10:   if $R$ has an empty leaf node $q$ then
11:      $ExpandList := \{q\}$
12:   else
13:      $ExpandList := \{$ nodes in the rightmost path of $R$ $\}$
14:   for every node $l \in ExpandList$ do
15:      for every node $m$ adjacent to the node labeled by $label(l)$ in $G$ do
16:         $D := (Q \cap terms(m)) - keywords(R)$
17:         for every set $K \in \mathcal{P}(D)$ do
18:            Add a node $p$ with $label(p) = label(m)$ and $keywords(p) = K$ and an
               edge $(l, p)$ to $R$.
19:            if $R$ is in canonical form and satisfies the structural constraints then
20:               if $keywords(R) = Q$ then
21:                  output $R$
22:               else
23:                  if $size(R) < T$ then
24:                     $enqueue(R, E)$
```

benchmark is a decision support database. The data are chosen to have broad industry wide relevance. The schema of the dataset[1] used here comprises eight relation schemas

[1]http://www.tpc.org/tpch/

Table 3.1 Queries on the (a) IMDB and (b) TPC-H Database

(a)

	Keywords
Q_1	cuadro, Alex
Q_2	cuadro, Alex, Rafael
Q_3	cuadro, Alex, Rafael, 173
Q_4	Musical, Anne, Bauman
Q_5	17485, Anne, Bauman, David
Q_6	Aabel, Steve, 352881, crime, Kodanda
Q_7	Brown, grandfather, Tony, Musical
Q_8	Anne, Adams, Rafael, Musical, cuadro, 1664
Q_9	Halloween, 2000, Musical, Comedy
Q_{10}	2012, David, Musical, Grandfather, Alex, Tony

(b)

	Keywords
Q_1	Supplier, clerk
Q_2	carefully, express
Q_3	truck, regular, customer
Q_4	Morocco, packages, return
Q_5	foxes, Brand, small
Q_6	return, spring, yellow
Q_7	Indian, Burnished, India, Brand
Q_8	Manufacturer#3, Manufacturer#4, large
Q_9	special, France, Brand#22, India
Q_{10}	yellow, green, Brand#23, ironic, clerk

and eight foreign keys. Its tables contain 866,602 tuples. The *IMDB* database is a repository of films, actors, reviews and related information. The schema of the dataset[2] used here comprises seven relation schemas and six foreign keys. Its tables contain 5,694,919 tuples. The data graphs used in the experiments are generated separately for each keyword query as explained below in the paragraph discussing the queries.

Queries. We generated different queries to evaluate on the datasets and we report on 10 of them for each dataset. They are displayed in Table 3.1. The queries have from one to six keywords. Given a dataset and a keyword query, the data graph to evaluate the query is an extension of the schema graph of the relational database generated as follows: for every relation R in the schema of the relational database, additional nodes are added to the schema graph for every combination of query keywords appearing in a tuple of relation R which does not have other query keywords. Edges are added between these nodes mirroring the key-foreign key edges between the corresponding relations in the database schema.

[2]https://relational.fit.cvut.cz/dataset/IMDb

Statistics about the queries including the total number of keyword instances in the generated graphs (#Ins) and the number of results (#Results) are provided in Tables 3.2. The number of nodes and edges of the generated (extended schema) data graphs for every query are also shown in these tables in the columns '$\#GNodes$' and $\#GEdges$, respectively.

Algorithms in comparison. We compared our algorithm with the algorithm designed for the $DISCOVER$ system [42] (adopted also in other systems [41, 55]). We also compared with the meaningful keyword search algorithm used in [45]. We refer to them as *DISC* and *MEAN*, respectively. Both of them search on graphs which are extended relational schemas constructed, for every query, as described above in the paragraph "Queries". The DISCOVER approach imposes some additional constraints on the query results which are ignored here to secure the computation of similar results by all three algorithms. Our interest on these algorithms in on the way they guarantee result sets without duplicates (that is, without isomorphic ordered result trees). Different approaches are followed by these two algorithms. Algorithm *DISC* computes the results and removes duplicate results at the end with a post-processing step to guarantee that there are not duplicates. Algorithm *MEAN* is similar to *DISC* but excludes duplicates by comparing intermediate results, while they are generated, with previously generated intermediate results. For the needs of the

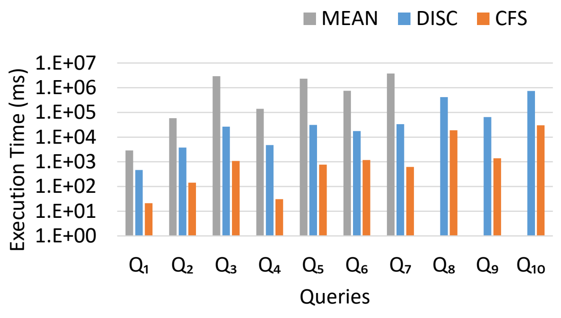
Table 3.2 Statistics for the Queries on the (a) IMDB and (b) TPC-H Database

(a)

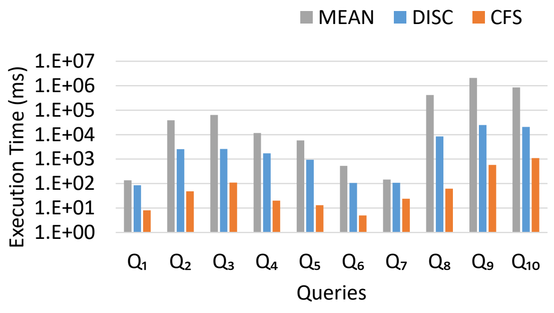
	#GNodes	#GEdges	#Ins	#Results
Q_1	13	19	6	89
Q_2	17	27	10	883
Q_3	17	27	10	6695
Q_4	20	30	13	1238
Q_5	27	43	20	9147
Q_6	24	46	14	411
Q_7	26	45	19	8802
Q_8	32	55	25	27837
Q_9	31	54	24	12086
Q_{10}	34	59	27	52515

(b)

	#GNodes	#GEdges	#Ins	#Results
Q_1	10	12	2	14
Q_2	30	54	22	618
Q_3	20	32	12	267
Q_4	19	31	11	45
Q_5	18	26	10	47
Q_6	12	13	4	19
Q_7	13	15	5	12
Q_8	29	51	21	1
Q_9	30	56	22	2337
Q_{10}	35	58	27	408



(a)



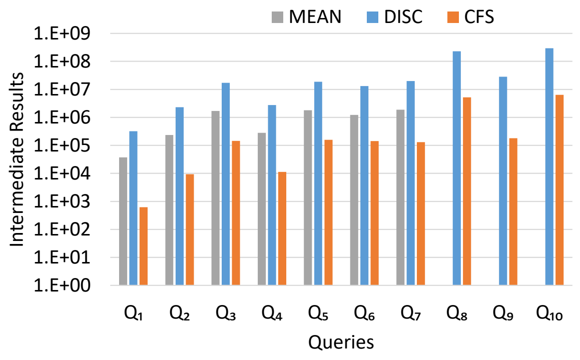
(b)

Figure 3.3 Execution time for the queries on the (a) IMDB, and (b) TPC-H database.

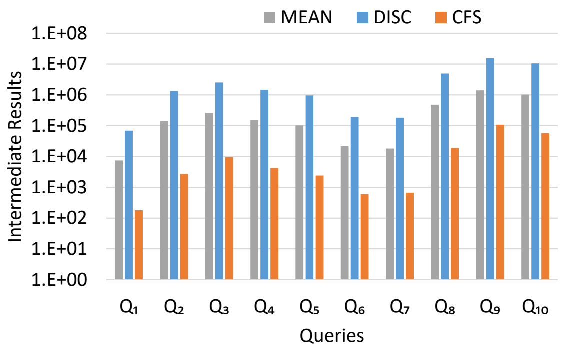
comparison, it assigns an ID to every generated tree based on tree isomorphism during the execution of the algorithm. The ID of a tree is compared with the IDs of the trees that are generated so far and the current tree is accepted only if it has not been generated previously. Algorithm *MEAN* computes top-k results. Therefore, for the needs of the comparison we assume that k is larger than the number of the query results so that all results are computed. We refer to our algorithm as CFS.

Experiments on the execution time. In our first experiment we measured the efficiency of our algorithm in terms of execution time and we compared these numbers with those achieved by $DISC$ and $MEAN$. The size limit was set to 6 edges. Figure 3.3 shows the execution time of the algorithms on the IMDB and the TPC-H datasets, respectively. Note that the scale of the y-axis is logarithmic.

One can see that Algorithm *CFS* is much faster than both *DISC* and *MEAN*. Algorithm *CFS* outperforms *DISC* by at least one order of magnitude in all cases. Algorithm *MEAN* is much slower than the other two. It is two orders of magnitude slower than *CFS* in most cases and in several cases slower by more than three orders of magnitude. Note than in several cases, *MEAN* could not finish within a reasonable amount of time and



(a)



(b)

Figure 3.4 Number of intermediate results produced for the queries on (a) the IMDB database, and (b) the TPC-H database.

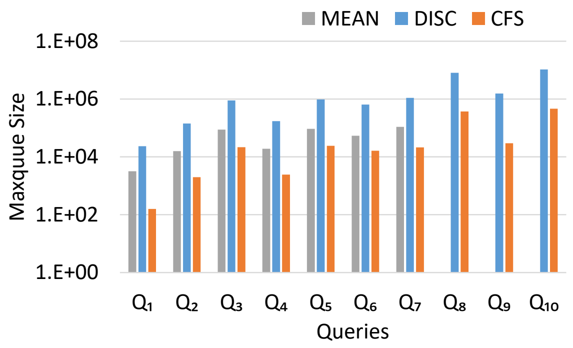
therefore, no value is displayed in the plots for the corresponding queries (queries Q_8, Q_9 and Q_{10} on IMDB).

Experiments on the number of intermediate results. In this experiment we measured the performance of the algorithms in terms of the number of intermediate results generated. Figure 3.4 shows the number of intermediate results produced on the IMDB and TPC-H databases, respectively. The intermediate results are the partial or complete result trees generated by the algorithms. Algorithm *DISC* pushes all the intermediate results into the queue maintained by the algorithm as long as they satisfy the size constraint. Algorithm *MEAN* pushes intermediate results into the queue if they satisfy the size constraint and no isomorphic intermediate result is found in the queue. Finally, algorithm *CFS* pushes intermediate results into the queue if they are in canonical form and satisfy the size restriction.

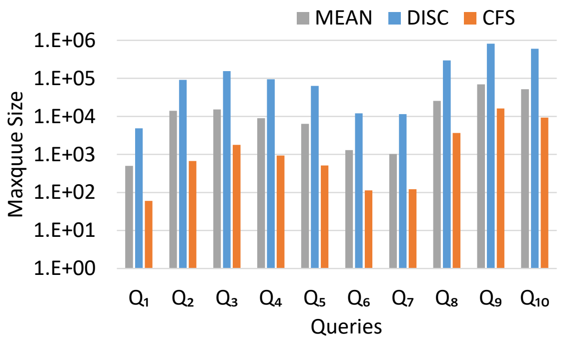
As one can see, algorithm *CFS* produces the smallest and *DISC* the largest number of intermediate results in all cases on both datasets. Algorithm *DISC* produces at least one and in most cases two orders of magnitude more intermediate results than *CFS*. The number of intermediate results produced by *MEAN* falls between the numbers produced by the other two algorithms. This is expected since *DISC* does not filter intermediate results and eliminates duplicate results only at the end through a post-processing step. *MEAN* filters intermediate results by comparing with other intermediate results stored in the queue while *CFS* filters intermediate results by producing only trees in canonical form. However, *DISC* and *MEAN* generate result trees by expanding all the nodes of a given intermediate result tree while *CFS* expands preferably an empty node and nodes in the rightmost path of an intermediate result tree. The sophisticated intermediate result expansion method of *CFS* and its intermediate result pruning technique based on canonical form explain the superior performance of *CFS*.

We can also observe that for a given algorithm, the execution time and the number of intermediate results are very closely correlated metrics. However, the number of intermediate results does not determine the execution time among different algorithms. Algorithm *CFS* produces less intermediate results than *DISC* and it is also faster. On the other hand, *MEAN* produces less intermediate results than *DISC* and it is much slower. The reason is that *MEAN* has to compare every qualified intermediate result with all the intermediate results currently stored in the queue. As the next experiment shows, there can be many intermediate results in the queue and this comparison can take a lot of time. This explains the inferior time performance of *MEAN* compared to *DISC*.

Experiments on memory consumption. We also measured the memory consumption of the three algorithms in terms of the maximum number of intermediate results put in the queue. Figure 3.5 shows the measured values. In order to minimize the memory footprint, in every algorithm, every occurrence of the query keyword selected to initiate the algorithm is added to the queue as a single node intermediate result only after all the intermediate

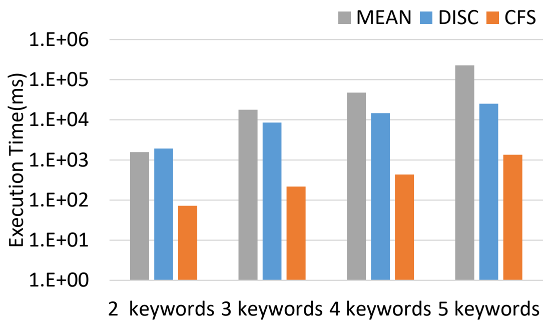


(a)

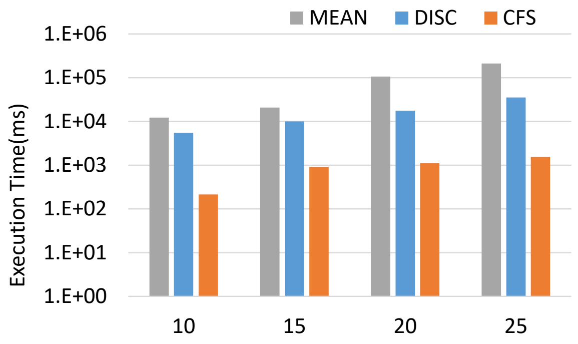


(b)

Figure 3.5 Memory consumption for the queries on the (a) IMDB and (b) TPC-H database.



(a)



(b)

Figure 3.6 Average execution time vs. number of keywords and (b) Average execution time vs. number of keyword instances.

results from the previous single node intermediate result have been processed and the queue is emptied.

As we can see *DISC* consumes one to two orders of magnitude more memory on the average than *CFS*. The difference between *MEAN* and *CFS* is not so pronounced. It is on the average less than one order of magnitude in favor of *CFS*. This difference is not expected since *MEAN* compares every produced intermediate result against all stored intermediate results and discards it if is found there. Nevertheless, *MEAN* expands intermediate results using all the nodes in the result tree as opposed to *CFS* which restricts this expansion to the empty node or the nodes in the rightmost path of the tree. This is the reason of the superior performance of *CFS* over *MEAN*.

Scalability Experiments. We also measured how the execution time evolves when the number of query keywords and the number of keyword instances increases. For this experiments we used a synthetic data set whose schema is similar to the IMDB data set. We generated ten queries each with 2, 3, 4 and 5 keywords and measured the average execution time when the number of keywords in the query increases. The results are shown in Figure 3.6a. We also considered ten five-keyword queries and we increased the total number of instances of each query from 10 to 25. The measured average execution time when the total number of instance increases is shown in Figure 3.6b. It can be observed that overall, *CFS* scales smoother than the other two approaches as it avoids the expensive comparisons for removing isomorphic intermediate and final results.

3.5 Conclusion

We have addressed the problem of efficiently evaluating keyword queries on graph data. We observed that existing algorithms generate numerous intermediate results and this negatively affects their execution time and memory consumption. We defined a canonical form for result trees and we showed how it can checked incrementally and efficiently. We devised rules that prune the intermediate result search space without sacrificing completeness. These rules were integrated in a query evaluation algorithm. Our experimental results show that our approach largely outperforms previous ones in terms of execution time and memory consumption and scales smoothly when the number of keywords and the number of keyword instances increases.

CHAPTER 4

LEVERAGING SCHEMA INFORMATION FOR SEMANTIC KEYWORD SEARCH OVER STRUCTURED DATABASES

A significant amount of the world's data is stored or exported in a form that exhibits some kind of (loose or tight) structure. For instance, a large part of corporate data resides in relational databases, popular NoSQL databases are organized as key-value stores, publicly available datasets can be downloaded in XML or JSON format, DBPedia is exported in RDF format, and social media connections are represented as graphs.

In recent years, a lot of researcher has been done on keyword search over structured data. There are different reasons for this strong interest of the database and IR communities on this subject. First, the users can retrieve information without mastering a complex structured query language (e.g., SQL, XQuery [15], SPARQL [68]). We call this benefit simple user emancipation. Second, they can issue queries against the data without having full or even partial knowledge of the structure (schema) of the data source. We call this second benefit data structure independence. Third, they can query different data sources in an integrated way: the same query can be issued against and extract information from multiple data sources which might structure their data differently. This is particularly important in web and big data environments where the data sources do not necessarily have the same structure/schema. We refer to this third benefit as data structure independence.

The problems. There is a price to pay for the simplicity, convenience and flexibility of keyword search. Keyword queries are imprecise and ambiguous in specifying the query answer. They lack expressive power compared to structured query languages. Consequently, they generate a very large number of candidate results. This is a typical problem in IR. However, it is exacerbated in the context of structured data. Indeed, in this context the result to a keyword query is not a whole document but a data fragment (e.g., a subtree, or a subgraph) and this exponentially increases the number of results. This weakness incurs two major problems.

The first problem is that the correct identification of the relevant results among a plethora of candidates becomes a very difficult task. Indeed, it is practically impossible for a search system to "guess" the user intent from a keyword query and the structure of the data source. Although previous approaches are intuitively reasonable, they are sufficiently ad-hoc and are frequently violated in practice resulting in low-quality results. We refer to this problem as a query answer quality problem.

The second problem is that existing algorithms for keyword search are of high complexity and cannot scale satisfactorily when the number of keywords and the size of the input dataset increase. We refer to this problem as the performance scalability problem. Note that top-k processing algorithms alone [41, 40, 61, 78, 38, 49, 83, 18, 35, 65, 31] do not solve the performance scalability problem as they still generate, in most cases, a large number of results (before identifying the top-k) or rely on specialized indexes which cannot be assumed to be available in practice.

One way to address these problems by bridging the gap between structured query languages and keyword search on datasets with structure is exploratory search [53, 54]. Though, this process requires interaction with the user.

These problems, which are inherent to keyword search have hindered the widespread use of keyword queries over data with structure. Without additional information, a flat keyword search cannot efficiently provide accurate answers on structured databases.

Contribution. In this chapter, we consider as a paradigm for structured databases the relational model. We focus on addressing the problems mentioned above in order to improve the efficiency and quality of keyword search on relational databases. We propose to exploit schema information as semantic information in order to improve the quality and efficiency of keyword search on relational databases. A novelty of our approach is that contrary to traditional approaches, query results include schema information and can involve relationships between schema components, between tuples of the same or different relations and between schema components and tuples. We cluster the results of a keyword query using the concept of "query pattern graph". Query pattern graphs summarize a set of query results which display the same structural and semantic information. They involve schema and/or data components and represent a possible structured interpretation of the flat keyword query on the structured database. The use of query pattern graphs scales down the number of query results we have to deal with in the first place. As they can be expressed as a structured query (e.g., an SQL query), all the machinery and optimization techniques developed for relational query engines can be used to efficiently compute the query results that a query pattern graph summarizes. We develop rules for scoring (and therefore, ranking) query pattern graphs and we further develop techniques for efficiently evaluating top-k query pattern graphs.

The main contributions are the following:

- We exploit semantic information to define the results of keyword queries on relational databases as graphs that involve both schema and data components and relationships between them. Using semantic information allows us to address the query answer quality problem.

- We introduce query pattern graphs which cluster together query results with the same structural and semantic characteristics. We focus on computing the query pattern graphs of a query on a database which are much fewer than the results corresponding to the matches of this query on the database.

- We provide rules for scoring the pattern graphs of a keyword query on a relational database. The scoring exploits semantic information for assigning scores to the edges of a query pattern graph and employs IR techniques for assigning scores to nodes.

- We design an efficient top-k algorithm for computing the pattern graphs of a keyword query. Our algorithm uses a canonical form for pattern graphs to avoid the computation of redundant intermediate results which are at the heart of the performance scalability problem.

- We experimentally evaluated the effectiveness of our approach and the time performance, memory consumption and scalability of our algorithm on two real datasets. The experiments confirmed the quality of our answers and the feasibility of our system.

Outline. The rest of the chapter is structured as follows. In the next section, we present the data and query model adopted and discuss the scoring of query pattern graphs. In Section 4.2, we outline our top-k evaluation strategy of keyword queries. In Section 4.3, we present and analyze our effectiveness and efficiency experimental results.

4.1 Semantic Search

We present now our approach for semantic keyword search by introducing query pattern graphs and describe our techniques for ranking query pattern graphs.

4.1.1 Data Model, Queries and Answers

We assume that a *relational database* D with *schema* S is given. The schema S can comprise key-foreign key relationships from a set of attributes of a relation to the primary key of another or the same relation thus forming a graph where the nodes are the relation schemas and the edges are the key-foreign key relationships. A *keyword query* on D is a set of keywords. The keywords can match attribute values in the tuples of the relation instances or schema elements (relation names and/or attribute names).

Figure 4.1 shows the schema graph of a simplified IMDB database that we use as an example. The nodes of the schema graph are annotated with possible matches of the keywords of the query $Q = \{movie, Pompeii, legend, actor, name\}$. For instance, we can see that for the node Movie, the keywords of Q can match the relation name "Movie", the attribute "name", the value "Pompei" of attribute Name in a tuple and the value "Legend" of attribute Name in another tuple. Such an annotated schema graph can be used for computing the query pattern graphs of query Q on the database D.

Query pattern graphs. In order to compute the results of a keyword query, we use the concept of query pattern graph defined below.

Definition 4.1 *A* query pattern graph *of a keyword query Q is a connected node and edge labeled undirected graph G_q, such that:*

(a) The nodes of G_q are partitioned into two types of nodes: schema nodes *and* tuple nodes. *Schema nodes are labeled by a relation name and schema. Tuple nodes are*

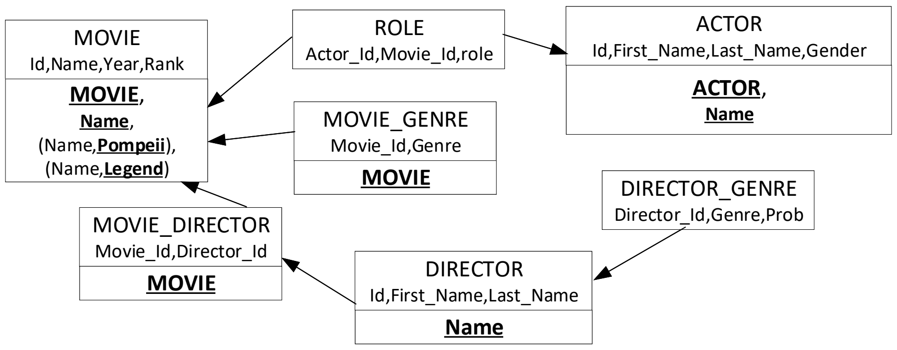


Figure 4.1 The schema graph of the IMDB database annotated with the query $Q = \{$movie, Pompeii, legend, actor, name$\}$.

labeled by a relation name and a tuple of that relation which might be full, partially full or empty. The non-empty fields are filled by keywords. The empty fields act as variables to be matched against values in the relation instance.

(b) *There are two types of edges:* inclusion edges *and* foreign key *edges. Inclusion edges are between a schema node and a tuple node of the same relation. Inclusion edges are labeled by the symbol $\in$. There is an inclusion edge between every schema node and a tuple node of the same relation. Foreign key edges are between two tuple nodes of the same or different relations. If the tuples come from different relations R and S, there should be a primary key-foreign key relationship between R and S in the schema graph. If the tuples come from the same relation R, there should be a recursive foreign key on R in the schema graph. A foreign key edge is labeled by the corresponding primary and foreign keys of the involved relations.*

(c) *For every keyword k in Q there is at least one term in G_q (relation name, attribute name, or attribute value) which matches k. For every keyword k in Q exactly one matching term in G_q is marked. Marked terms are shown in bold or underlined in the figures.*

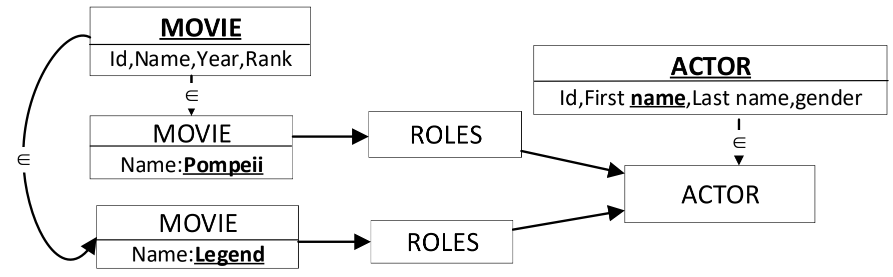


Figure 4.2 A pattern graph for the query $Q = \{$movie, Pompeii, legend, actor, name$\}$ on the IMBD database.

(d) Schema nodes appear in G_q only if they have at least one marked term. A schema node for a relation can appear only once in G_q. If the schema nodes and their incident edges are removed from a pattern graph, the resulting graph is a forest. A tuple node without a marked term is called empty *tuple node. Any external node in the trees of the forest that is not connected to a schema node in G_q and any single-node tree should be non-empty (*minimality condition*).*

We adopt AND semantics for keyword queries. The presence of every keyword of Q in a pattern graph guarantees the *completeness* of the results.

Figure 4.2 shows a pattern graph for the query $Q = \{movie, Pompeii, legend, actor, name\}$ on an IMDB database whose annotated schema is shown in Figure 4.1. A query pattern graph represents an interpretation for the flat keyword query. For instance, this query pattern asks for the names of authors who played some roles in the movies “Pompei” and “Legend”. Note that it is possible that other terms in a query pattern graph besides the marked terms can match a keyword. For example, keyword `name` in the pattern graph of Figure 4.2 matches an attribute of relation `Actor` but it matches equally well the attribute `Name` in relation `Movie`. The marked term indicates the interpretation this query keyword has in this query pattern graph.

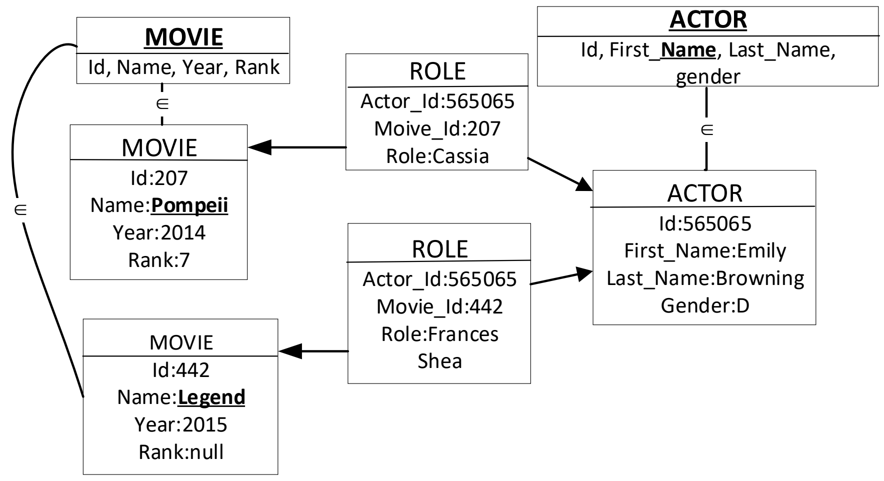


Figure 4.3 A result graph for the query {movie, Pompeii, legend, actor, name}.

A query can have multiple pattern graphs. SQL queries can be used to produce the corresponding result graphs. All the machinery developed over the years for optimizing SQL queries can be used for producing the results of patterns graphs when the data are stored in a relational database. For instance, the result graph of Figure 4.3 can be produced from the pattern graph of Figure 4.2. The pattern graphs of a query cluster the results of a query. Our goal is to return to the user the pattern graphs instead of the individual results as there are far fewer pattern graphs than individual results.

Traditional keyword approaches on relational databases cannot handle effectively queries which contain keywords matching schema elements and keywords matching data elements. The approach in [12] aims at viewing keywords which match schema elements as role descriptions of other elements but assumes that the instance of the database is not available and only guesses keyword-to-data matches. In our approach, we capture associations of value keyword matches with relation name and attribute keyword matches in a query pattern graph with inclusion edges.

Our goal is to design scoring functions to rank the pattern graphs instead of ranking the individual results. A scoring function will select the most promising among them.

4.1.2 Query Pattern Graph Ranking

In order to rank the query pattern graphs of a query on a database, we use semantics-based techniques to assign scores to the connections between the nodes that contain matching keywords. We take into account both the structure and the nature of these connections (e.g., whether a keyword matching an attribute in a schema node is connected to a keyword matching a value of this attribute in a tuple node, or whether a keyword matching a relation name in a schema node is connected to an empty tuple node). We refer to this type of scoring as *edge scoring*. For the ranking, we also assign a score to a pattern graph, by considering its nodes. We refer to this type of scoring as *node scoring*. For this process we employ Information Retrieval (IR) techniques [9] adapted to the relational model. IR-based techniques adapted in different ways to the relational model have been used frequently in the past for ranking the results of keywords on relational databases [41, 55, 40, 61, 50]. We argue that this is not sufficient for effective keyword search on relational databases and, although we leverage also IR style techniques, we focus on exploiting semantic information as this is expressed by the connections between different relational schema and data components in query pattern graphs. The pattern graphs of a query are ranked based on their edge score. If there is a tie, the node score is used to rank the pattern graphs with the same edge scores.

Edge Scoring In order to compute the edge score $ES(G_q)$ of the pattern graph G_q of a query Q we proceed by gradually removing schema nodes and the incident inclusion edges from G_q and by merging tuple nodes and foreign key edges. The process starts by removing schema nodes and terminates when one node (or a collection of disconnected nodes) is left. At every step (removal of a schema node) a score is computed which is added to the edge score. At the beginning, the edge score $ES(G_q)$ of G_q is initialized to 0. Some steps might add a zero score to the edge score but entail the computation of some parameters which ultimately affect the overall edge score. At every moment in the process, a number of nodes in the pattern graph are characterized as *keyword nodes*. All the nodes in the query pattern graph that contain a marked term (matched keyword) are keyword nodes. Some empty tuple nodes become (virtual) keyword nodes in the process (specifically those that are neighbors of a schema node). The different phases of the edge scoring process are described below.

Phase 1: Removal of schema nodes. In this phase all the schema nodes and their incident inclusion edges are removed from the query pattern graph. Note that if a schema node is present in a query pattern graph, it has at least one marked term (matching keyword). As mentioned in Definition 4.1, the graph resulting from the removal of schema nodes is a forest which contains only tuple nodes.

Given a schema node s, with neighbouring non-empty tuple nodes $t_1, \ldots, t_n$ and corresponding inclusion edges $e_1, \ldots, e_n$ a score $score(t_i)$ is assigned to every tuple node t_i and a score $score(e_i)$ is assigned to every edge e_i which are computed as follows: $score(t_i)$ is equal to $m-1$ where m is the number of marked terms in t_i. This score privileges pattern graphs where multiple keywords of the query are matched to terms in the same tuple. The value of $score(e_i)$ is the sum of the values of the parameter c_m for every marked term m in t_i, where c_m is determined by Table 4.1.

Note that the last case corresponds to the situation where the schema node s has a marked term which is an attribute and the tuple node has a marked term which is a value of a different attribute. The intuition behind the values of c_m is that a connection between a keyword which matches an attribute and a keyword that matches a value of this attribute is very strong and gets the highest possible score. A connection between a keyword which

Table 4.1 Different Values of Parameter c_m.

Condition for marked term m in schema node s and tuple node t_i	c_m
s has a marked term which is an attribute name and m is a value for this attribute	1
s has a marked term which is a relation name	0.5
None of the above	0.2

matches a relation name and a keyword that matches a value of an attribute of this relation is less strong. The connection between a keyword which matches an attribute of a relation and a keyword that matches a value of another attribute of the same relation is weak but not insignificant.

A score $score(s)$ is also assigned to the schema node s which is equal to $m-1$ where m is the number of marked terms in s. This score again privileges pattern graphs where multiple keywords of the query are matched to components (relation name and attributes) of the same schema node.

For every schema node s in G_q with non-empty tuple nodes $t_1, \ldots, t_n$ and corresponding inclusion edges $e_1, \ldots, e_n$, the sum

$$score(s) + \sum_{i=1}^{n} score(t_i) + \sum_{i=1}^{n} score(e_i) \tag{4.1}$$

is added to the edge score $ES(G_q)$ of G_q. If s does not have any non-empty tuple nodes, only $score(s)$ is added to $ES(G_q)$.

Example 1 *Consider the query pattern graph G_q of Figure 4.2. After the removal of the non-empty tuple nodes* `MOVIE`*, only the two corresponding inclusion edges contribute to the edge score of G_q by 0.5 each, and thus $ES(G_q) = 1$.*

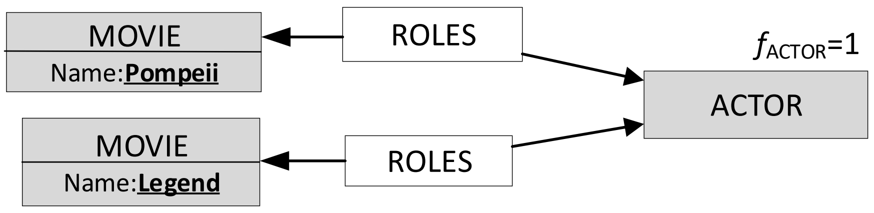


Figure 4.4 Pattern graph with schema node removed.

Given a schema node s, with neighbouring empty tuple nodes $t'_1, \ldots, t'_n$ no score is assigned to the empty nodes because of the removal of s from G_q in this phase. Instead, all the empty tuple nodes $t'_1, \ldots, t'_n$ become virtual keyword nodes and a factor $f_i,\ i = 1, \ldots, n$, is associated with each one of them. Factor f_i is equal to $1/m$, where m is the number of neighbouring (empty and non-empty) tuple nodes of the schema node s. This factor will be used to adjust the score the connection of this keyword node to another keyword node gets in the next phase of the edge scoring process. The empty tuple nodes are characterized as virtual keyword nodes because, even though they do not have a keyword, they are assumed to inherit the keyword of the neighboring schema node (which has at least one keyword) after the removal of the latter from the query pattern graph.

Example 2 *Consider again the query pattern graph G_q of Figure 4.2. After the removal of the empty tuple nodes* `ACTOR`*, the edge score of G_q is not modified ($ES(G_q) = 1$). The empty tuple node* `ACTOR` *becomes a virtual keyword node and the resulting tree is shown in Figure 4.4. The keyword nodes (regular and virtual) in the figure are shown in gray. The empty tuple node* `ACTOR` *is associated with the factor $f = 1$ since the* `ACTOR` *schema node has only one neighbouring node.*

Phase 2: Merging of keyword nodes. In this phase, we gradually merge keyword nodes (both regular and virtual) with other neighbouring keyword nodes along with all the edges in the path between them. After every merger, the edge score is increased accordingly.

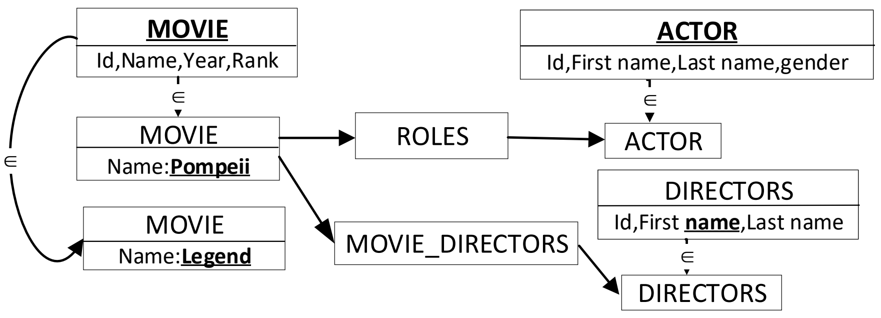


Figure 4.5 A pattern graph for the query $Q = \{\text{movie, Pompeii, Legend, actor, name}\}$ on the IMBD database.

We start by choosing a leaf node t (all leaf nodes are keyword nodes) with the smallest distance from another keyword node t'. All the nodes in the path from t to t' are collapsed into node t and all their edges become incident to node t. The edge score $ES(G_q)$ is increased by $1/p$, where p is the length of the path from t to t' in term of number of edges. If there are multiple edges t' in the same distance from node t, one of them is randomly chosen for merging. This process ends when there is only one node (or a set of disconnected nodes) left.

Example 3 *In our running example with the query pattern graph G_q of Figure 4.2, we can chose the node* `MOVIE` *with the marked term* `Pompeii` *and merge it with the virtual keyword node* `ACTOR` *which adds 1/2 to $ES(G_q)$, and then chose the node* `MOVIE` *with the marked term* `legend` *and merge it with the virtual keyword node* `ACTOR` *which also adds 1/2 to $ES(G_q)$. At that point the resulting graph has only one node* `ACTOR` *and the final edge score $ES(G_q) = 3$.*

Example 4 *As another example, consider the query pattern graph G'_q of Figure 4.5. This is another pattern graph for the same keyword query $Q = \{movie, Pompeii, legend, actor, name\}$ on the schema graph of Figure 4.1 which produced the pattern graph of Figure 4.2. After the two phases of the edge scoring process, one can see that the edge* *score of G'_q, $ES(G'_q) = 2$. This score is lower than the score of the query pattern graph G_q of Figure 4.2 and, therefore, G'_q will be ranked below G_q by our system.*

Node Scoring Several query pattern graphs may get the same edge score. To distinguish between these ties, we use IR-style scoring techniques to assign a score to pattern graphs. As this type of scoring assigns a score to the keywords which are associated with nodes, we call this type of scoring *node scoring*. IR systems assume a collection of documents and estimate the relevance of a document to a keyword query by assigning a score. An extensively used formalization for modelling queries and documents is the vector space model wherein documents and queries are represented by a vector of terms. The term space is defined by all the terms in the document collection (or at least those that can be of interest to a specific application) and each term is a dimension in this term space. Each item in a document or query vector has a non-negative value which represents the importance of the corresponding term in the document or query. A score measures the similarity between a document vector and the query vector. This score can be computed as the inner product of the two vectors (the query vector QV of the query Q and the document vector VD of the document D). That is,

$$score(Q, D) = \sum_{k \in Q,D} QV(k) * DV(k) \tag{4.2}$$

Determining the items in $QV(k)$ is quite simple: $QV(k)$ can be the frequency of a keyword k in Q. In the case where there are no repetitions in the keyword query Q, this is the reciprocal of the size of Q: $QV(k) = 1/|Q|$. Deciding the items in $DV(k)$ is decisive in determining the quality of the similarity metric. The best known technique in IR for defining the items in $DV(k)$ is the document frequency-inverse document frequency technique (*tf*idf*).

There are different variations of this technique. One of the most widely used is the pivoted normalization document length technique [74, 73, 72] which defines $DV(k)$ as follows:

$$DV(k) = \frac{ntf}{ndl} * nidf \tag{4.3}$$

where ntf, $nidf$ and ndl are the normalized term frequency, inverted document frequency and document length, respectively:

$$ntf = 1 + ln(1 + ln(tf)) \tag{4.4}$$

$$nidf = ln\frac{N+1}{df} \tag{4.5}$$

$$ndl = (1 - s) + s * \frac{dl}{avgdl} \tag{4.6}$$

The symbol s, the *slope*, in Equation (4.6) is a parameter which is usually set equal to 0.2. Symbol N in Equation (4.5) is the number of documents in the document. avgdl in Equation (4.6) is the average length of the documents in the document collection. These normalizations substantially improved the performance of keyword search on documents.

We adapt these formulas to our context of keyword search over relational databases and query pattern graphs for keyword queries. Query keywords can match relation names, attributes, and attribute values in tuples. When a keyword matches an attribute value in a tuple, we define tf, df, and dl as follows:

In Equation (4.4), term frequency tf for a keyword k which is a value of an attribute A is the frequency (number of occurrences) of k in the column A. The intuition is that the higher the frequency of a keyword in a column, the higher the contribution of the keyword to the score of the pattern graph. Research in IR has concluded that if the score is linearly dependent on tf, then the contribution of tf is exaggerated. For the normalized frequency ntf, a log function is applied twice to tf to blunt the contribution of tf.

In Equation (4.5), *df* is the number of attribute values in the database which are matched by keyword k, while N is the number of non-null textual attribute values in the database. Intuitively, the more frequent a keyword is in the database, the less important the contribution of tf should be to the score. In the normalized version *nidf* of *idf*, $N + 1$ is divided by *df* and is then dampened by applying a log function.

In Equation (4.6), for a keyword k which is a value of an attribute A, *dl* is the number of non-null attribute values in column A. *avgdl* is the average number of non-null attribute values in the textual columns of the database tables. Intuitively, larger columns will have larger values for *dl* and consequently smaller values for the overall score of a pattern graph. When attribute A has many k values, the normalized version *ndl* of *dl* reduces the contribution of ntf to $DV(k)$ by dividing *dl* by *avgdl*. The slope, s is set to 0.2.

Example 5 *Consider the keyword query* {legend} *against an IMDB database whose schema is shown in Figure 4.1. Two matches for* legend *in this database are a value of attribute* `Name` *in Table* `MOVIE` *and a value of attribute* `Last_name` *in table* `ACTOR`*.* legend *is more frequent in column* `Name` *of table* `MOVIE` *than in column* `Last_name`*. The first pattern graph gets a node score of 31.84 (ndf = 2.90, nidf = 9.81, ndl = 0.90) while the second one gets a node score of 17.02 (ndf = 1.74, nidf = 9.89, ndl = 1.01). Therefore, the system ranks the first pattern graph higher.*

When a keyword k matches an attribute A in the database schema, $DV(k)$ is computed as the maximum $DV(k_i)$ among all the terms k_i of interest in column A.

When a keyword k matches a relation name R in the database schema, $DV(k)$ is computed as the maximum $DV(k_i)$ that can be obtained from the terms k_i of interest in columns of relation R. The value of $DV(k)$ is precomputed and stored in the database for every term k which is an attribute or relation name in the database schema.

4.2 The Semantic Keyword Search Algorithm

We present now a top-K algorithm for computing the pattern graphs of a query on a relational database. We first present the techniques we use to reduce the number of redundant intermediate results, which is the bottleneck of keyword search algorithms on databases with structure. We then elaborate on the techniques we use to compute the top-K results without computing and ranking all the results.

Annotated schema graph. Given a keyword query Q on a database with schema S, an *annotated schema graph* is a schema graph where each node/relation is annotated with all the query keywords in Q which match an attribute value in the instance of this relation, or an attribute in the schema of this relation or the name of this relation, along with the type of this match. Figure 4.1 provides an example of an annotated schema graph for the keyword query $Q = \{movie, Pompeii, legend, actor, name\}$ on an IMDB database. If multiple keywords match attribute values in the same tuple of a relation, this information is also recorded in the annotated schema graph.

Expanded schema graph. From the annotated schema graph we construct an *expanded schema graph* G which is a n-partite graph, where n is at least the number of nodes/relations in the database schema S and at most twice this number. For every node/relation R in the annotated graph, G has two independent sets of nodes VR and SR. Each node VR_i in VR is annotated by an annotation of R referring to an attribute value matching keyword and

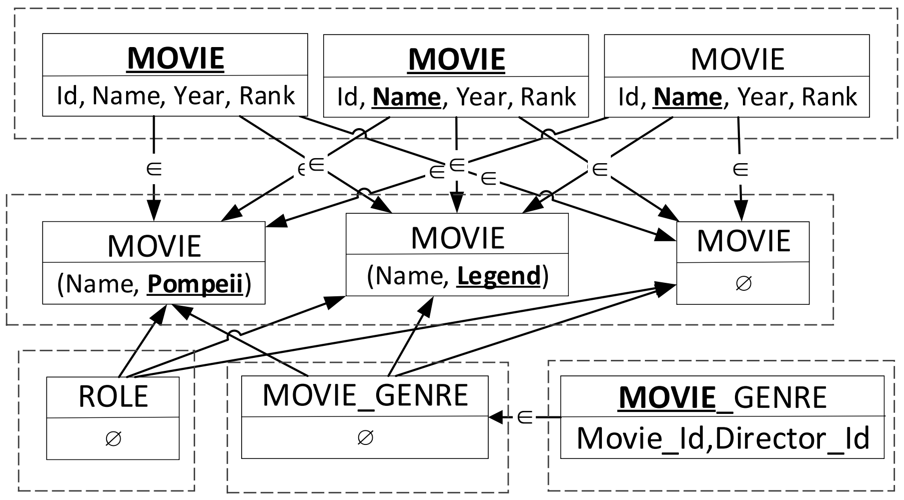


Figure 4.6 Part of the expanded schema graph constructed based on the annotated schema graph of Figure 4.1.

.

one of them is annotated by the empty subset. Each node SR_i in SR is annotated by a non-empty subset of the annotations of R referring to schema element matching keywords so that no two annotations referring to the same keyword are included in the subset more than once. If there are no annotations of R in the annotated graph referring to schema element matching keywords, SR is empty and is omitted. There are inclusion edges in G between every node in SR and every node in VR. There is a key-foreign key edge between every node VR_i in VR and every node VW_i in the independent node set VW for another relation W in the schema S if there is a key-foreign key relationship between R and W.

Example 6 *Figure 4.6 shows part of the expanded schema graph constructed from the annotated schema graph shown in Figure 4.1 involving the independent node sets for the schema nodes MOVIE, ROLE and MOVIE_GENRE (and the corresponding inclusion and key-foreign key edges). There are two independent node sets of three nodes each for the annotated schema graph node MOVIE, which has annotations referring to schema elements and annotations referring to attribute values. There are two singleton independent node sets for the annotated schema graph node MOVIE_GENRE. Finally, there is one singleton* *independent node set for the annotated schema graph node ROLE and its node is annotated by the empty set.*

Our algorithm for computing patterns graphs for a keyword query takes as input a keyword query Q, an expanded schema graph G, a size threshold T for the query pattern graphs, and a threshold K on the number of query pattern graphs to be returned.

4.2.1 An Unconstrained Expansion Strategy

The algorithm constructs query pattern graphs which are trees and which are transformed into graphs with cycles at the end of the process by adding inclusion edges if needed. It starts by choosing randomly a keyword k from the input keyword query Q and by constructing intermediate pattern graphs with a single node. For every node R_i in the expanded schema graph whose annotation contains keyword k, one node n labeled by R and annotated by the annotation of R_i is constructed and pushed into a stack. The query pattern graphs of Q are constructed by expanding these initial single node query pattern graphs.

In the absence of any expansion constraint, the expansion process can be outlined as follows: consider a node n labeled by R_i in an intermediate query pattern graph P built upon the node R_i in G. For every adjacent node W_j to node R_i in G, P is expanded by adding an adjacent node m to n labeled by W_j and annotated by the annotation of W_j in G. For each one of these expansions to be possible, the keywords in the annotation of m should not appear in the annotation of any other node in P. Note that the annotation of a new node in P can be empty. All the nodes in the intermediate pattern graph can be considered for expansion. Intermediate results which contain all the keywords of Q in the keyword sets of their nodes are characterized as final query pattern graphs and are returned to the user if they do not have empty tuple leaf nodes (minimality condition of query pattern graphs). If they do not satisfy the minimality condition, they are discarded. Intermediate query pattern graphs whose size is equal to the given size threshold and are not final are not expanded anymore and they are discarded. This process will generate duplicate results (isomorphic ordered versions of unordered trees). Hence, the pattern graphs will have to be compared for isomorphism with previous pattern graphs if duplicates are unwanted. At the end of the process, inclusion edges are added in any pattern graph between schema nodes and tuple nodes of the corresponding relation.

The expansion process can be constrained without missing any results. One technique we are using for this is to leverage a canonical form for pattern graphs. .

4.2.2 Exploiting a Canonical Form for Pattern Graphs

Our algorithm for computing query pattern graphs initially constructs trees starting from a root node. Therefore, initially query pattern graphs are generated as rooted trees.

Two labeled rooted trees R and S are *isomorphic* to each other if there is a one-to-one mapping from the nodes of R to the nodes of S that preserves node labels, edge labels, adjacency and roots. An *automorphism* is an isomorphism that maps a tree to the same unordered tree.

To reduce the generation of isomorphic trees (which are redundant since they represent the same unordered tree) and produce only one tree for every automorphic group, we use a canonical form for the trees initially generated by our algorithm for query pattern graphs.

Given a node n in a tree R, let $root(R)$ denote the root of R and $subtree(n)$ denote the subtree of R rooted at n.

An Order for Labels and Trees. We first define an order for trees. Let $\leq$ be a linear order on the set of labels that can appear on the nodes of trees. Abusing notation, we also denote as $\leq$ an order on trees defined recursively as follows:

Definition 4.2 (Tree Order) *Given two trees R and S, let $r = root(R)$ and $s = root(S)$, respectively. Let also $r_1, \ldots, r_m$ and $s_1, \ldots, s_n$ denote the list of the children of r and s, respectively. Then, $R \leq S$ iff either:*

1. *$label(r) \leq label(s)$, or*
2. *$label(r) = label(s)$ and either:*

   (i) $m \geq n$ and $\forall i \in [1, n], subtree(r_i) = subtree(s_i)$, or

   (ii) $\exists k \in [1, min(m, n)]$ such that: $\forall i \in [1, k-1], subtree(r_i) = subtree(s_i)$ and $subtree(r_k) \leq subtree(s_k)$.

The tree order above is essentially the same as the one introduced in [67, 88].

A Tree Canonical Form. We can define now the canonical form adopted for a tree

Definition 4.3 (Tree Canonical Form) *A tree R is in* canonical form *if for every tree S which is automorphic to R, $R \leq S$.*

To check whether a tree is in canonical form in an efficient way, a string representation for trees and the results presented in [60] can be leveraged.

Exclusive expansion of pattern graphs in canonical form. For the pattern graphs, it is sufficient to have trees in canonical form (the other isomorphic versions of the pattern graphs are redundant). It turns out that intermediate pattern graphs which are not in canonical form do not need to be considered in the expansion process. This is shown by the next proposition.

Proposition 6 *All the pattern graphs of a query on a database can be computed by considering, during the expansion process, only intermediate pattern graphs in canonical form.*

Therefore, intermediate pattern graphs which are not in canonical form can be discarded. Clearly, eliminating non-canonical intermediate pattern graphs from the search space allows a significant reduction of the number of pattern graphs generated.

Empty node expansion. It can be further observed that if an intermediate result tree has an empty leaf node, it can be chosen for expansion without missing any results. This is shown by the next proposition.

Proposition 7 *Given an expanded schema graph G, all the pattern graphs of a query Q can be computed by expanding only empty leaf nodes in the pattern graph trees under construction (if empty leaf nodes exist).*

Following this remark, if an empty node exists in an intermediate result, it should be chosen for expansion.

Rightmost path node expansion. In the absence of an empty leaf node, all the nodes of intermediate pattern graphs need to be expanded to guarantee the computation of all the pattern graphs. The next proposition shows that many intermediate pattern graphs nodes can be excluded from expansion without affecting the completeness of the pattern graph set computed.

Proposition 8 *Given an expanded schema graph G, all the pattern graphs of a query Q can be computed by expanding only the nodes on the rightmost path of the pattern graphs trees under construction.*

If this expansion strategy is applied, the intermediate pattern graph tree nodes which are not on the rightmost path are not expanded. These nodes represent in most cases the majority of the nodes.

Checking the canonicity of pattern graph trees incrementally. In order to check whether a pattern graph tree is in canonical form we can use previous results on tree canonical forms and a string representation for trees to perform the comparisons of trees efficiently. However, as mentioned above: (a) only trees which are in canonical form are expanded, and (b) only nodes in the rightmost path of an intermediate pattern graph tree are expanded. These remarks allow for an incremental checking of the canonicity of the newly generated intermediate pattern graph tree. Details on how these techniques can be implemented are presented in [60].

Exploiting the techniques presented in this section can reduce dramatically the number of redundant intermediate pattern graphs computed and can directly benefit the performance of the pattern graph computation algorithm since it is the large number of intermediate results that constitutes the bottleneck of keyword query evaluation algorithms on structured databases.

4.2.3 The Top-k Strategy

The goal of a top-K strategy is to find the top-K pattern graphs (the K pattern graphs such that there are no other pattern graphs with higher score) without necessarily computing and ranking all of them. As mentioned above, we use edge scoring to initially rank the pattern graphs while node scoring is used to break the ties. Therefore, our top-K strategy focuses on computing top-K pattern graphs based on their edge score. Node scoring can be employed to further rank these pattern graphs if there are pattern graphs with the same edge score among the top-K.

Our approach computes patterns graphs by expanding intermediate pattern graphs. The basic idea of our top-K strategy is to predict an upper bound B for the score of each pattern graph derived by expanding the current intermediate pattern graph P during the computation of the pattern graphs. If B is lower than the score of the K^{th} pattern graph in the ranked list of pattern graphs computed so far (ranked in descending score order), then P does not need to be expanded anymore and the computation moves to consider another intermediate pattern graph for expansion. This can substantially improve the performance of the pattern graph computation algorithm in terms of both execution time and memory consumption.

Recall that each intermediate pattern graph P is derived from another pattern graph P' by adding an adjacent node n to a node n' in P' and node n becomes the rightmost leaf node of P (we called this expansion of P). Based on our expansion strategy (Section 4.2.2) node n' in P' has to be on the rightmost path of P' and can be either an empty or non-empty node of P'. If the rightmost leaf node of P' is an empty node, this is the only node of P' that can be expanded. Assume that the missing keywords (those keywords of Q which are not already marked in P') have exclusively matches on tuple values in the annotated schema graph and the corresponding relations do not have any schema nodes in P'. We claim that the score of the pattern graph P constitutes an upper bound to the score of all the pattern graphs that can be derived by expanding P' assuming that all the missing keywords in P' are marked terms on the newly added node n.

To further explain this, observe that if m missing keywords are marked terms in the tuple node n, they contribute a co-occurrence score of $m-1$ to the edge score of the final pattern graph. If these m keywords are distributed in other tuple nodes, their contribution to the edge score of the final pattern graph due to the co-occurrence score and because of the merging of keyword nodes can never exceed $m-1$.

4.2.4 The Algorithm

Our algorithm proceeds as described in Section 4.2.1. It computes only intermediate and final query pattern graphs in canonical form and follows the expansion rules described in Section 4.2.2. This reduces substantially the generation of intermediate pattern graphs and eliminates the generation of redundant intermediate tree pattern graphs in the pattern graph generation process. It further implements the top-K strategy presented in Section 4.2.3. The pseudo code for the algorithm, named *topK-QPGC* (for top-K Query Pattern Graph in Canonical form computation) is presented in Algorithm 2. The algorithm returns a list L of top-K query pattern graphs ranked in descending order on their edge score. Function $mterms(R)$ denotes the marked terms in a pattern graph/pattern graph node R. Function $predictScore(R)$ returns an upper bound for the edge scores of the pattern graphs that can be generated by expanding the intermediate pattern graph R. Procedure $insertRanked(R, L)$ inserts the query pattern R in the right position in the ranked list L. If L is already full, it discards the last pattern graph in L.

As described in Section 4.2.1, Algorithm *topK-QPGC* exhaustively expands the intermediate pattern graphs of a keyword query Q starting with nodes that contain a randomly selected query keyword. It avoids expanding pattern graphs which are not in canonical form and expends nodes in a pattern graph from the rightmost path or just the

Algorithm 2 *Algorithm topK-QPGC*

Input: *expanded schema graph G, keyword query Q, pattern graph size threshold T, number of pattern graphs threshold K*

Output: *a ranked list L of top-K query pattern graphs of Q of size up to T*

```
 1: $L := \emptyset$
 2: $E := \emptyset$     /* $E$ is a queue of pattern graphs under construction
 3: $KThreshold := 0$
 4: Let $k$ be a keyword in $Q$
 5: for every node $n$ in $G$ s.t. $k \in annotation(n)$ do
 6:     Let $R$ be a new node labeled by $n$  /* $R$ is a new pattern graph in construction
 7:     $mterms(R) = annotation(n)$
 8:     $enqueue(R, E)$
 9: while $E \neq \emptyset$ do
10:     $R :=$ dequeue($E$)
11:     if $R$ has an empty leaf node $q$ then
12:         $ExpandList := \{q\}$
13:     else
14:         $ExpandList := \{$ nodes in the rightmost path of $R$ $\}$
15:     for every node $l \in ExpandList$ do
16:         for every node $m$ adjacent to the node $label(l)$ in $G$ do
17:             if $annotation(m) \cap mterms(R) = \emptyset$ then
18:                 Add a node $p$ with $label(p) = label(m)$ and $mterms(p) = annotations(m)$
                    and an edge $(l, p)$ to $R$.
19:                 if $R$ is in canonical form and satisfies the structural constraints then
20:                     if $mterms(R) = Q$ & $ES(R) > KThreshold$ then
21:                         $insertRanked(R, L)$
22:                         if $size(L) = K$ then
23:                             $KThreshold := ES(lastPattern.L)$
24:                     else
25:                         if $size(R) < (T - 1)$ then
26:                             if $predictScore(R) \leq KThreshold$ then
27:                                 drop $R$
28:                             else
29:                                 $enqueue(R, E)$
```

rightmost empty leaf node if there are any. As we showed in Section 4.2.2, these expansion restrictions do not prevent the computation of any pattern graph of Q.

Proposition 9 *Given the expanded schema graph G of a schema S based on a keyword query Q, Algorithm topK-QPGC correctly computes the top-K pattern graphs of Q.*

4.3 Experimental Evaluation

We run experiments to assess the efficiency of the algorithm in terms of execution time, memory consumption and the effectiveness of the approach.

4.3.1 Databases, Queries, Metrics

Databases. We used both a version of the IMDB database[1] and a version of the Mondial database[2] in our experiments. The IMDB database contains actor, director and role information for movies. Its schema comprises seven tables which are connected with six foreign keys. The Mondial database contains geographic and demographic information. The schema of this database contains thirty-three tables and forty-eight foreign keys. Note that the number of nodes and edges in the expanded schema graph for a given query is usually a multiple of the respective numbers in the database schema graph.

Query Sets. We use a set of 10 queries for each database for evaluation. The queries have from two to six keywords. Most queries contain keywords which match both schema elements (attributes and/or relation names) and tuple values. Tables 5.2 and 5.4 show statistics for the queries. Column #kws displays the number of keywords of each query. Column #nodes (resp. #edges) shows the number of nodes (resp. edges) in the

[1]https://relational.fit.cvut.cz/dataset/IMDb
[2]https://www.dbis.informatik.uni-goettipeii ngen.de/Mondial/

Table 4.2 Statistics for Queries on the IMDB Dataset

IMDB				
Query#	#keywords	#nodes	#edges	#pat. graphs
Q1	3	19	53	3353
Q2	3	24	86	5281
Q3	4	24	81	28226
Q4	3	20	53	3496
Q5	4	27	95	38028
Q6	5	24	72	3609
Q7	4	26	107	31862
Q8	6	32	151	362274
Q9	4	31	145	43401
Q10	6	24	85	6960

Table 4.3 Statistics for Queries on the Mondial Dataset

Mondial				
Query#	#keywords	#nodes	#edges	#pat. graphs
Q1	5	69	272	158588
Q2	3	50	141	14043
Q3	2	57	169	76617
Q4	2	55	220	97214
Q5	3	51	204	2744
Q6	3	47	156	60094
Q7	2	43	109	15202
Q8	5	55	190	16636
Q9	4	57	231	264675
Q10	6	68	293	1676

expanded schema graph generated for each query. Column #pat. graphs shows the total number of pattern graphs generated for each query on the expanded schema graph.

Metrics. For the effectiveness experiments, we measured *precision@N*, *Normalized Cumulative Gain* (*nDCG*) at position N and *Kendall rank correlation coefficient* which we further describe in Section 4.3.3.

4.3.2 Efficiency Experiments

In these experiments, we measured the efficiency of our algorithm, *topK-QPGC*, in terms of execution time, number of intermediate results generated, and memory consumption.

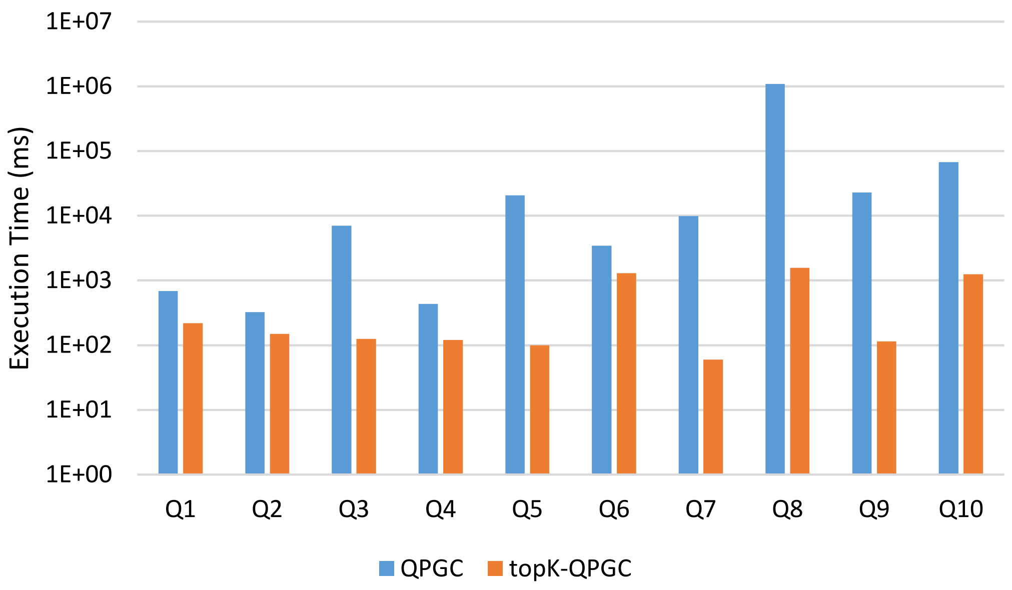


Figure 4.7 Execution time for queries on the IMDB database.

"Results" refers here to pattern graphs. Intermediate results are the partial or complete pattern graph that are generated during the execution of the algorithm. Memory consumption is expressed by the maximum number of intermediate results ever kept in memory. We also compared the measurements with those of an algorithm which computes the query pattern graphs leveraging the canonical form of pattern graphs but without using the top-k strategy. We refer to this latter algorithm as *QPGC*.

Figures 4.7 and 4.8 show the execution time of the algorithms on the IMDB and the Mondial databases, respectively. The scale of the y-axis is logarithmic. One can see that algorithm *topK-QPGC* is much faster than *QPGC* which does not leverage the top-K strategy and therefore generates first all the pattern graphs, and uses the scoring functions to score them and rank them. The top-K strategy improves the execution time by least one order of magnitude in most of the cases. For Query 10 on the Mondial database, the execution times between the two algorithms do not display much difference. This is because Query 10 does not produce many pattern graphs. For a case like that one the top-K strategy does not improve the execution time significantly.

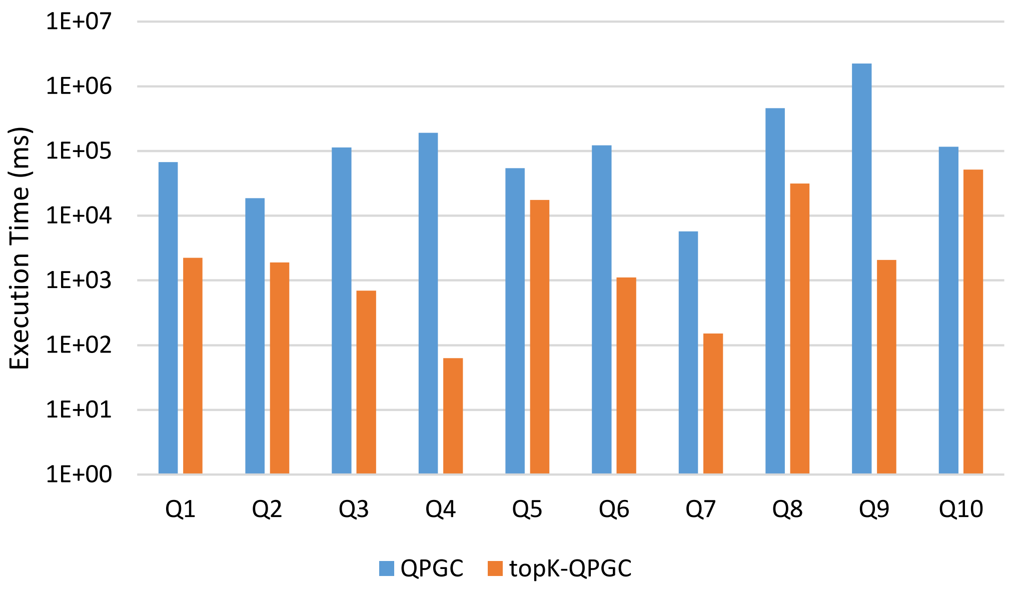


Figure 4.8 Execution time for queries on the Mondial database.

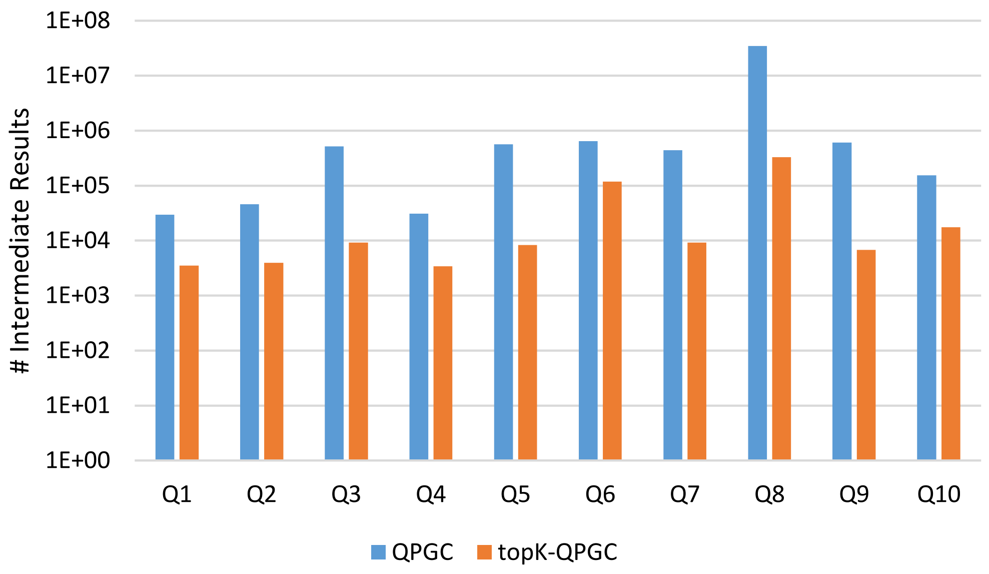


Figure 4.9 Number of intermediate results produced by the queries on the IMDB database.

Figures 4.9 and 4.10 show the performance of the two algorithms in terms of the number of intermediate results produced on the IMDB and Mondial databases, respectively. As described in Section 4.2.3, Algorithm *topK-QPGC* avoids generating subsequent results from the intermediate result under consideration if their score is predicted to be lower than the score of the Kth pattern graph in the ranked list of pattern graphs computed that far.

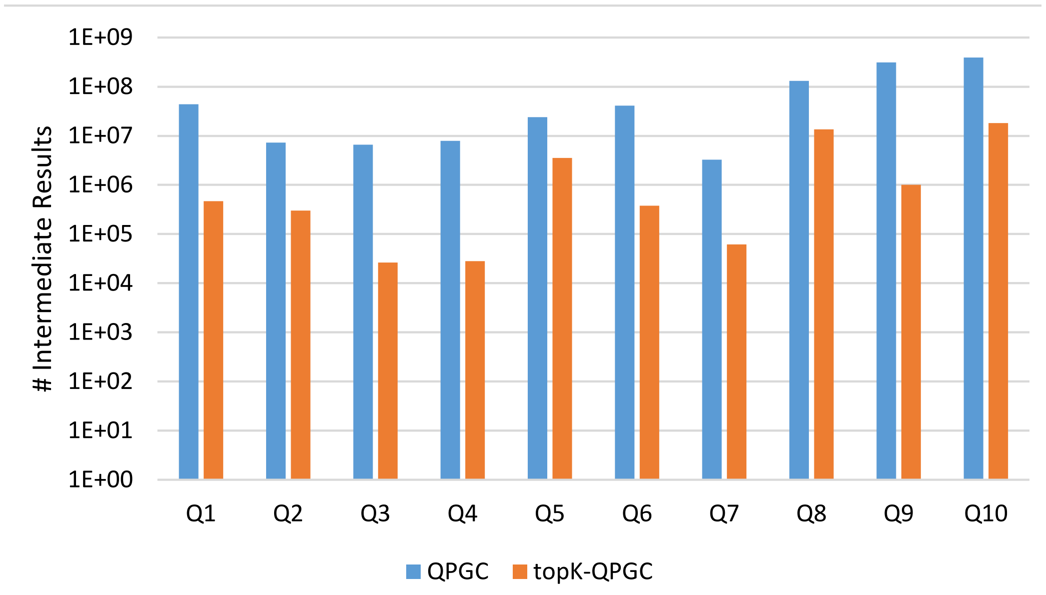


Figure 4.10 Number of intermediate results produced by the queries on the Mondial database.

This reduces the total number of intermediate results generated by *topK-QPGC* compared to *QPGC*. As one can see in the figures, *topK-QPGC* produces far fewer intermediate results than *QPGC* in all cases on both datasets. The number of intermediate results produced by Algorithm *QPGC* is at least one order of magnitude larger than those of *topK-QPGC* in most of the cases.

As one can also observe in the plots, the execution time and the number of the intermediate results produced are strongly correlated. Despite the fact that *topK-QPGC* spends time on almost every intermediate result predicting the maximum score of the pattern graphs that can be generated from the pattern graph under consideration by the algorithm, this delay is not large enough to counter the benefit in execution time obtained from the reduction in the number of intermediate results generated.

Figures 4.11 and 4.12 show the memory consumption of the algorithms on the IMDB and the Mondial databases, respectively. We measured the memory consumption in terms of the maximum number of intermediate results stored in the queue that keeps the intermediate

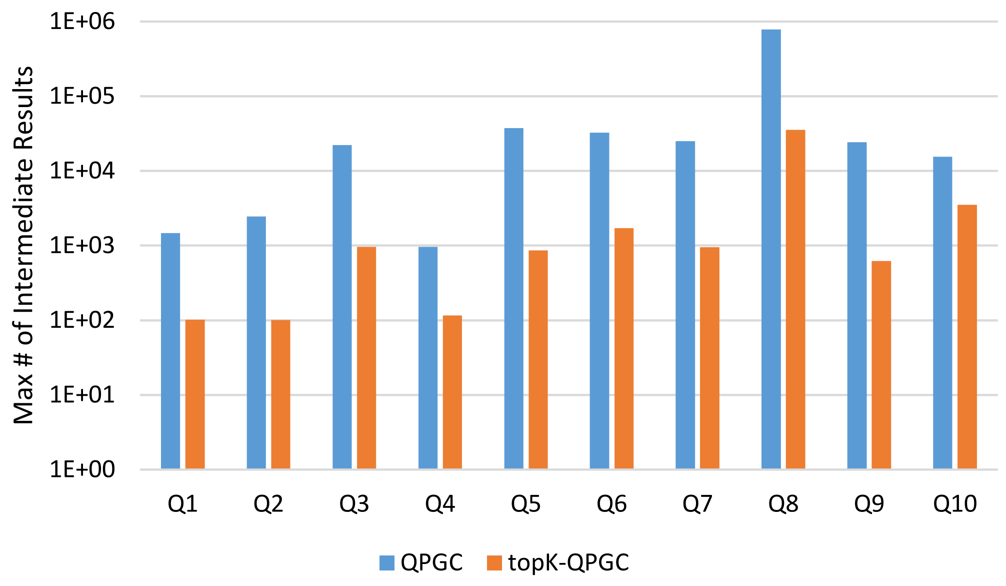


Figure 4.11 Memory consumption for queries on the IMDB database.

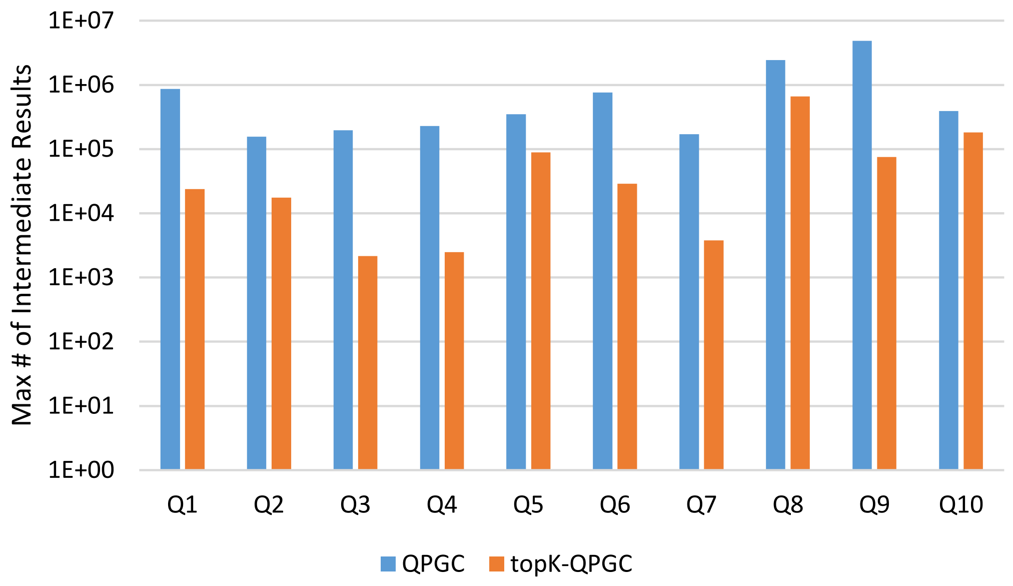


Figure 4.12 Memory consumption for queries on Mondial the database.

results (Section 5.3) at a specific moment in time. The scale of the y-axis is logarithmic. In all cases, the memory consumption of *topK-QPGC* is smaller than that of *QPGC*. More specifically, in most of the cases, the memory consumption of *QPGC* is ten times bigger than the memory consumption of *topK-QPGC*. This memory footprint difference is expected since the top-K strategy prevents the generation of intermediate results which will not yield high enough scores.

4.3.3 Effectiveness Experiments

We also run experiments to assess the quality of our approach. Our algorithm returns a list of K results ranked (based on their computed scores). To measure the quality of the results we used *Precision@N* and to measure the quality of the ranking we used *nDCG* and Kendall tau.

The ground truth for the query results was determined as follows: for each of the queries, a user not related to this project got the unordered result sets and assigned to the computed pattern graphs a score from 0 to 3, with 3 meaning very relevant and 0 meaning completely irrelevant. To characterize a pattern graph simply as relevant or irrelevant, the scores 3 and 2 defined a relevant pattern graph and the scores 1 and 0 an irrelevant pattern graph. The user also produced a strict ranking of the pattern graphs consistent with the scores.

Presicion@N. Precision@N is the ratio of the number of relevant pattern graphs in the first N positions in a ranking of results returned by the system, to N. The relevance (or irrelevance) of a pattern graph at any position is determined as described above. Precision@N ranges between 0 and 1 with 1 meaning that all the first N pattern graphs are relevant and 0 meaning that all of them are irrelevant. We measured presicion@N for $N = 5$. Figures 4.13 and 4.14 show presicion@5 for all the queries on the IMDB and the Mondial databases, respectively. The measurements show satisfactory precision@5, with all the values but two being above 60%.

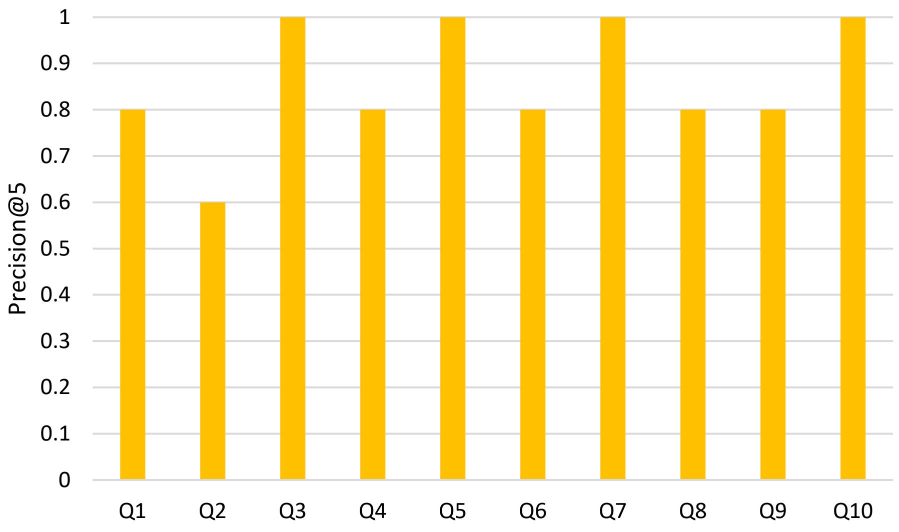


Figure 4.13 *Precision@5* for queries on the IMDB database.

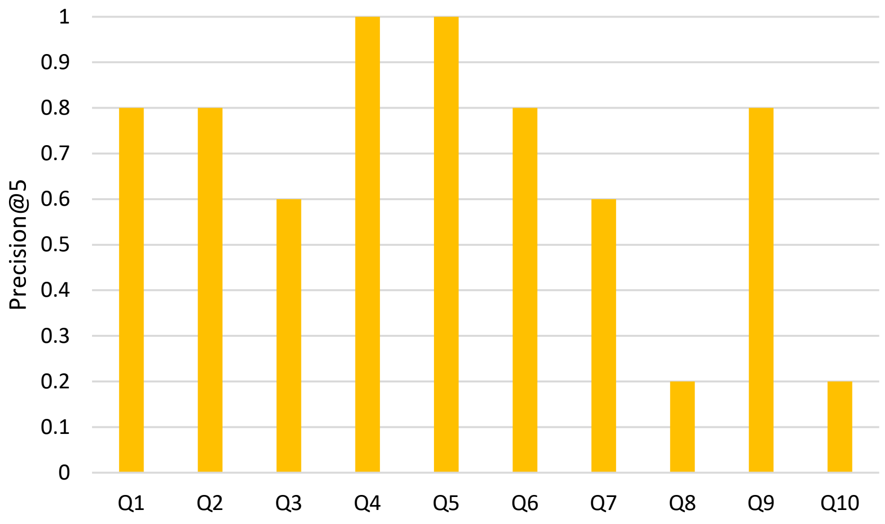


Figure 4.14 *Precision@5* for queries on the Mondial database.

Normalized Discounted Cumulative Gain. Discounted Cumulative Gain (DCG) is used often in information retrieval to measure the ranking quality of web search engine algorithms. It is based on the assumptions that highly relevant results are considered to be more useful when appearing earlier in the ranked list returned by the system, and that they are also more useful than less relevant results. Hence, a discounting function is used

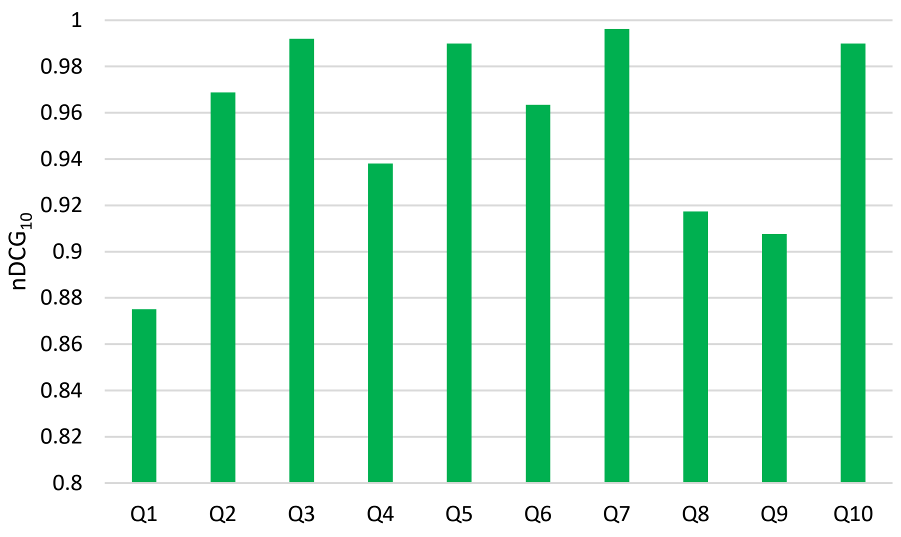


Figure 4.15 $nDCG_{10}$ for queries on the IMDB database.

over cumulative gain to measure DCG at position N ($nDCG_N$), which is defined as the sum of the relevance scores (determined by the user) of all the items at positions 1 to N in the ranked list produced by the system, each divided by the logarithm of its respective position in the ranked list. $nDCG_N$ is the result of normalizing DCG_N with the DCG_N of the list that is ranked based on the ground truth scores; it is computed by dividing the DCG_N value of the system's ranked list by the DCG_N value of the correctly ranked list. Clearly, $nDCG_N$ favors a ranked list which is similar to the correct ranked list. Its values range between 0 and 1.

We measured DCG_N with $N = 10$. Figures 4.15 and 4.16 show the $nDCG_10$ for each query of the IMDB and the Mondial database, respectively. As one can see, $nDCG_10$ is over 0.9 in most of the cases and in some cases very close to 1.0.

Kendall tau. The Kendall tau rank correlation coefficient is used to measure the association between two different rankings of the same set of items. In our setting, we have the list of k pattern graphs computed and ranked by our system and the same list ranked based on the ground truth. We want to see if the comparison of the ranked list produced

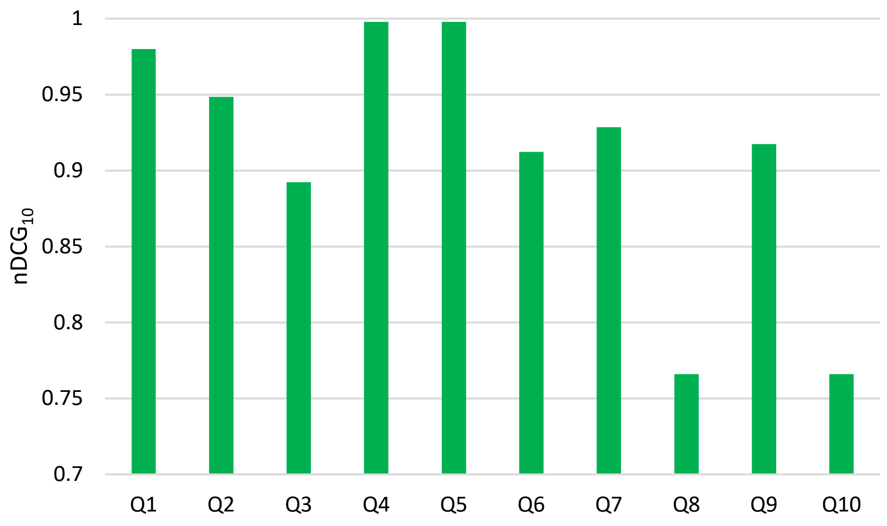


Figure 4.16 $nDCG_{10}$ for queries on the Mondial database.

by our system with the correctly ranked list which is produced by the user suggests that the former possesses a reliable judgment of the relevance of the pattern graphs produced by the latter. If two items have the same (resp. different) relative rank order in the two lists, then the pair is said to be *concordant* (resp. *discordant*) pair. The value of Kendall tau is normalized to range values from -1 to 1. If the number of concordant pairs is much larger than the number of discordant pairs, then the two lists are positively correlated (the coefficient is close to 1). If the number of concordant pairs is much less than the discordant pairs, then the two lists are negatively correlated (the coefficient is close to -1). Finally, if the number of discordant and concordant pairs are about the same, then the two lists are weakly correlated (the coefficient is close to 0). In this case, the two lists are independent.

Figures 4.17 and 4.18 show the Kendall rank correlation coefficient for the top 10 ranked results returned by our system for each query on the IMDB and the Mondial databases, respectively. As one can see, the Kendall tau coefficient is over 0.5 in most of the cases on both databases. Only for query Q10 in the Mondial database the value of Kendall tau is 0.6, as the pattern graph ranked at position eight by the system was found to be more semantically meaningful by the user and was placed first in the ground truth list.

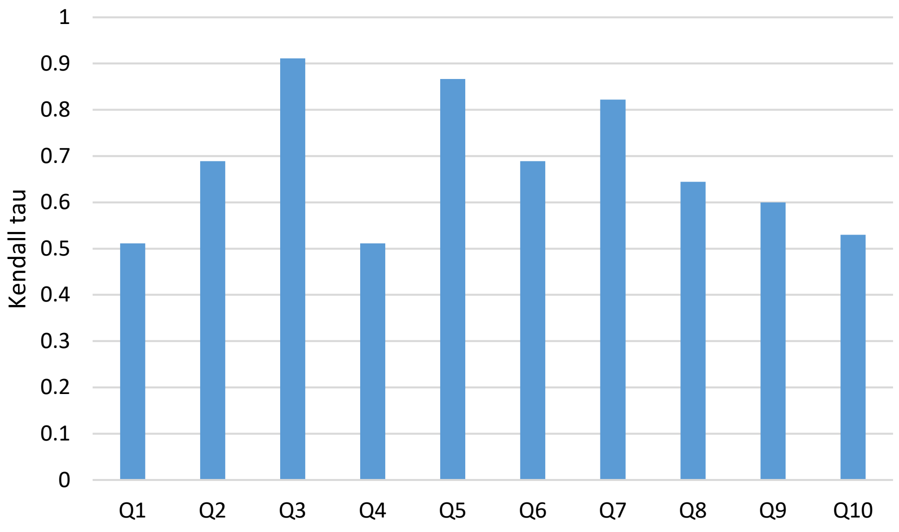


Figure 4.17 *Kendall tau* for queries on the IMDB database.

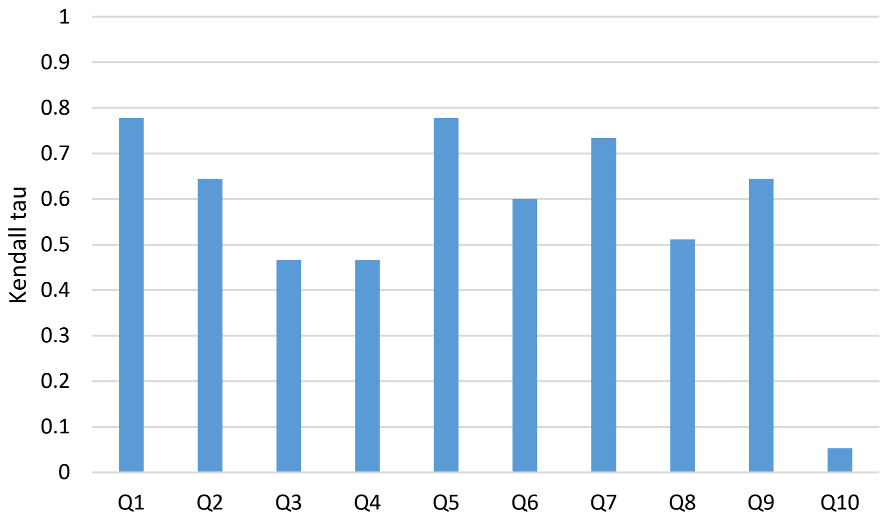


Figure 4.18 *Kendall tau* for queries on the Mondial database.

Overall, the ranking of the pattern graphs produced by the system was highly correlated with the ground truth ranking given by the user.

CHAPTER 5

SEMANTICS AND EVALUATION OF COHESIVE KEYWORD QUERIES ON STRUCTURED DATABASES

5.1 Introduction

We leverage in this chapter keyword queries with cohesiveness constraints (called *cohesive queries*) to address the problems related to keyword search on structured databases.

A *cohesive group* G in a keyword query Q is a proper subset of Q. A cohesive group G defines a *cohesiveness constraint* in Q which states that the keywords in G constitute a cohesive whole. In other words, the instances of the keywords of G in a pattern graph of Q should be closer to each other than to the instance of any other keyword in Q outside G.

We specify cohesive groups in cohesive queries by enclosing their keywords between parentheses. For example, *(SQL (James Smith) (John Johnson))* is a cohesive keyword query against a bibliographic database. The user is searching with this cohesive keyword query for publications on SQL related to the authors James Smith and John Johnson. This query indicates that (*James Smith) and (*John Johnson) are cohesive groups and therefore, the system can use this information to return more accurate results. For instance, it will be able to filter out publications on SQL by John Smith and James Johnson. The early exclusion of irrelevant results also allows for substantial improvements in the query evaluation time during the computation.

Cohesive groups can be nested. For instance with the query (*SQL (James Smith)(citation (John Johnson))*) the user looks for a paper on SQL written by James

Smith which cites a paper by John Johnson. Additional examples are included in the related work section of the dissertation.

5.2 Keyword Queries with Cohesiveness Constraints

We provide semantics for cohesive keyword queries by identifying below the cases where the keyword matches in a pattern graph of the flat keyword query violate the cohesiveness constraints (and, therefore, this pattern graph is not part of the answer to the cohesive keyword query).

Let a, b and c denote keywords of a cohesive keyword query Q such that a and b belong to a cohesive group G in Q and c does not. Let also P be a pattern graph for keyword query Q without any cohesiveness constraints.

Case 1: Keyword a matches an attribute in the schema node of a relation R and keyword c matches a value for attribute a in a tuple node of relation R. If b matches any other term in P, c is considered to be closer to a than b is to a (that is, the cohesiveness of group G is breached and the cohesiveness constraint is violated). This case is depicted in Figures 5.1(a) and (b).

Case 2: Keywords a and c match values in a tuple node and keyword b matches any term other than the attribute whose value is matched by keyword a. The relationship of keywords a and c appearing in the same tuple node is stronger than other possible relationships between a and b in P. This case is depicted in Figure 5.1(c) and (d).

Case 3: Keyword c matches an attribute of relation R in the schema node for R, a matches a value of attribute c in a tuple node of R, and b matches any other term in the pattern graph. The attribute/attribute value relationship between c and a is stronger than

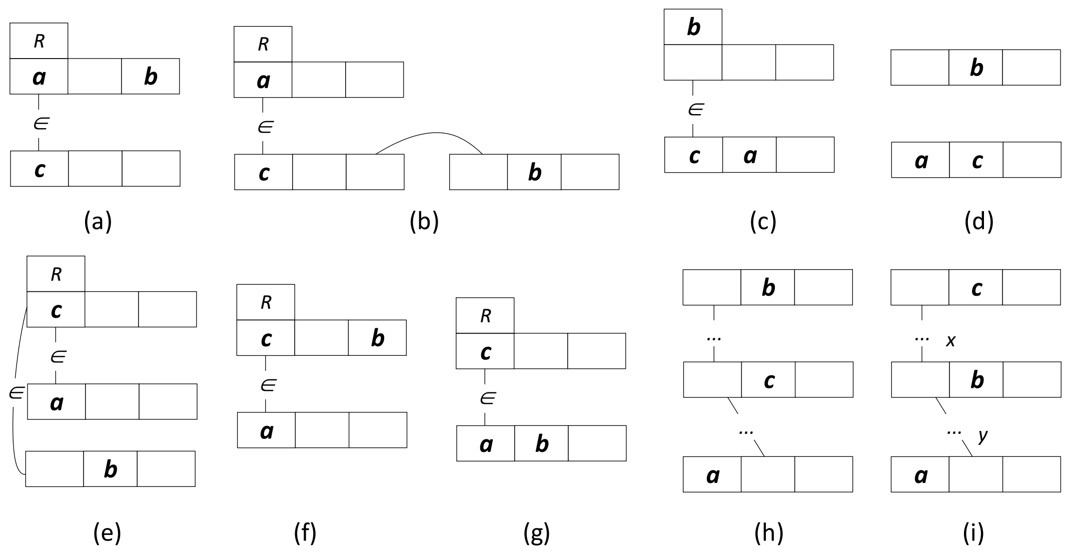


Figure 5.1 Result graphs.

any other relationship between a and b and the cohesiveness constraint is violated. This case is depicted in the Figures 5.1(e), (f) and (g).

Case 4: Keywords a, b and c match values in different tuple nodes t_a, t_b and t_c, respectively, in P and there is a path of key-foreign key relationships between t_a and t_c and a path of key-foreign key relationships between t_b and t_c and no path of key-foreign key relationships between t_a and t_b. Clearly, here, a and c are closer than a and b and the cohesiveness constraint is violated. This case is depicted in Figure 5.1(h).

Case 5: Keywords a, b and c match values in different tuple nodes t_a, t_b and t_c, respectively, in P and there is a path of key-foreign key relationships between t_b and t_c of length x and a path of key-foreign key relationships between t_b and t_a of length x and no path of key-foreign key relationships between t_a and t_c, and x, y. Clearly, here, b and c are closer to each other than a and b and the cohesiveness constraint is again violated. This case is depicted in Figure 5.1(i).

5.3 Algorithm

Our algorithm for computing the answer of cohesive keyword queries on relational databases is called *CKER* (for <u>C</u>ohesive <u>K</u>eyword query <u>E</u>valuation algorithm on <u>R</u>elational databases) and is shown in Algorithm 3. The algorithm takes as input a cohesive query Q, the expanded schema graph of the relational database and a pattern graph size limit T.

Algorithm *CKER* starts by generating all the atomic cohesiveness constraints in Q. This is done by avoiding the redundant generation of atomic constraints defined by nested cohesive groups. The atomic cohesiveness constraints are stored into a checklist AC. The algorithm proceeds for generating patterns graphs the same way Algorithm 2 does. The cohesiveness constraints are checked for violations incrementally: for each new intermediate or final result generated during the evaluation of the query, if a new keyword is introduced, the algorithm checks whether a new atomic cohesiveness constraint is also introduced, in which case it checks this constraint for violation. If the atomic constraint is violated the (intermediate) result is discarded and its expansion is discontinued. Otherwise, the expansion continues until all the keywords are collected or the size limit is reached.

5.4 Experimental Evaluation

We implemented the cohesive query evaluation algorithm and we run experiments to assess the effectiveness of the approach and the efficiency of the algorithm in terms of execution time, number of intermediate results, memory consumption and scalability.

Algorithm 3 *Algorithm CQER*

Input: *cohesive query Q, schema graph G, pattern graph size Limit T*

Output: *a ranked list L of query pattern graphs of Q of size up to T*

```
 1: AC = AtomicConhesivenessConstraints(Q)
 2: RQ := ∅     /* RQ is a queue of intermediate results each associated with a Boolean
                   list indexed by the atomic constraints in AC.
 3: Choose a keyword k in Q
 4: for every node n in G containing k do
 5:    M is initialized to false everywhere /* false indicates that the corresponding
                   atomic constraint has not been checked yet.
 6:    if n contains occurrences of the keywords of a constraint in AC then
 7:       if no atomic constraint is violated by the keyword occurrences in n then
 8:          for every atomic constraint A satisfied by the keyword occurrences in n do
 9:             $M(A) := true$
10:          $enqueue(RQ, (n, M))$
11: while $RQ \neq \emptyset$ do
12:    $(CR, CM)$ := dequeue(RQ)
13:    for every node cn in G linked to some node in $CR$ do
14:       construct a new pattern result graph $NR$ by adding cn and its connecting edge to
          $CR$
15:       $NM := CM$
16:       if $NR$ satisfies all the restrictions imposed on pattern graphs including the size
          restriction then
17:          if $NR$ contains occurrences of the keywords of an atomic constraint A in AC
             s.t. $NM(A) = false$ then
18:             if no atomic constraint A in $NR$ s.t. $NM(A) = false$ is violated by the
                keyword occurrences in $NR$ then
19:                for every atomic constraint A s.t. $NM(A) = false$ satisfied by the keyword
                   occurrences in $NR$ do
20:                   $NM(A) := true$
21:                if $NR$ contains all the keywords in Q then
22:                   put $NR$ in the right rank in L
23:                else
24:                   $enqueue(RQ, (NR, NM))$
```

5.4.1 Experimental Setting

Databases. We used both a version of the IMDB database[1] and a version of the Mondial database[2] in our experiments. The IMDB database contains actor, director and role information for movies. Its schema comprises seven tables which are connected with six foreign keys. The Mondial database contains geographic and demographic information. The schema of this database contains thirty-three tables and forty-nine foreign keys. Note that the number of nodes and edges in the expanded schema graph for a given query is usually a multiple of the respective numbers in the database schema graph.

Queries. For the effectiveness and performance evaluation experiments, we generated 10 cohesive queries with different characteristics for each one of the databases. For the scalability experiments we varied the number of constraints in a number of cohesive (and flat keyword) queries and the number of keyword instances on the databases. And for the effectiveness experiences, each of the databases has ten queries which are chosen by the users with the semantic meaning cohesive constraints.

5.4.2 Effectiveness Experiments

We run experiments to assess the quality of our approach. Our algorithm returns, for a given cohesive query Q and a database D, the patterns graphs for Q on D ranked based on the ranking criteria presented in Section 5.3. To measure the quality of the results of a query we collected the pattern graphs in the top rank of the ranked list of pattern graphs and we measured precision in this set of pattern graphs. Precision indicates the percentage of relevant pattern graphs in the set of pattern graphs considered. In case the top rank had too few pattern graphs, more than one top ranks in the returned ranked

[1]https://relational.fit.cvut.cz/dataset/IMDb
[2]https://www.dbis.informatik.uni-goettipeii ngen.de/Mondial/

list were considered. The ground truth for the query results (that is, the relevant pattern graphs) were determined by a user not related to this project. We denote this metric as $precision@TopRank$.

For every cohesive keyword query considered, we also measured the $precision@TopRank$ of the corresponding flat keyword query and we compared the two measurements. In case the cohesive query did not have any pattern graphs in the top rank of the flat keyword query (for instance because all the pattern graphs of the flat keyword query in this rank violated the cohesiveness constraints of the corresponding cohesive query) the top rank of the cohesive query was used for this query and all the top ranks above and including this one were used for the flat keyword query.

We used a set of 10 queries for each database for evaluation. The queries have from four to seven keywords. The queries may contain keywords which match both schema elements (attributes and/or relation names) and tuple values. Tables 5.1 and 5.3 show the queries used in the experiments and Tables 5.2 and 5.4 show statistics for them on the IMDB and the Mondial databases, respectively. Column #kws records the number of keywords in the query. Column #ccs records the number of cohesiveness constraints and column #accs the number of atomic cohesiveness constraints in the query. Columns #nodes and #edges records the number of nodes and edges on the expanded schema graph. Column #ccpgs denotes the number of pattern graphs of the cohesive keyword query on the database and column #pgs denotes the number of pattern graphs of the corresponding flat keyword query (the flat keyword query obtained by removing the cohesiveness constraints) on the database. As expected, the cohesive queries have less pattern graphs than their corresponding flat keyword queries, and in some cases, orders of magnitude less pattern

Table 5.1 Keyword Queries on the IMDB Database

Query#	**Cohesive keyword query**
Q1	(Movie Musical) Grandfather Tony
Q2	(Movie 2000) Halloween Rafael Adams
Q3	Reunion (actors cuadro) Comedy
Q4	(Anne Brown movies) (actors cuadro)
Q5	(actors (Aabel Steve)) 352881 Crime Kodanda
Q6	(2012 David) (actors Musical movies)
Q7	(Aagaard Valentine) actors roles movies
Q8	17485 (movie Musical) actors
Q9	movie home (Flamenco Antonio) cuadro
Q10	((Anne Adams) Rafael) (movies Musical) (cuadro 1664)

Table 5.2 Statistics for the Queries on the IMDB Database

Query#	**#kws**	**#ccs**	**#accs**	**#nodes**	**#edges**	**#pgs**	**#ccpgs**
Q1	4	1	2	24	97	2956	1969
Q2	5	1	3	32	169	30665	5757
Q3	4	1	2	19	57	1266	733
Q4	5	2	9	24	90	9795	453
Q5	6	2	10	27	96	808	40
Q6	6	2	9	27	105	16314	3752
Q7	5	1	3	24	85	13405	13405
Q8	4	1	2	25	88	2939	1892
Q9	5	1	3	24	100	8570	75
Q10	7	4	23	35	185	89346	3368

graphs. The missing pattern graphs are those that violate some cohesiveness constraint of the cohesive query.

Figures 5.2 and 5.3 show the precision at top rank for the queries of Tables 5.2 and 5.4 on two databases. As one can see, the *precision@TopRank* for the cohesive queries is at least 81% on the IMDB database and at least 75% on the Mondial database. The *precision@TopRank* for the flat keyword queries is much smaller, sometimes only a small fraction of the *precision@TopRank* of the flat keyword queries. This is expected since the cohesiveness constraints capture part of the intention of the user and are used to filter a good number of irrelevant pattern graphs.

Table 5.3 Keyword Queries on the Mondial Database

Query#	**Cohesive keyword query**
Q1	(Ammersee Ammer) ((Baffin Island)1.478
Q2	(Nordrhein (Sibirian Baffin)) (transitional government politics)
Q3	((Limmat 1110) 45095300) ((parliamentary democracy) country)
Q4	(Nordrhein Westfalen) (Ammersee Ammer) (Baffin Island)
Q5	(1.478 Limmat) ((parliamentary democracy) Aberconwy)
Q6	(parliamentary democracy) ((transitional Sibirian) 1110)
Q7	(American Canada mountain)(Gannett rocky)
Q8	(republic Government) (politics country)(Atlantic Ocean)
Q9	(Nordrhein Westfalen) Limmat (Baffin Island) Amazonas
Q10	Norwegian (((Colwyn river) Amazonas) lake borders)

Table 5.4 Statistics for the Queries on the Mondial Database

Query#	**#kws**	**#ccs**	**#accs**	**#nodes**	**#edges**	**#pgs**	**#ccpgs**
Q1	5	3	7	74	332	15110	4400
Q2	6	3	10	50	123	67	37
Q3	6	4	11	50	107	128	44
Q4	6	3	12	63	229	9962	9962
Q5	5	3	7	60	202	31468	9788
Q6	5	3	7	44	80	68	23
Q7	5	2	6	50	187	13899	2079
Q8	6	3	12	70	282	85201	3398
Q9	6	2	8	69	296	76439	2489
Q10	5	3	17	77	235	61468	2145

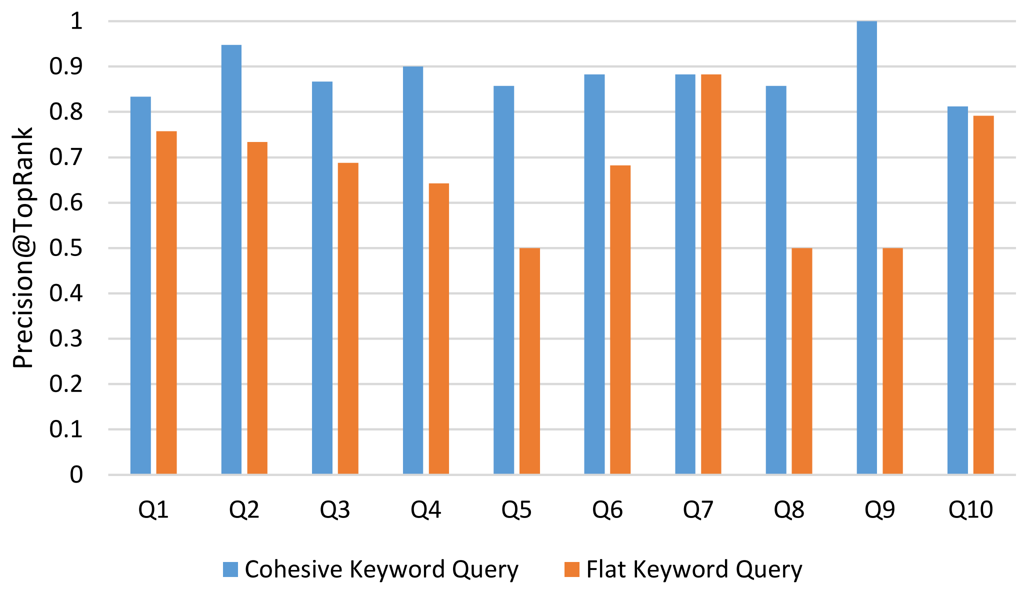


Figure 5.2 Precision at top rank for the queries on the IMDB database

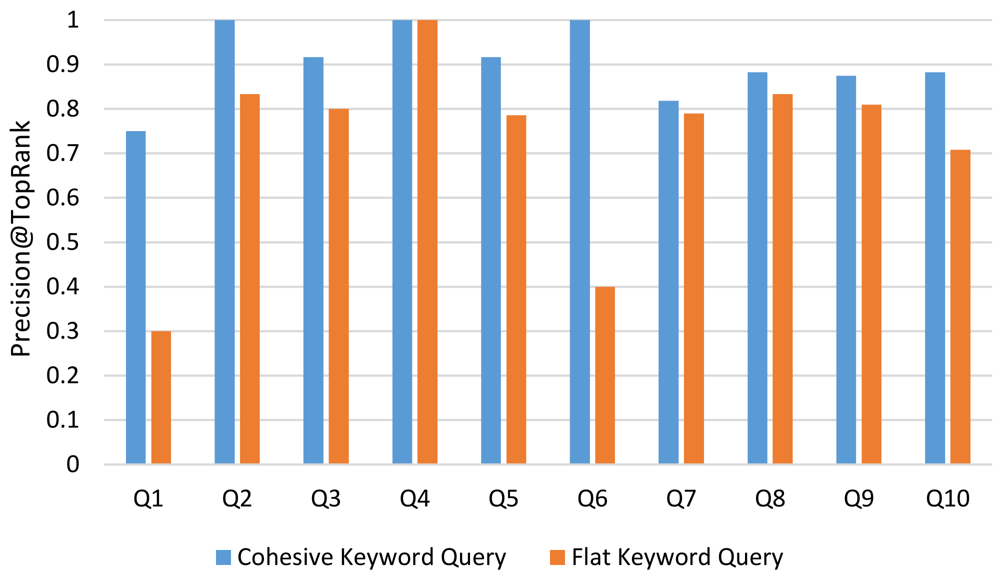


Figure 5.3 Precision at top rank for the queries on the Mondial database.

5.4.3 Efficiency Experiments

We run experiments to measure the execution time of the algorithm on cohesive and flat keywords queries, the number of intermediate results produced, the memory consumption, and its scalability.

Execution time. Figures 5.4 and 5.5 show the execution time for the queries shown in Tables 5.1 and 5.3 on the IMDB and Mondial databases, respectively. We measured

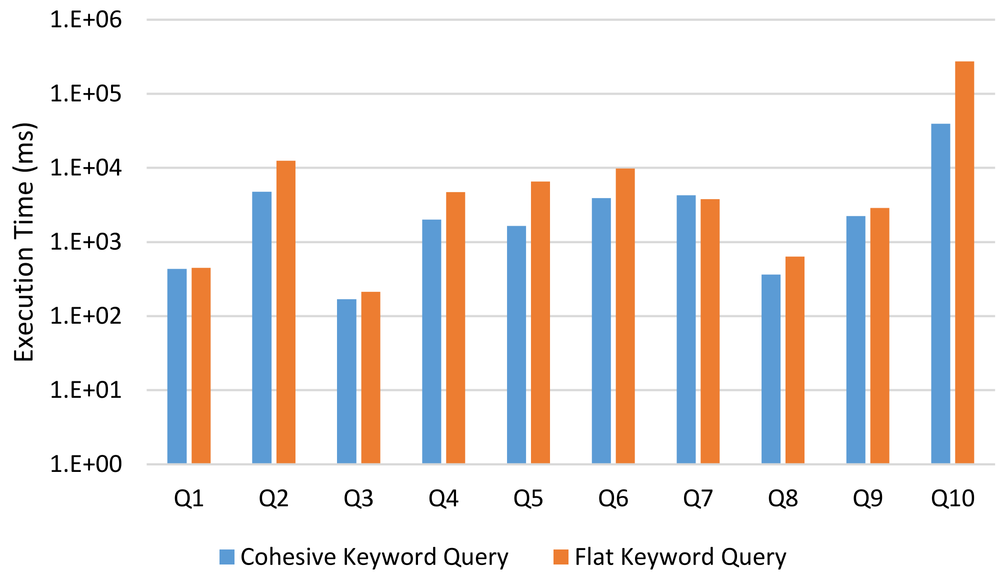


Figure 5.4 Execution time for the queries on the IMDB database.

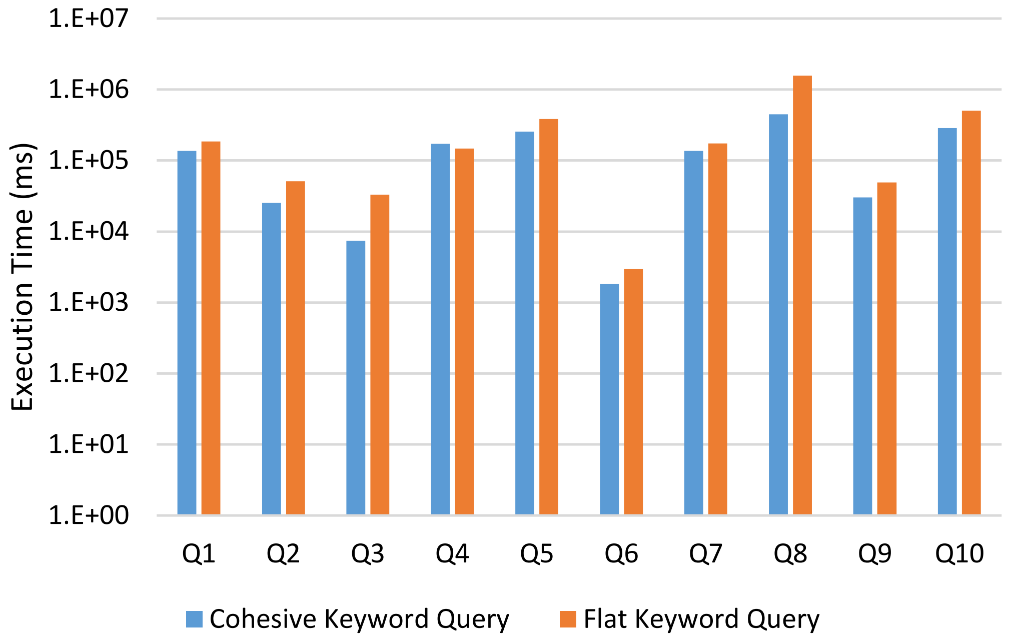


Figure 5.5 Execution time for the queries on the Mondial database.

and displayed both the execution time of the cohesive queries and the execution time of the corresponding flat keyword queries. The scale of the y-axis is logarithmic. For evaluating the cohesive queries we employed Algorithm 3. For evaluating flat keyword queries, we used the same algorithm modified so that it does not check the intermediate results for cohesiveness constraint violations. As one can see, the cohesive keyword queries are substantially faster than the flat keyword queries. This can be explained from the fact that the evaluation algorithm discontinues the expansion of intermediate pattern graphs as soon as they are discovered to violate an atomic cohesiveness constraint. This prunes the search space of intermediate results of the flat keyword queries and substantially reduces its size. The only exceptions are queries Q_7 on the IMDB database and Q_4 on the Mondial database which are slightly slower than their corresponding flat keyword queries, and this is explained in the next paragraph.

Number of intermediate results. Figures 5.6 and 5.7 show the number of intermediate pattern graphs produced during the execution of the algorithm on the two databases both for the cohesive queries and their flat keyword query versions. The scale of the y-axis is logarithmic. The cohesive queries produce much fewer intermediate results than the corresponding flat keyword queries. One can see that the number of intermediate pattern graphs produced is strongly correlated with the execution time of the queries shown in Figures 5.4 and 5.5 for both types of queries. The reason is that the execution time is determined mainly by the number of intermediate results that need to be produced. The pruning of the search space of flat keyword queries achieved using the cohesiveness constraints equally affects the execution time of cohesive queries. Cohesive query Q_7 on the IMDB and Q_4 on the Mondial database produce the same number of intermediate pattern graphs as the corresponding flat keyword query. This is so as no pattern graph for the flat keyword query violates any of the cohesiveness constraints of the cohesive query. However, as shown in Figure 5.5 the execution times of Q_7 and Q_4 are slightly higher. This is due to the overhead the checking of the cohesiveness constraints inflicts on the evaluation of the cohesive pattern graphs.

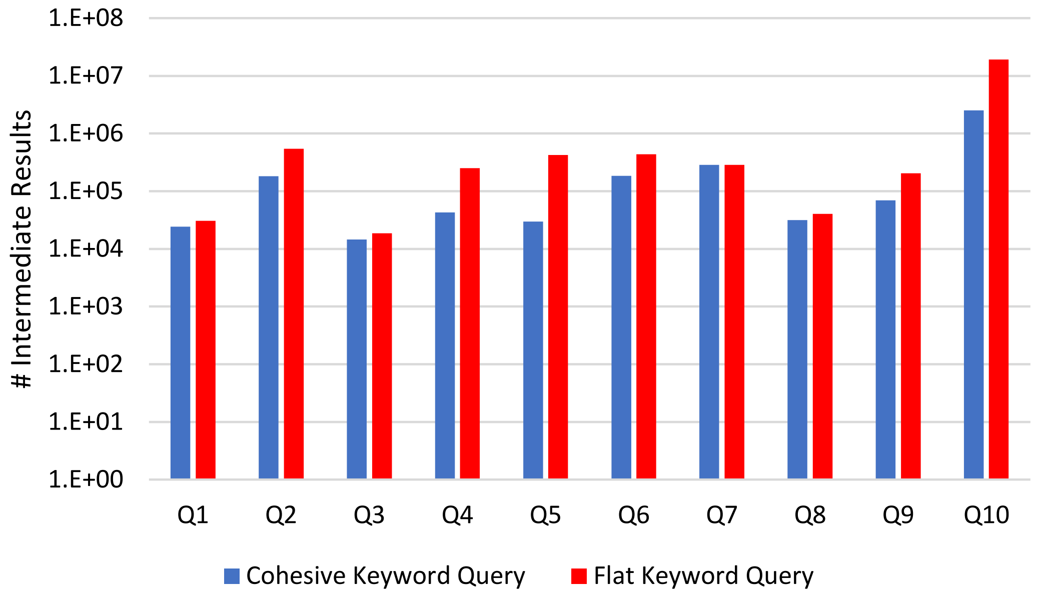


Figure 5.6 Number of intermediate results for the queries on the IMDB database.

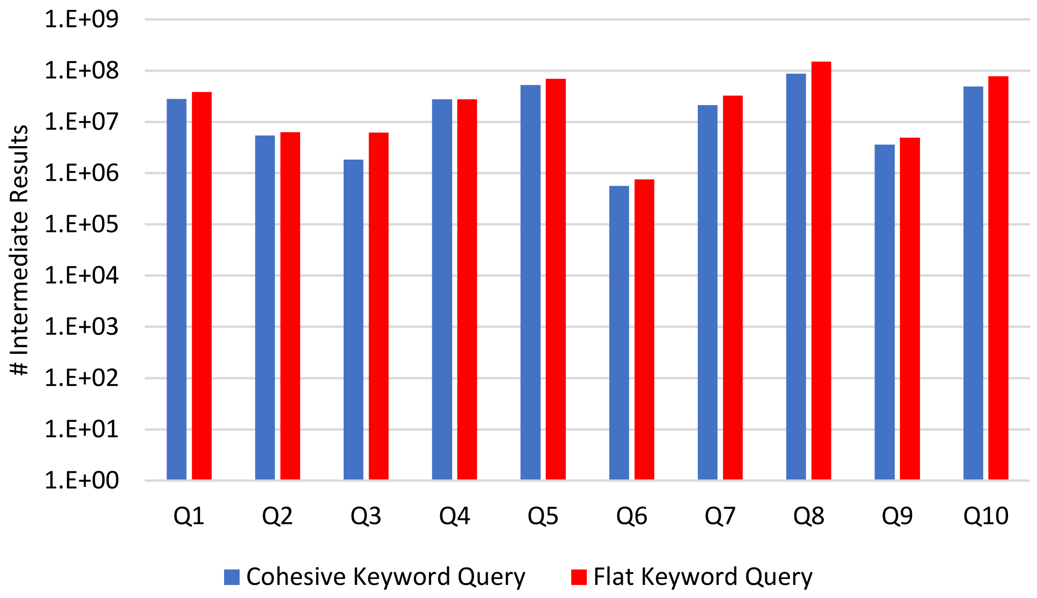


Figure 5.7 Number of intermediate results for the queries on the Mondial database.

Memory consumption. We measured the memory consumption of our algorithm. As this depends on the maximum number of intermediate pattern graphs stored at any moment in time during the execution of the algorithm, we measured the memory consumption in terms of the maximum size of the queue RQ of intermediate pattern graphs in Algorithm

CKER (Algorithm 3). We compare the memory consumption of Algorithm *CKER* to that of the algorithm that computes flat keyword queries. Figures 5.8 and 5.9 show the memory consumption of the cohesive queries shown above and their flat counterparts on the IMDB and the Mondial databases, respectively. The scale of the y-axis is logarithmic. As one can see, the reduction in the search space of intermediate pattern graphs for flat keyword queries inflicted by the cohesiveness constraints also reflects on the memory footprint which is substantially reduced for cohesive queries compared to flat keyword queries.

Scalability varying the number of cohesiveness constraints. We also run experiments to measure how the query execution time and the number of intermediate results scale when the number of cohesiveness constraints in a query increases. For this experiment, we considered a cohesive keyword query with six keywords and four cohesiveness constraints for each one of the two databases and we gradually produced multiple (relaxed) versions of the query which have less cohesiveness constraints, including the corresponding flat keyword query (which has 0 constraints). We measured the execution time of all these queries and

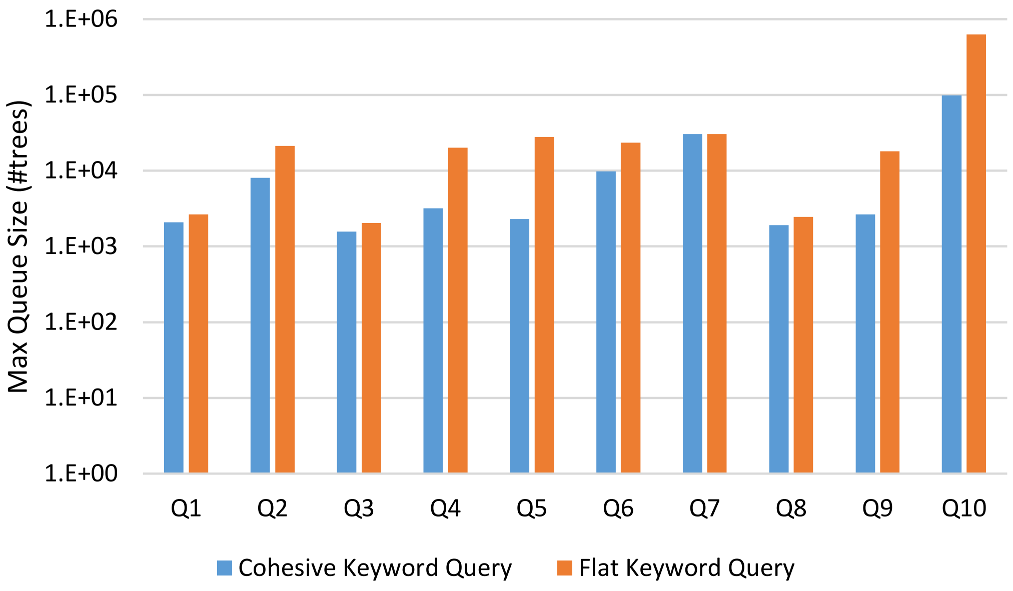


Figure 5.8 Memory consumption for the queries on the IMDB database.

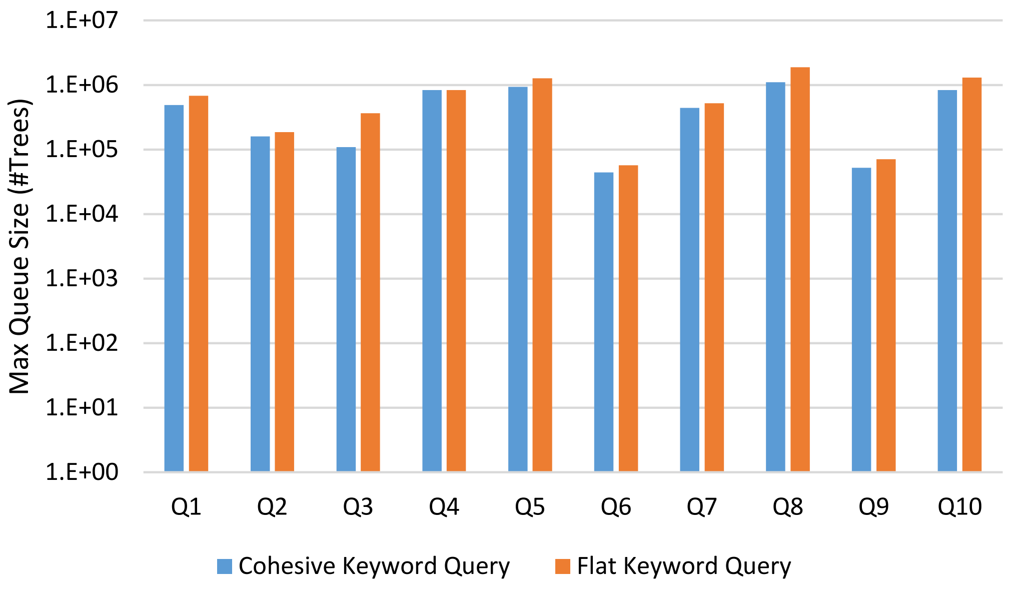


Figure 5.9 Memory consumption for the queries on the Mondial database.

the number of intermediate pattern graphs they produce and averaged the measurements for the queries with the same number of constraints for each one of the databases.

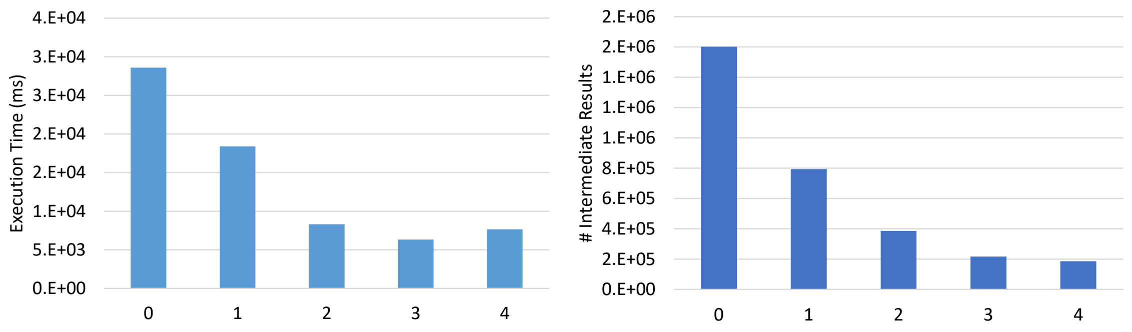


Figure 5.10 Average execution time and average number of Intermediate results for queries on the IMDB database increasing the number of cohesiveness constrains from 0 to 4.

Figures 5.10 and 5.11 display the average execution time and the average number of intermediate pattern graphs when the number of cohesiveness constraints in the queries increases from 0 to 4 in the two databases. One can see that both the average execution time and the average number of intermediate results decreases, in most cases sharply, when the number of cohesiveness constraints increases. This is expected since the additional constraints prune intermediate results early on in the keyword query evaluation process.

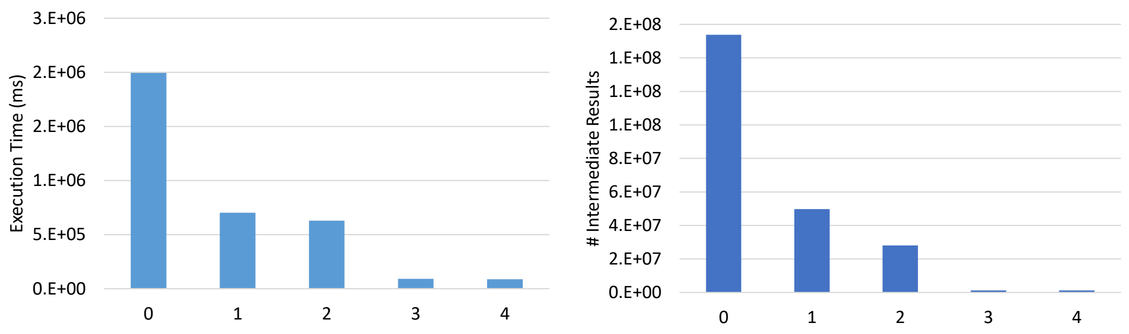


Figure 5.11 Average execution time and average number of Intermediate results for queries on the Mondial database increasing the number of cohesiveness constrains from 0 to 4.

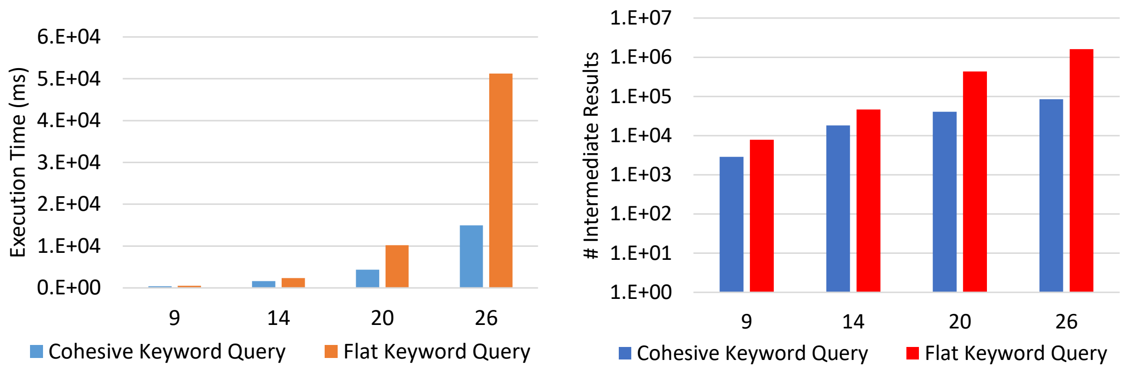
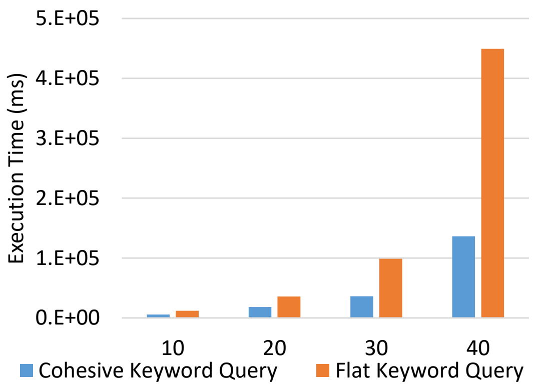


Figure 5.12 Execution time and number of intermediate results a query on the IMDB database increasing the total number of query keyword occurrences.

The only exception is the average execution time of the queries on the IMDB database when the number of cohesiveness constraints increases from 3 to 4. In this case, the average execution time increases instead of decreasing. This can be explained by the fact that even though there is a slight decrease in the number of intermediate results in this case, the increase in the overhead for checking the satisfaction of the atomic cohesiveness constraints (which are now more numerous) in each intermediate pattern graph exceeds the benefit from the reduction of the number of intermediate pattern graphs.

Scalability varying the total number of keyword occurrences of the cohesive query in the database. Finally, we run experiments to measure how the query execution time and the number of intermediate results scale when the total number of keyword



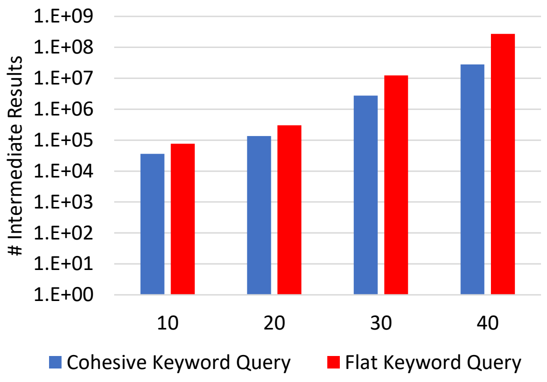


Figure 5.13 Execution time and number of intermediate results a query on the Mondial database increasing the total number of query keyword occurrences.

occurrences of the cohesive query in the database varies. In this last experiment we also compared the results with those of the corresponding flat keyword query.

Figures 5.12 and 5.13 show the execution time and the number of intermediate pattern graphs of query on the two databases when the total number of occurrences of the query increases from 9 to 26 in the IMDB database and from 10 to 40 in the Mondial database. The scale of the y-axis in the plots that display the number of intermediate pattern graphs increases is logarithmic. As one can see, both metrics scale smoothly with the cohesive query, much smoother than with the corresponding flat keyword query.

Conclusion. In summary, cohesive keyword queries execute much faster than the corresponding flat keyword queries as they produce far fewer intermediate pattern graphs. They also consume substantially less memory space and they scale much smoother than flat keyword queries on larger expanded schema graphs.

CHAPTER 6

CONCLUSION

Keyword search has been seen for several years as an attractive way for querying data with some form of structure. Indeed, it allows simple users to extract information from databases without mastering a complex structured query language and without having knowledge of the schema of the data. It also allows for the integrated search of heterogeneous data sources. However, as keyword queries are ambiguous and not expressive enough, keyword search cannot scale satisfactorily on big datasets (performance scalability problem) and the answers are, in general, of low accuracy (query answer quality problem). Therefore, flat keyword search alone cannot efficiently return high-quality results on large data with structure. In this dissertation, we improved keyword search over databases by designing efficient keyword query evaluation algorithms, by exploiting semantic information of the data, and by extending flat keyword queries with semantic information.

We designed an algorithm for keyword search over graph databases which exploits techniques developed for mining tree patterns. We focused on avoiding the generation of redundant intermediate results when the keyword queries are evaluated, which is the bottleneck of query evaluation algorithms. We defined a canonical form for the isomorphic representations of the intermediate results and we showed how it can be checked incrementally and efficiently. We devised rules that prune the search space without sacrificing completeness and we integrated them in a query evaluation algorithm. Our experiments show that our algorithm outperforms previous algorithms by one to two orders of magnitude in terms of efficiency and memory consumption and displays smooth scalability.

Furthermore, we leveraged semantic information of the data to address the aforementioned problems and improve the effectiveness and efficiency of keyword search on relational databases. We follow a schema-based approach for evaluating keyword queries on relational databases, which computes patterns mapped onto the schema graph of the database. Pattern graphs summarize and cluster query results. They are representatives for clusters of query results. As such, they are much fewer than the actual query results and can be translated into SQL queries on the relational database which can produce the results in the cluster. Contrary to traditional methods, our novel approach introduces schema components in the pattern graphs, which also capture key-foreign key relationships and inclusion relationships. We employed information-retrieval-based and semantics-based techniques for scoring query pattern graphs and design an efficient top-k algorithm for computing the patterns graphs of a keyword query. An extensive experimental evaluation demonstrates the effectiveness of our approach and the time and memory efficiency of our algorithm.

Finally, we studied employing keyword queries enhanced with cohesiveness constraints (cohesive keyword queries) to query relational databases. Cohesive keyword queries bridge the gap between flat keyword queries and structured queries. Cohesive queries allow the user to flexibly and effortlessly convey her intention using cohesive keyword groups. A cohesive group of keywords in a query indicates that the keywords of the group should form a cohesive whole in the query results. We formally defined semantics for cohesive queries on relational databases and designed an efficient evaluation algorithm which relies on the extended database schema to generate pattern graphs that satisfy the cohesiveness constraints. Our experiments demonstrate the efficiency of our algorithm and the effectiveness of cohesive keyword queries in improving the result quality and in pruning the space of pattern graphs compared to flat keyword queries. Most importantly, these improvements are attained without compromising the simplicity and convenience of traditional keyword search.

The work presented in this dissertation can be extended by considering and leveraging additional semantic information from the relational database. New algorithms will be then needed for efficiently computing the new pattern graphs. Another research direction would leverage cohesiveness constraints for exploratory search. The system would extract and suggest cohesive groups until a cohesive query that best represents the user's intention is constructed from an initial flat keyword query. As cohesive queries are data model independent, this technique can be used for integrated exploratory search of the heterogeneous data sources.

BIBLIOGRAPHY

[1] Hilit Achiezra, Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Exploratory keyword search on data graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1163–1166, 2010.

[2] Sanjay Agrawal, Kaushik Chakrabarti, Surajit Chaudhuri, Venkatesh Ganti, Arnd Christian Konig, and Dong Xin. Exploiting web search engines to search structured databases. In *Proceedings of the 18th international conference on World wide web*, pages 501–510. ACM, 2009.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 5–16. IEEE, 2002.

[4] Cem Aksoy, Ananya Dass, Dimitri Theodoratos, and Xiaoying Wu. Clustering query results to support keyword search on tree data. In *Web-Age Information Management - 15th International Conference, WAIM 2014, Macau, China, June 16-18, 2014. Proceedings*, pages 213–224, 2014.

[5] Cem Aksoy, Aggeliki Dimitriou, and Dimitri Theodoratos. Reasoning with patterns to effectively answer XML keyword queries. *VLDB J.*, 24(3):441–465, 2015.

[6] Sihem Amer-Yahia, Emiran Curtmola, and Alin Deutsch. Flexible and efficient XML search with complex full-text predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 575–586, 2006.

[7] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. Flexpath: Flexible structure and full-text querying for XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 83–94, 2004.

[8] Sihem Amer-Yahia and Senjuti Basu Roy. Interactive exploration of composite items. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 513–516. OpenProceedings.org, 2018.

[9] Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition.* Pearson Education Ltd., Harlow, England, 2011.

[10] Zhifeng Bao, Yong Zeng, H. V. Jagadish, and Tok Wang Ling. Exploratory keyword search with interactive input. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 871–876, 2015.

[11] Mikhail Bautin and Steven Skiena. Concordance-based entity-oriented search. *Web Intelligence and Agent Systems: An International Journal*, 7(4):303–319, 2009.

[12] Sonia Bergamaschi, Elton Domnori, Francesco Guerra, Raquel Trillo Lado, and Yannis Velegrakis. Keyword search over relational databases: a metadata approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 565–576. ACM, 2011.

[13] Sonia Bergamaschi, Francesco Guerra, and Giovanni Simonini. Keyword search over relational databases: Issues, approaches and open challenges. In *Bridging Between Information Retrieval and Databases - PROMISE Winter School 2013, Bressanone, Italy, February 4-8, 2013. Revised Tutorial Lectures*, pages 54–73, 2013.

[14] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 431–440. IEEE, 2002.

[15] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language. 2002.

[16] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 27(2):153–187, 2002.

[17] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. The onion technique: indexing for linear optimization queries. In *ACM Sigmod Record*, volume 29, pages 391–402. ACM, 2000.

[18] Liang Jeff Chen and Yannis Papakonstantinou. Supporting top-k keyword search in xml databases. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 689–700. IEEE, 2010.

[19] Yi Chen, Wei Wang 0011, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.

[20] Yi Chen, Wei Wang, and Ziyang Liu. Keyword-based search and exploration on databases. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1380–1383. IEEE, 2011.

[21] Tao Cheng, Kevin Chen-Chuan Chang, et al. *Entity Search Engine: Towards Agile Best-Effort Information Integration over the Web*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[22] Yun Chi, Yirong Yang, Yi Xia, and Richard R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings*, pages 63–73, 2004.

[23] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml. In *Proceedings of the 29th international conference on Very Large Databases-Volume 29*, pages 45–56. VLDB Endowment, 2003.

[24] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S Sudarshan. Keyword search on external memory data graphs. *Proceedings of the VLDB Endowment*, 1(1):1189–1204, 2008.

[25] Ananya Dass, Cem Aksoy, Aggeliki Dimitriou, and Dimitri Theodoratos. Exploiting semantic result clustering to support keyword search on linked data. In *Web Information Systems Engineering - WISE 2014 - 15th International Conference, Thessaloniki, Greece, October 12-14, 2014, Proceedings, Part I*, pages 448–463, 2014.

[26] Ananya Dass, Cem Aksoy, Aggeliki Dimitriou, and Dimitri Theodoratos. Relaxation of keyword pattern graphs on RDF data. *J. Web Eng.*, 16(5&6):363–398, 2017.

[27] Elena Demidova, Xuan Zhou, and Wolfgang Nejdl. Iq$^{\mathrm{p}}$: Incremental query construction, a probabilistic approach. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 349–352, 2010.

[28] Elena Demidova, Xuan Zhou, and Wolfgang Nejdl. A probabilistic scheme for keyword-based incremental query construction. *IEEE Trans. Knowl. Data Eng.*, 24(3):426–439, 2012.

[29] Martin Dillon. *Introduction to modern information retrieval.* Pergamon, 1983.

[30] Aggeliki Dimitriou, Ananya Dass, Dimitri Theodoratos, and Yannis Vassiliou. Cohesive keyword search on tree data. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*, pages 137–148, 2016.

[31] Aggeliki Dimitriou, Dimitri Theodoratos, and Timos Sellis. Top-k-size keyword search on tree structured data. *Information Systems*, 47:178–193, 2015.

[32] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 836–845. IEEE, 2007.

[33] Shady Elbassuoni and Roi Blanco. Keyword search over RDF graphs. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 237–242, 2011.

[34] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 656–665, 2008.

[35] Jianhua Feng, Guoliang Li, and Jianyong Wang. Finding top-k answers in keyword search over relational databases using tuple units. *IEEE Transactions on Knowledge and Data Engineering*, 23(12):1781–1794, 2011.

[36] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[37] Lise Getoor and Christopher P Diehl. Link mining: a survey. *Acm Sigkdd Explorations Newsletter*, 7(2):3–12, 2005.

[38] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 927–940, 2008.

[39] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27. ACM, 2003.

[40] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 305–316. ACM, 2007.

[41] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proceedings of the 29th international conference on Very Large Databases-Volume 29*, pages 850–861. VLDB Endowment, 2003.

[42] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 670–681. Elsevier, 2002.

[43] Haoliang Jiang, Haixun Wang, S Yu Philip, and Shuigeng Zhou. Gstring: A novel approach for efficient search in graph databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 566–575. IEEE, 2007.

[44] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very Large Databases*, pages 505–516. VLDB Endowment, 2005.

[45] Mehdi Kargar, Aijun An, Nick Cercone, Parke Godfrey, Jaroslaw Szlichta, and Xiaohui Yu. Meaningful keyword search in relational databases with large and complex schema. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 411–422. IEEE, 2015.

[46] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 779–790. ACM, 2004.

[47] Thuy Ngoc Le and Tok Wang Ling. Survey on keyword search over XML documents. *SIGMOD Record*, 45(3):17–28, 2016.

[48] Fei Li, Tianyin Pan, and Hosagrahar Visvesvaraya Jagadish. Schema-free SQL. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1051–1062, 2014.

[49] Guoliang Li, Chen Li, Jianhua Feng, and Lizhu Zhou. SAIL: structure-aware indexing for effective and progressive top-k keyword search over XML documents. *Information Sciences*, 179(21):3745–3762, 2009.

[50] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and

structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 903–914, 2008.

[51] Yunyao Li, Cong Yu, and H. V. Jagadish. Enabling schema-free xquery with meaningful query focus. *VLDB J.*, 17(3):355–377, 2008.

[52] Yunyao Li, Cong Yu, and HV Jagadish. Schema-free xquery. In *Proceedings of the Thirtieth international conference on Very Large Databases-Volume 30*, pages 72–83. VLDB Endowment, 2004.

[53] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. *Data Exploration Using Example-Based Methods.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[54] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. Example-based search: a new frontier for exploratory search. In *Proceedings of the 42nd International ACM SIGIR Conference*, pages 1411–1412, 2019.

[55] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2006.

[56] Jian Liu and D. L. Yan. Answering approximate queries over XML data. *IEEE Trans. Fuzzy Systems*, 24(2):288–305, 2016.

[57] Xiping Liu, Changxuan Wan, and Lei Chen. Returning clustered results for keyword search on XML documents. *IEEE Trans. Knowl. Data Eng.*, 23(12):1811–1825, 2011.

[58] Ziyang Liu and Yi Chen. Return specification inference and result clustering for keyword search on XML. *ACM Trans. Database Syst.*, 35(2):10:1–10:47, 2010.

[59] Ziyang Liu and Yi Chen. Processing keyword search on XML: a survey. *World Wide Web*, 14(5-6):671–707, 2011.

[60] Xinge Lu, Dimitri Theodoratos, and Aggeliki Dimitriou. Leveraging pattern mining techniques for efficient keyword search on data graphs. In *International Conference on Web Information Systems Engineering*, pages 98–114. Springer, 2020.

[61] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 115–126. ACM, 2007.

[62] Federica Mandreoli, Riccardo Martoglia, and Wilma Penzo. Approximating expressive queries on graph-modeled data: The gex approach. *Journal of Systems and Software*, 109:106–123, 2015.

[63] Federica Mandreoli, Riccardo Martoglia, Giorgio Villani, and Wilma Penzo. Flexible query answering on graph-modeled data. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 216–227, 2009.

[64] Gary Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[65] Khanh Nguyen and Jinli Cao. Top-k Answers for XML Keyword Queries. *WWW*, 15(5-6):485–515, 2012.

[66] Zaiqing Nie, Ji-Rong Wen, and Wei-Ying Ma. Object-level vertical search. In *CIDR*, pages 235–246, 2007.

[67] Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *1st International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[68] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[69] Jeffrey Pound, Ihab F. Ilyas, and Grant E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 423–434, 2010.

[70] Gerard Salton, Edward A Fox, and Harry Wu. Extended boolean information retrieval. Technical report, Cornell University, 1982.

[71] Dennis Shasha, Jason TL Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–52. ACM, 2002.

[72] Amit Singhal, Chris Buckley, and Manclar Mitra. Pivoted document length normalization. In *Acm sigir forum*, volume 51, pages 176–184. ACM New York, NY, USA, 2017.

[73] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[74] Amit Singhal, Gerard Salton, Mandar Mitra, and Chris Buckley. Document length normalization. *Information Processing & Management*, 32(5):619–633, 1996.

[75] Sandeep Tata and Guy M. Lohman. SQAK: doing more with keywords. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 889–902, 2008.

[76] Arash Termehchy and Marianne Winslett. Keyword search over key-value stores. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*, pages 1193–1194, 2010.

[77] Dimitri Theodoratos and Xiaoying Wu. Assigning semantics to partial tree-pattern queries. *Data & Knowledge Engineering*, 64(1):242–265, 2008.

[78] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 405–416, 2009.

[79] Haixun Wang and Charu C Aggarwal. A survey of algorithms for keyword search on graph data. In *Managing and Mining Graph Data*, pages 249–273. Springer, 2010.

[80] Ryen W. White and Resa A. Roth. *Exploratory Search: Beyond the Query-Response Paradigm.* Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2009.

[81] Xiaoying Wu, Stefanos Souldatos, Dimitri Theodoratos, Theodore Dalamagas, Yannis Vassiliou, and Timos K. Sellis. Processing and evaluating partial tree pattern queries on XML data. *IEEE Trans. Knowl. Data Eng.*, 24(12):2244–2259, 2012.

[82] Dong Xin, Yeye He, and Venkatesh Ganti. Keyword++: A framework to improve keyword search over entity databases. *PVLDB*, 3:711–722, 2010.

[83] Yanwei Xu, Yoshiharu Ishikawa, and Jihong Guan. Effective top-k keyword search in relational databases considering query semantics. In *Advances in Web and Network Technologies, and Information Management*, pages 172–184. Springer, 2009.

[84] Xifeng Yan, Philip S Yu, and Jiawei Han. Substructure similarity search in graph databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 766–777. ACM, 2005.

[85] Cong Yu and H. V. Jagadish. Querying complex structured databases. In *Proceedings of the 33rd International Conference on Very Large Databases, University of Vienna, Austria, September 23-27, 2007*, pages 1010–1021, 2007.

[86] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE Data Engineering Bulletin*, 33(1):67–78, 2010.

[87] Mohammed Javeed Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 71–80, 2002.

[88] Mohammed Javeed Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52, 2005.