Journal of Web Engineering, Vol. 16, No. 5&6 (2017) 363–398
© Rinton Press

RELAXATION OF KEYWORD PATTERN GRAPHS ON RDF DATA

ANANYA DASS
Computer Science Department, New Jersey Institute of Technology
150 Bleeker St, Newark, New Jersey 07102, USA
ad292@njit.edu

CEM AKSOY
Computer Science Department, New Jersey Institute of Technology
150 Bleeker St, Newark, New Jersey 07102, USA
ca64@njit.edu

AGGELIKI DIMITRIOU
School of Electrical and Computer Engineering, National Technical University of Athens
Heroon Polytechniou 9, Athens, 15780, Greece
angela@dblab.ntua.gr

DIMITRI THEODORATOS
Computer Science Department, New Jersey Institute of Technology
150 Bleeker St, Newark, New Jersey 07102, USA
dth@njit.edu

One of the facets of the data explosion in recent years is the growing of the repositories of RDF Data on the Web. Keyword search is a popular technique for querying repositories of RDF graph data. Recently, a number of approaches leverage a structural summary of the graph data to address the typical keyword search related problems of: (a) identifying relevant results among a multitude of candidates, and (b) performance scalability. These approaches compute queries (pattern graphs) corresponding to alternative interpretations of the keyword query and the user selects one that matches her intention to be evaluated against the data. Though promising, these approaches suffer from a drawback: because summaries are approximate representations of the data, they might return empty answers or miss results which are relevant to the user intent. In this paper, we present a novel approach which combines the use of the structural summary and the user feedback with a relaxation technique for pattern graphs. We leverage pattern graph homomorphisms to define relaxed pattern graphs that are able to extract more results potentially of interest to the user. We introduce an operation on pattern graphs and we prove that it is complete, that is, it can produce all relaxed pattern graphs. To guarantee that the result pattern graphs are as close to the initial pattern graph as possible, we devise different metrics to measure the degree of relaxation of a pattern graph. We design an algorithm that computes relaxed pattern graphs with non-empty answers in relaxation order. To improve the successive computation of relaxed pattern graphs, we suggest subquery caching and multiquery optimization techniques adapted to the context of this computation. Finally, we run experiments on different real datasets which demonstrate the effectiveness of our ranking of relaxed pattern graphs, and the efficiency of our system and optimization techniques in computing relaxed pattern graphs and their answers.

Keywords: RDF Data, Semantic Web, Keyword Search, Pattern Graph Relaxation

1. Introduction

In the era of big data, the information stored is growing every minute on the Web. This information is usually unstructured or semistructured and graphs are often an organizational option. Keyword search is the most popular technique for querying data on the Web because it allows the user to retrieve information without knowing any formal query language (e.g., SPARQL) and without being aware of the structure/schema of the data sources against which the keyword query is issued. The same keyword query can be used to extract data from multiple data sources with different structures and this is particularly useful in the web where the data sources that can provide the answers are not known in advance. Unfortunately, the convenience and the simplicity of keyword search come along with a drawback. Keyword queries are imprecise and ambiguous. For this reason, they return a very large number of results. This is a typical problem in IR. However, it is exacerbated in the context of tree and graph data where a result to a query is not a whole document but a substructure (e.g., a subtree, or a subgraph) which exponentially increases the number of results. As a consequence, the keyword search on graph data faces two major challenges: (a) effectively identifying relevant results and (b) coping with the performance scalability issue.

In order to identify relevant results, previous algorithms for keyword search over graph data compute candidate results in an approximate way by considering only those which maintain the keyword instances in close proximity [1, 2, 3, 4, 5, 6, 7, 8]. The filtered results are ranked and top-k processed usually by employing IR-style metrics for flat documents (e.g., tf*idf or PageRank) adapted to the structural characteristics of the data [9, 10, 11, 12, 13]. Nevertheless, the statistics-based metrics alone cannot capture effectively the diversity of the results represented in a large graph dataset neither identify the intent of the user. As a consequence, the produced rankings are, in general, of low quality. In order to cope with the second challenge, the performance scalability issue, current algorithms compute all the results of a certain form whose size is below a certain threshold. Despite the restriction, these algorithms are still of high complexity and they do not scale satisfactorily when the size of the data graph and the number of query keywords increase.

Leveraging the structural summary. In order to address these challenges, recent approaches to keyword search on graph data developed techniques which exploit a structural summary of the data graph [10, 11, 14, 15, 16]. This is a concept similar to the 1-index [17] or data guide [18] in tree databases. The structural summary summarizes the structure of an RDF graph and associates inverted lists of keyword instances (extensions) with nodes. A structural summary is typically much smaller than its data graph. These techniques use the structural summary to produce pattern graphs for a given keyword query. The pattern graphs are structured queries corresponding to interpretations of the imprecise keyword query. Evaluating the pattern graphs on the data graph, the candidate results for the keyword query can be produced. Interestingly, a pattern graph can be expressed as a SPARQL query. Therefore, all the machinery of query engines and optimization techniques developed for SPARQL can be leveraged to efficiently evaluate pattern graphs.

Example 1 Consider the RDF graph D shown in Figure 1(a). This is a database about publications, projects, researchers and universities. Let's assume that the query $Q=\{$`Ananya, NJIT, 2014`$\}$ is issued against the database D. The user is looking for publications by `Ananya` published in `2014` which have an author working for `NJIT`. Figure 1(b) depicts the structural

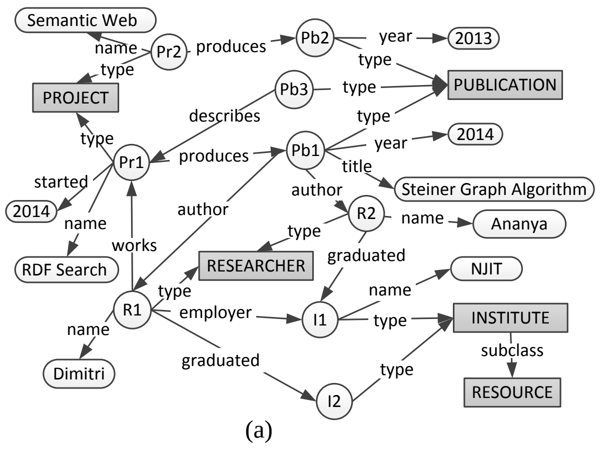
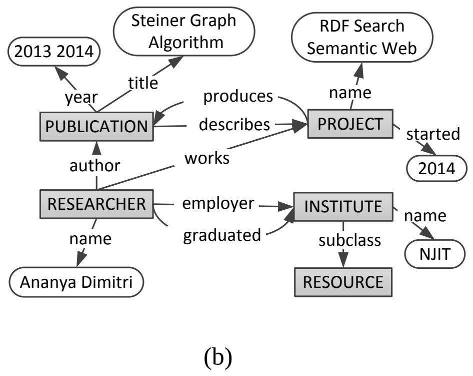



Fig. 1. (a) An RDF data graph D, (b) The structural summary S_D of D

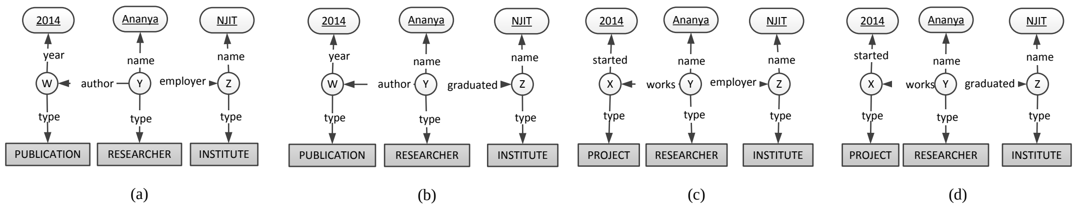


Fig. 2. Pattern graphs for the keyword query {`Ananya, NJIT, 2014`}

summary S_D of D. Finding the instances of the keywords `Ananya, NJIT` and `2014` on S_D, the system will construct pattern graphs. Different algorithms can be employed for this task which aim at constructing pattern graphs by connecting the instances of the keywords in the structural summary in some minimal way [10, 11, 14, 19, 20]. All these algorithms will basically construct the pattern graphs shown in Figure 2. These pattern graphs are structured queries that can be matched against the data graph to produce answers for the keyword queries (X,Y,Z are variable nodes to be matched against entity nodes in the data graph). Notice that these pattern graphs represent different interpretations of the given keyword query. For instance, the first pattern graph looks for publications by `Ananya`, who works at `NJIT`, published in `2014` while the last one is looking for a project initiated in `2014` which employs a researcher named `Ananya` who graduated from `NJIT`. Evaluating these queries on D will return all the results for the initial keyword query Q on D.

Benefits of the structural summary. A structural summary approach can resolve the challenges mentioned above of: (a) effectively identifying relevant results, and (b) coping with the performance scalability issue. Indeed, the pattern graphs (structured queries) can be ranked using a scoring function and the top-k of them are presented to the user. As these structured queries represent different interpretations of the keyword query on the data graph, the user can choose one that meets his intention, and only the corresponding structured query is evaluated against the data graph [10, 11]. A more recent approach exploits a hierarchical clustering of the pattern graphs. In order to select a relevant pattern graph, the user chooses semantic interpretations for the query keywords and only the pattern graphs that correspond to these interpretations are generated and presented to the user [14]. Effectiveness studies show that the approaches based on the structural summary display good precision [11]. Further, computing, ranking and identifying top-k subgraphs (query results) for a keyword query directly on the data graph is very expensive even when answers are computed in an approximate way [1, 2]. In contrast, since the structural summaries are much smaller than the actual data, generating and manipulating relevant pattern graphs can be done efficiently. Therefore, the structural summary-based approaches scale satisfactorily and compute answers of keyword queries efficiently even on large RDF graphs stored in external memory [11, 14, 21].

The missing relevant result problem. Despite its advantages, the structural summary-based approach for keyword search on RDF data has a drawback. The problem is that the pattern graph selected by the user might return no result when evaluated against the RDF graph even though results that match the user intent exist in the RDF graph. It is also possible that the pattern graph returns a non-empty answer but misses relevant results. This might happen even if a pattern graph is correctly selected by the user, that is, even if the selected pattern matches the user intent.

Example 2 In our running example the pattern graph of Figure 2(a) is relevant and is selected by the user. One can see that this pattern graph does not have a match on the RDF graph D shown in Figure 1(a). Indeed, `Ananya` has authored a paper in 2014 but she does not work for `NJIT`. However, there is a result in the data graph D which matches the user intent since there is a publication authored by `Ananya` in `2014` which has an author (the co-author named Dimitri) working for `NJIT`. This relevant result cannot be directly obtained from any of the pattern graphs of Figure 2. It is missed by the structural summary-based approach.

As another example consider the keyword query {`publication, describes, project, produces, Steiner`} on the RDF graph D. The user is looking for a `publication` which `describes` a `project` that `produces` a paper titled `Steiner` graph algorithm. The structural summary-based approach will generate one pattern graph shown in Figure 3(a). As one can easily see, this pattern graph does not have a match on D since there is no publication Y having "Steiner" in its title, is produced by a project X and the same project X is described in the publication Y. However, there is a result in D which matches the user intent. Figure 3(b) shows a subgraph of D which reflects this result. This relevant result is again missed by the structural summary-based approach. The reason for this discrepancy is that the structural summary merges entity vertices of the same type of the RDF data into one vertex and this coarse representation loses information as to how an entity vertex is related to other entity vertices or is assigned properties and values.

Our Approach. In this paper, we provide an approach for keyword search over RDF graph data which addresses the weakness of the structural summary based approach while maintaining its advantages. Our system allows the user to navigate through a clustering hierarchy to select a relevant pattern graph. It then enables the gradual relaxation of this pattern graph so that additional results of interest to the user are retrieved from the RDF graph, if needed

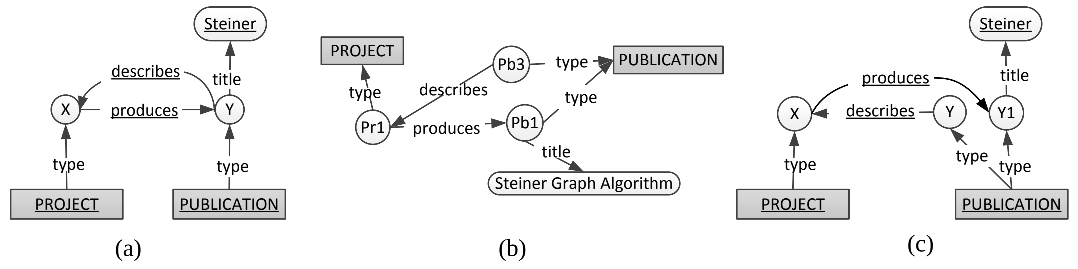


Fig. 3. (a) Pattern graph (b) Result graph (c) Relaxed pattern graph

(for example, if the original pattern graph returns no result or if the user wants to extract more semantically similar results). For instance, in the example of Figure 3, our system will produce the relaxed pattern graph of Figure 3(c) from the pattern graph of Figure 3(a). This relaxed pattern graph can retrieve the relevant result shown in Figure 3(b) missed by the original pattern graph of Figure 3(a).

Contribution. The main contributions of the paper are the following:

- We present a clustering hierarchy for the query results. Our clustering hierarchy allows the user to disambiguate the query and select a relevant pattern graph while examining a small portion of the hierarchy components. To shorten the user interaction we devise ranking techniques for the hierarchy components that take into account structural, and semantic information, and occurrence frequency statistics.

- We leverage pattern graph homomorphisms to define relaxed pattern graphs. Relaxed pattern graphs can expand the result space of an original pattern graph. They can produce additional results of possible interest to the user based on her choice of an original pattern graph.

- We define an operation on pattern graphs (vertex split operation) in order to allow the construction of relaxed pattern graphs. A vertex split operation creates a split image of an entity variable vertex in a pattern graph and partitions its incident edges between the two vertices in order to increase the chances for the relaxed pattern graph to have embeddings to the RDF graph. We prove that this operation is complete, that is it can produce all the relaxed pattern graphs.

- Since we want to relax a pattern graph so that it is as close to the initial pattern graph as possible, we introduce three metrics of decreasing importance to measure the degree of relaxation of a pattern graph. All three metrics take into account structural and semantic characteristics of the relaxed pattern graph and depend on the vertex split operations applied to the original pattern graph.

- If an original pattern graph has an empty answer on an RDF graph, we would like to identify its vertices which contribute to this condition. We call these vertices empty vertices and we provide necessary and sufficient conditions for characterizing them in a pattern graph. Empty vertices are used to guide the relaxation process so that relaxed pattern graphs with non-empty answers are produced.

- We design an algorithm which takes a pattern graph as input and gradually generates relaxed pattern graphs having non-empty answers. The algorithm returns the relaxed patterns graph in ascending order of relaxation as this is defined by the three relaxation metrics mentioned above, and computes their answer on the RDF graph.
- The evaluation of the relaxed pattern graphs in the algorithm requires the concurrent and sequential computation of multiple graph queries which share common sub expressions. We exploit subquery caching and multiquery optimization techniques to suggest a global evaluation plan for the produced relaxed pattern graphs which reduces the repeated computation of common subexpressions.
- In order to measure the effectiveness of our ranking of relaxed pattern graphs, we run experiments with two different metrics on two real datasets. We also run experiments to measure the efficiency of our system in computing relaxed pattern graphs and their answers. The results show that our ranking of relaxed pattern graphs is of high quality, and our system displays interactive times while our optimization techniques substantially reduce the execution time of the algorithm.

Outline. The next section presents our data model and the semantics of keyword queries. Section 3 outlines how pattern graphs can be constructed from the structural summary. Section 4 describes the clustering hierarchy and the selection of pattern graph by the user. Section 5 defines relaxed pattern graphs and the vertex split operation and introduces metrics for measuring the relaxation of pattern graphs. In Section 6, we present our algorithm for computing relaxed pattern graphs and our optimization techniques. The experimental study is reported in Section 7. Related work is discussed in Section 8 while concluding remarks are provided in Section 9.

2. Data Model and Query Language Semantics

We now formally represent our data model. Next, we define keyword queries and their semantics.

2.1. *Data Model*

Resource Description Framework (RDF) provides a framework for representing information about web resources in a graph form. The RDF vocabulary includes elements that can be broadly classified into Classes, Properties, Entities and Relationships. All the elements are resources. RDF has a special class, called *Resource* class and all the resources that are defined in an RDF graph belong to the *Resource* class. Our data model is an RDF graph defined as follows:

Definition 1 (RDF Graph) An *RDF graph* is a quadruple $G = (V, E, L, l)$ where:

V is a finite set of vertices, which is the union of three disjoint sets: V_E (representing entities), V_C (representing classes) and V_V (representing values).

E is a finite set of directed edges, which is the union of four disjoint sets: E_R (inter-entity edges called *relationship* edges representing entity relationships), E_P (entity-to-value edges called *property* edges representing property assignments), E_T (entity-to-class edges called *type* edges representing entity-to-class membership) and E_S (class-to-class edges called *subclass* edges representing class-subclass relationship).

L is a finite set of labels that includes the labels “type”, “subclass” and “resource”.

l is a function from $V_C \cup V_V \cup E_R \cup E_P \cup E_T \cup E_S$ to L. That is, l assigns labels to class and values vertices and to relationship, property, type and subclass edges. In particular, l assigns the label “type” and the label “subclass” to a type and subclass edge respectively.

Entity and class vertex and edge labels are Universal Resource Identifiers (URIs). Vertices are identified by IDs which in the case of entities and classes are URIs. Every entity belongs to a class. Figure 4 shows an example RDF graph (inspired by the Jamendo[a]dataset). For simplicity, vertex and edge identifiers are not shown in the example graph below.

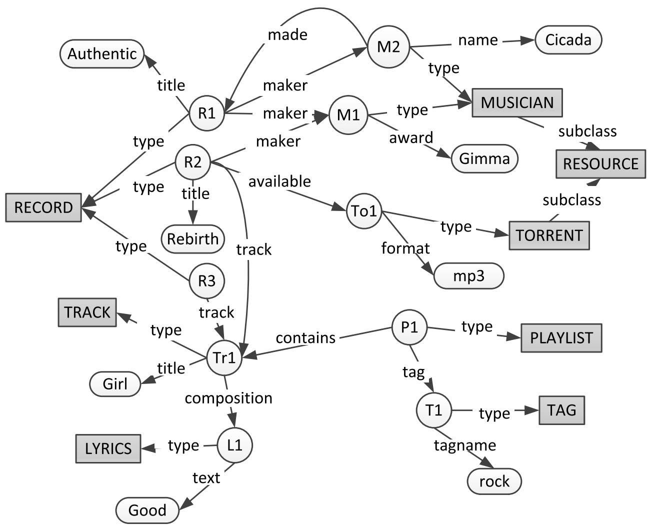


Fig. 4. An RDF graph

2.2. *Queries and Answers*

A *query* is a set of keywords. The *answer* of a query Q on an RDF graph G is a set of subgraphs (*result graphs*) of G, where each result graph involves at least one instance of every keyword in Q. A *keyword instance* of a keyword k in Q is a vertex or edge label containing k. In order to facilitate the interpretation of the semantics of the keyword instances, every instance of keyword in a query is matched against a small subgraph (*matching construct*) of

[a]http://dbtune.org/jamendo/

the graph G which involves this keyword instance. Each matching construct provides a deeper insight about the context of a keyword instance in terms of classes, entities and property and relationship edges. We link one matching construct for every keyword in the query Q through edges (*inter-construct connection*) and common vertices into a connected component to form a *result graph*. A query Q can have multiple signatures, representing different interpretations for the keywords. Each signature can generate multiple result graphs. Next, we provide definitions for the concepts of keyword matching construct, query signature, inter-construct connection and result graph in order to formally define a query answer.

Definition 2 (Matching Construct) Given a keyword k of a query and an RDF graph G, for every instance of k in G, we define a *matching construct* as a small subgraph of G. If the instance i of k in G is:

- the label of a class vertex $v_c \in V_C$, the matching construct of i is the vertex v_c (*class matching construct*).
- the label of a value vertex $v_v \in V_V$, the matching construct of i comprises the value vertex v_v, the corresponding entity vertex, and its class vertices along with the property and type edges between them (*value matching construct*).
- the label of relationship edge $e_r \in E_R$, the matching construct of i comprises the relationship edge e_r, its entity vertices and their class vertices along with the type edges between them (*relationship matching construct*).
- the label of property edge $e_p \in E_P$, the matching construct of i comprises the property edge e_p, its value vertex and entity vertex, and the class vertex of the entity vertex along with the type edges between them (*property matching construct*).

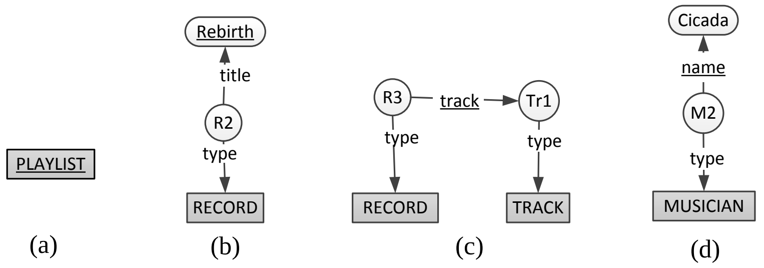


Fig. 5. matching constructs (a) class (b) value (c) relationship (d) property

Figure 5 shows a class, value, relationship and property matching constructs for different keyword instances in the RDF graph of Figure 4. Underlined labels in a matching construct denote the keyword instances on which the matching construct is defined (called *active keyword instances* of the matching construct). This is necessary because a value instance and a property instance can share the same matching construct. Each matching construct provides information about the semantic context of the keyword instance under consideration. For instance, the matching construct of Figure 5(b) shows that `Rebirth` is the title of entity $R2$ of type Record.

Definition 3 (Query Signature) Given a query Q and an RDF graph G, a *signature* S of Q is a function from the keywords of Q that matches every keyword k to a matching construct of k in G.

Figure 5 shows a query signature for the query {`Playlist, Rebirth, track, name`}. Note that a signature of a query Q can have less matching constructs than the keywords in Q, since one matching construct can have more than one active keyword instance.

Definition 4 (Inter-construct Connection) Given a query signature S, an *inter-construct connection* between two distinct matching constructs C_1 and C_2 in S is a simple path augmented with the class vertices of the intermediate entity vertices in the path (if not already in the path) such that: (a) one of the terminal vertices in the path belongs to C_1 and the other belongs to C_2, and (b) no vertex in the connection except the terminal vertices belong to a construct in S.

Figure 6 shows an inter-construct connection between the matching constructs for keywords `Torrent` and `Cicada` in the RDF graph of Figure 4. The matching constructs are shaded and the inter-construct connection is circumscribed.

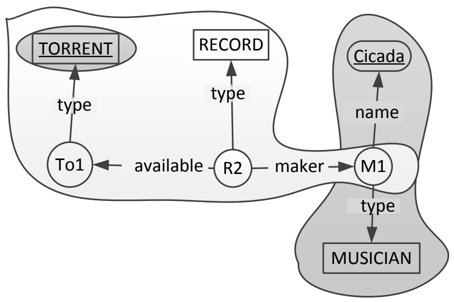


Fig. 6. Inter-construct connection

In order to define result graphs we need the concept of acyclic subgraph with respect to a query signature. Let G_s be a subgraph of the RDF graph that comprises all the constructs in the signature of a query. We construct an undirected graph G_c as follows: there is exactly one vertex in G_c for every matching construct and for every vertex not in a matching construct in G_s. Further: (a) if v_1 and v_2 are non-construct vertices in G_c, there is an edge between v_1 and v_2 in G_c iff there is an edge between the corresponding vertices in G_s, (b) If v_1 is a construct vertex and v_2 is a non-construct vertex in G_c, there is an edge between v_1 and v_2 in G_c iff there is an edge between a vertex of the construct corresponding to v_1 and the vertex corresponding to v_2 in G_s, and (c) if v_1 and v_2 are two construct vertices, there is an edge between them in G_c iff there exists in G_s, an edge between a vertex of the construct corresponding to v_1 and a vertex of the construct corresponding to v_2 that edge does not occur in any one of the constructs. Graph G_s is said to be *connection acyclic* if there is no cycle in G_c.

Consider the query $Q = \{$`Cicada, mp3, Record`$\}$ on the RDF graph G of Figure 4. Figure 7 shows two subgraphs of G which comprise a signature of Q on G. The active keyword instances are underlined and the corresponding matching constructs are shaded. One can see that the subgraph in Figure 7(a) is connection cyclic while the subgraph in Figure 7(b) is connection acyclic.

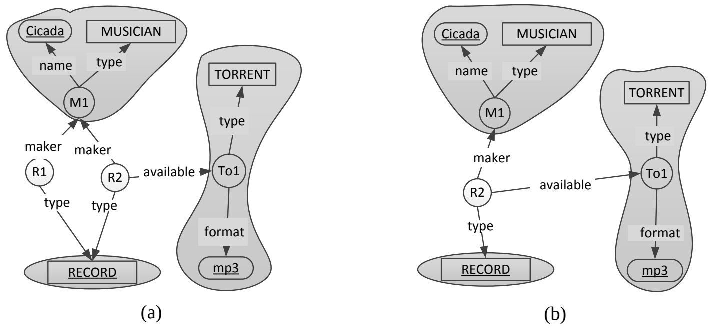


Fig. 7. (a) Invalid result graph (b) Valid result graph for $Q = \{$`Cicada, mp3, Record`$\}$

Note that a result graph can be connection acyclic even if it contains a cycle. One example is shown in Figure 8(b). The keyword instances are underlined and the matching constructs are shown shaded. We can now define result graphs.

Definition 5 (Result Graph) Given a signature S for a query Q over an RDF graph G, a *result graph* of Q for S is a connected connection acyclic subgraph G_R of G which contains only the matching constructs in S and possibly inter-construct connections between them.

Therefore, a result graph of a query contains all the matching constructs of a signature of the query and guarantees that they are linked with inter-construct connections into a connected whole. Note that a result graph might not contain any inter-construct connection (this can happen if every matching construct in the query signature overlaps with some other matching construct). However, if inter-construct connections are used within the result graph, no redundant (cycle creating) inter-construct connections are introduced.

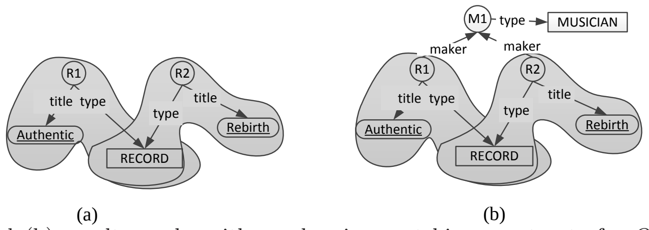


Fig. 8. (a) and (b) result graphs with overlapping matching constructs for $Q = \{$`Authentic, Rebirth`$\}$

Consider the query $Q = \{$`Authentic, Rebirth`$\}$ on the RDF graph G of Figure 4. Figure

8(a) shows a result graph for Q in G that is formed by overlapping matching constructs without any inter-construct connections. The result graph in Figure 8(b) has the same overlapping matching constructs but it also includes an inter-construct connection between them. This is permissible since this subgraph is connection acyclic. We can now define the *answer* of a query.

Definition 6 (Query Answer) The *answer* of a query Q on an RDF graph G is the set of result graphs of Q on G.

3. Computing Pattern Graphs on the Structural Summary

We introduce in this section the structural summary of a data graph and the pattern graphs of a query. Then, we present an algorithm which uses the structural summary for computing the pattern graphs of a query.

3.1. *The Structural Summary and Pattern Graphs*

In order to construct pattern graphs we use the structural summary of the RDF graph. Intuitively, the structural summary of an RDF graph G is a special type of graph which summarizes the data graph showing vertices corresponding to class and value vertices and edges corresponding to the property, relationship and subclass edges in G. Entity vertices are omitted. Roughly speaking, the structural summary can be constructed by merging all the entity vertices of a class with their class vertex and then by merging together all the property edges and all the relationship edges that are of the same kind.

Definition 7 (Structural Summary) The *structural summary* of an RDF graph G is a vertex and edge labeled graph constructed from G as follows:

(a) Merge every class vertex and its entity vertices into one vertex labeled by the class vertex label and remove all the type edges from G.

(b) Merge all the value vertices which are connected with a property edge labeled by the same label to the same class vertex into one vertex labeled by the union of the labels of these value vertices. The label of this new value vertex is called *extent* of the vertex. Merge also the corresponding edges into one edge labeled by their common label.

(c) Merge all the relationship edges between the same class vertices which are labeled by the same label into one edge with that label.

Figure 9(a) shows the structural summary for the RDF graph G of Figure 4. Similarly to matching constructs on the data graph we define matching constructs on the structural summary. We refer to these constructs also as "matching constructs" or MCs. Since the structural summary does not have entity vertices, matching constructs on structural summaries possess one distinct entity variable vertex for every class vertex labeled by a distinct variable. The underlined keywords in matching constructs indicate active keyword instances.

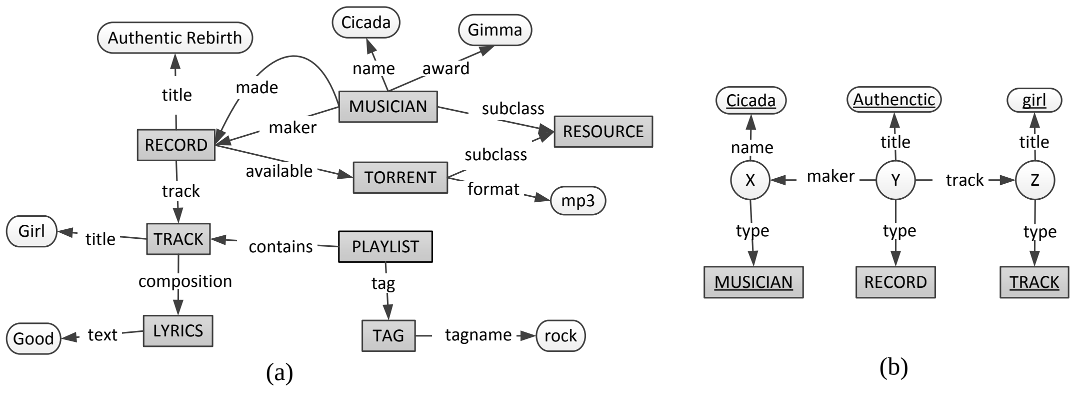


Fig. 9. (a) Structural Summary, (b) Query Pattern Graph

The next objective is to compute result pattern graphs. These are subgraphs of the structural summary, strictly consisting of one matching construct for every keyword in the query Q and the connections between them without these connections forming a cycle.

Definition 8 (Pattern Graph) A *pattern graph* for a keyword query Q is a graph similar to a result graph for Q, with the following two exceptions :

(a) the labels of the entity vertices in the result graph, if any, are replaced by distinct variables in the pattern graph. These variables are called *entity variables* and they range over entity vertices.

(b) The labels of the value vertices are replaced by distinct variables whenever these labels are not the active keyword instances in the result graph. These variables are called *value variables* and they range over value labels in the RDF graph.

Therefore a pattern graph is a connected graph which comprises one matching construct for every keyword. Figure 9(b) shows an example of a pattern graph, for the keyword query $Q = \{$`Cicada, Authentic, Girl`$\}$ on the RDF graph of Figure 4. Labels X, Y, and Z are entity variables.

In order to compute pattern graphs, we use the algorithm presented in [14]. This algorithm takes as input a structural summary and a signature S of a keyword query and produces the pattern graphs for S. The produced pattern graphs are r-radius Steiner graphs [6] whose radius r is minimum. A pattern graph is an r-radius Steiner pattern graph if there is a class vertex in it whose distance from all the other class vertices is r or less and there is no class vertex in it whose distance from all the other vertices is less than r.

4. Selection of a Pattern Graph through Semantic Hierarchical Clustering and Ranking

We describe in this section how the user can chose a relevant pattern graph by navigating through a hierarchy which also ranks the pattern graphs to better support the relevant pattern graph selection process.

Semantic hierarchical clustering. The hierarchy has two levels on top of the result graph layer. The pattern graphs of a query Q on an RDF graph G define a partition of the result graphs of Q on G. The pattern graphs constitute the *first level* of the clustering hierarchy. Multiple pattern graphs can share the same signature. The signatures determine a partition of the pattern graphs of Q on G. They, in turn, define a partition of the results which is coarser than that of the pattern graphs. The signatures constitute the *second (top) level* of the hierarchy.

Hierarchy navigation. In order to navigate through the hierarchy after issuing a query the user starts from the top level. The top level may have numerous signatures. However, the user does not have to examine all the signatures. Instead, she is presented with the MC list for one of the query keywords. We describe below how this list of MCs is ranked. As mentioned earlier, the MCs of a keyword provide all the possible interpretations for this keyword in the data. The user selects the MC that she considers relevant to her intent. Subsequently, she is presented with the MC lists of the other query keywords, though some of the MC lists can be skipped. This can happen if the user selects an MC which involves more than one keyword instances that she wants to see combined together in one MC. Once MCs for all keywords have been selected, that is, a query signature has been determined, the system presents a ranked list of all the pattern graphs that comply with the signature. The user chooses the pattern graph of her preference which is evaluated by the system. The result graphs are returned to the user.

Ranking. The MCs for a keyword are ranked in an MC list based on the following rules: (a) MCs that involve more than one active keyword instances are ranked first in order of the number of active keyword instances they contain, (b) class MCs, relationship MCs and property MCs are ranked next in that order, (c) value MCs follow next and are ranked in descending order of the frequency of their value. The *value frequency* f_m^v of a value MC m with property p, class c and value (keyword) v is the number $n_{p,c}^v$ of occurrences of the value v in matching constructs involving p and c in the data graph divided by the number $n_{p,c}$ of occurrences of property matching constructs in the data graph involving p and c. That is, $f_m^v = n_{p,c}^v / n_{p,c}$.

This ranking of the MCs favors MCs with multiple keyword instances based on the assumption that keywords that occur in close proximity are more relevant to the user's intent. Further, it favors MCs whose active keyword matches a schema element (class, relationship or property), favoring most class MCs which have unique occurrences in the data graph. Finally, value MCs are ranked at the end since they are more specific. The value frequency of a value MC reflects the popularity of this MC in the data. Therefore, value MCs with high value frequency are ranked higher than value MCs for the same value with low value frequency.

The pattern graphs the system ranks share the same signature. Thus, they are r-radius graphs with the same r. In almost all the cases they have the same number of edges and they differ only in the relationship edges which are not part of any MC. For this reason, the pattern graphs are ranked in descending order of their connecting edge frequency defined next. Given a pattern P, its *connecting edge frequency* $f_c(P)$ is the sum of the number of occurrences n_e in the data graph of the relationship edges e in P that do not occur in an MC in P divided by the total number $|E_R|$ of relationship edges in the data graph. That is, if E_c is the set of these relationship edges in P, $f_c(P) = \sum_{e \in E_c} n_e/|E_R|$.

In order to rank MCs and pattern graphs our system needs statistics about value MCs and their property edges and about connecting relationship edges in pattern graphs. This information is precomputed and stored with the structural summary when this one is constructed. Therefore, no access to the data graph is needed.

5. Relaxing Pattern Graphs

A pattern graph selected by the user might return no results, or if it does, it might miss some interesting results the user would like to see. In order to expand the result set of this pattern graph chosen by the user and get additional results for the same query signature that involve the same classes, relationships, properties and values but additional entities, we relax this pattern graph. In this section, we first define relaxed pattern graphs. We then introduce an operation on pattern graphs, called vertex split operation, and we show that a pattern graph can be relaxed by applying vertex split operations. Pattern graphs which are less relaxed are preferable over pattern graphs which are more relaxed since, they are closer to the original pattern graph selected by the user. Therefore, we introduce different metrics to characterize the degree of relaxation of a relaxed pattern graph.

5.1. *Relaxed Pattern Graphs*

In order to define relaxed patterns, we need the concept of homomorphism between pattern graphs.

Definition 9 (Pattern Graph Homomorphism) Let P_1 and P_2 be two pattern graphs. A *homomorphism* from P_1 to P_2 is a function h from the variable vertices (entity variable and value variable vertices) of P_1 to the variable vertices of P_2 such that, if X is an entity variable vertex in P_1:

(a) for any type edge (X, c) in P_1, there is a type edge $(h(X), c)$ in P_2. That is, X in P_1 and $h(X)$ in P_2 are of the same type c.

(b) for every relationship edge (X, Y) in P_1 labeled by r, where Y is another entity variable in P_1, there is a relationship edge $(h(X), h(Y))$ in P_2 labeled by the same label r.

(c) for every property edge (X, Y) in P_1 labeled by p, where Y is a value variable vertex, there is a property edge $(h(X), h(Y))$ in P_2 labeled by the same label p.

(d) for every property edge (X, v) in P_1 labeled by p, where v is a value vertex labeled by the value (keyword) V, there is a property edge $(h(X), v')$ in P_2 labeled by the same label p, where v' is a value vertex also labeled by V.

Figure 10 shows four pattern graphs P_1, P_2, P_3 and P_4 and a homomorphism from the pattern graph P_2 to the pattern graph P_1. The vertex mapping is illustrated with dashed arrows. One can see that there are also homomorphisms from the pattern graphs P_3 and P_4

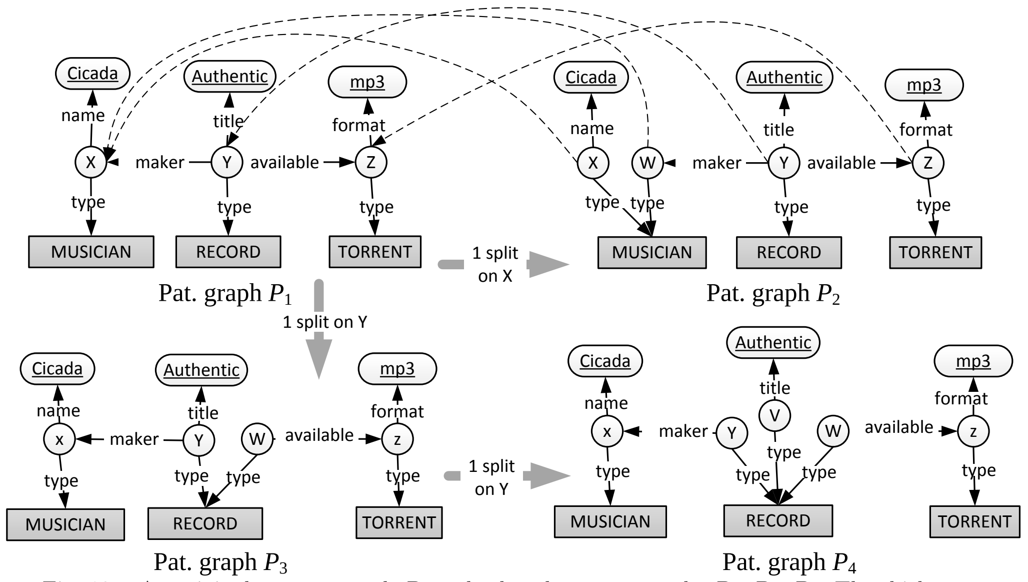


Fig. 10. An original pattern graph P_1 and relaxed pattern graphs P_2, P_3, P_4. The thick arrows indicate vertex split operations and their labels indicate the vertex on which the operation is applied.

to the graph pattern P_1. However, there is no homomorphism from pattern graph P_1 to any one of the other pattern graphs.

We use the concept of homomorphism to define a relation on pattern graphs.

Definition 10 (Relation $\prec$) Let P_1 and P_2 be two pattern graphs. We say that P_2 is a *relaxation* of P_1 or that P_2 is a *relaxed version* of P_1 if there is a homomorphism from P_2 to P_1 but there is no homomorphism from P_1 to P_2. In this case, we write $P_1 \prec P_2$.

In the example of Figure 10, $P_1 \prec P_2$ and $P_1 \prec P_3 \prec P_4$. No other $\prec$ relationship holds between these patterns.

Clearly, relation $\prec$ is a strict partial order on the set of pattern graphs (it is irreflexive, asymmetric and transitive). We call its minimal elements *original* pattern graphs. In an original pattern graph every class vertex is connected through a type edge exactly to one entity variable vertex. The patterns initially presented to the user are original pattern graphs and one of them is selected and possibly relaxed. If an (original) pattern graph P has an embedding to an RDF graph, a relaxed version of P also has an embedding to the same RDF graph. The opposite is not necessarily true. Therefore with relaxed pattern graphs we can expand the result set of an original pattern graph.

5.2. *Vertex Splitting*

A pattern graph is relaxed by applying the *vertex split* operation to one or more of its entity variable vertices. The split operation "splits" an entity variable vertex in a pattern graph into two entity variable vertices of the same type and partitions the incident edges of the original entity variable vertex between the two new vertices as indicated by the operation.

Definition 11 (Vertex split operation) Let P be a pattern graph, v be an entity variable vertex in P connected with a type edge to a class vertex c, and $E = \{e_1, \ldots, e_k\},\ k \geq 1$, be a proper subset of the set of non-type edges incident to v in P. Assume the edges $e_1, \ldots, e_k$, are connecting the pairs of vertices $(v, v_1), \ldots, (v, v_k)$, respectively. The *vertex split* operation *split*(P, v, E) returns a pattern graph constructed from P as follows:

(a) Add to P a new entity variable vertex v' of type c.

(b) Remove all the non-type edges (incident to v) that occur in E.

(c) Add k edges $(v', v_1), \ldots, (v', v_k)$ having the same labels as the edges $e_1, \ldots, e_k$, respectively.

Splitting one or more of the vertices of an original pattern graph P results in a relaxed pattern graph (a relaxed version of P). Applying the split operation in sequence can create a pattern graph where the non-type edges incident to v are partitioned into more than two sets attached to different vertices, as desired.

Not all the entity variable vertices are interesting for splitting. This operation is defined only on candidate split vertices. An entity variable vertex is a *candidate split* vertex if it has at least two non-type edges.

As an example, consider the original pattern graph P_1 of Figure 10. This is a pattern graph for the keyword query {`Cicada, Authentic, Girl`}. Applying *split*$(P_1, X, \{maker\})$ to P_1 results in the pattern graph P_2. Applying *split*$(P_1, Y, \{track\})$ to P_1 results in the pattern graph P_3. Applying, in turn, *split*$(P_3, Y, \{title\})$ to P_3 produces the pattern graph P_4.

Since any partitioning of the edges incident to a vertex in an original pattern graph can be obtained in a relaxed pattern graph by a successive application of vertex split operation, the following proposition can be shown.

Proposition 1 *Let P_1 and P_2 be two pattern graphs. Then, $P_1 \prec P_2$ iff P_2 can be produced from P_1 by applying a sequence of vertex split operations.*

Proof: *If part*: If P_2 can be produced from P_1 by applying a vertex split operation then, as stated in Definition 11, the non-type edges of an entity variable vertex X in P_1 are partitioned into two sets of edges which are incident to vertex X and its split image in P_2. Then clearly, there is a homomorphism from P_2 to P_1. If a sequence of vertex split operations is applied then this homomorphism exists by transitivity. That is, $P_1 \prec P_2$.

Only-if part: Since P_1 and P_2 are both pattern graphs there is a homomorphism from P_2 to P_1 only if for any entity variable vertex X in P_1 there are $X_1, \ldots, X_k,\ k \geq 1$, vertices of the same type as X in P_2 and the non-type edges of X are partitioned among these vertices such that each one of $X_1, \ldots, X_k$ has at least one non-type edge (Definition 9). Then clearly, P_2 can be obtained from P_1 by applying in sequence $k - 1$ vertex split operations for every vertex in P_1 where $k > 1$. $\square$

The above proposition shows that the vertex split operation is sound and complete w.r.t. relaxed pattern graphs.

5.3. *Measuring Pattern Graph Relaxation*

Usually we want to relax a pattern graph so that it is as close to the initial pattern graph as possible. To this end, we introduce three metrics of decreasing importance to measure the degree of relaxation of a pattern graph. All these three metrics depend on the vertex split operations applied to the original pattern graph. The first one is called keyword connectivity of the pattern graph. In order to define the keyword connectivity of a pattern graph we use the concept of tightly connected pair of keyword instances.

Two keyword instances in a pattern graph P are *tightly connected* if there exists a simple path between them which does not go through a class vertex. For instance, in the pattern graph of Figure 11(b), the keyword instances `Rebirth` and `mp3` are tightly connected whereas the keyword instances `Cicada` and `Rebirth` are not.

Definition 12 (Pattern graph Keyword connectivity) The *keyword connectivity* of a pattern graph is the number of unordered keyword instance pairs that are strongly connected divided by the total number of unordered keyword instance pairs.

In an original pattern graph, all pairs of keyword instances are strongly connected. Therefore, its keyword connectivity is 1. Relaxing such a pattern graph by applying the vertex split operation to any entity variable vertex produces a pattern graph of lower or the same keyword connectivity. Relaxing an acyclic original pattern graph by applying vertex splitting to any entity variable vertex always reduces the keyword connectivity of the original pattern graph. For instance, the keyword connectivity of the pattern graph in Figure 11(a) is 1. The keyword connectivity of the relaxed pattern graphs of Figures 11(b) and (d) is 0.4 and the keyword connectivity of those of Figures 11(c), (e) and (f) is 0.3.

In order to distinguish between relaxed pattern graphs of the same pattern graph which have the same keyword connectivity, we introduce another metric called "dispersion" of the keyword instances of a pattern graph. Roughly speaking, this metric is used to capture how much the keywords are dispersed as a result of vertex split operations in the pattern graph. To formally define the keyword instance dispersion metric we introduce the concept of "split distance". The *split distance* of two keyword instances in a pattern graph P is the minimum number of class vertices in the simple paths between these two keyword instances in P excluding the terminal vertices. The term "split distance" is explained by the fact that a class vertex is introduced in a simple path between two keyword instances only as a result of the application of a split operation. For instance, in the pattern graph of Figure 11(c), the split distance of the keyword instances of `Gimma` and `mp3` is 1 and that of `Gimma` and `Rebirth` is 2. The more split operations we apply to the vertices on a path between two keyword instances, the more syntactically dispersed these keyword instances become in the pattern

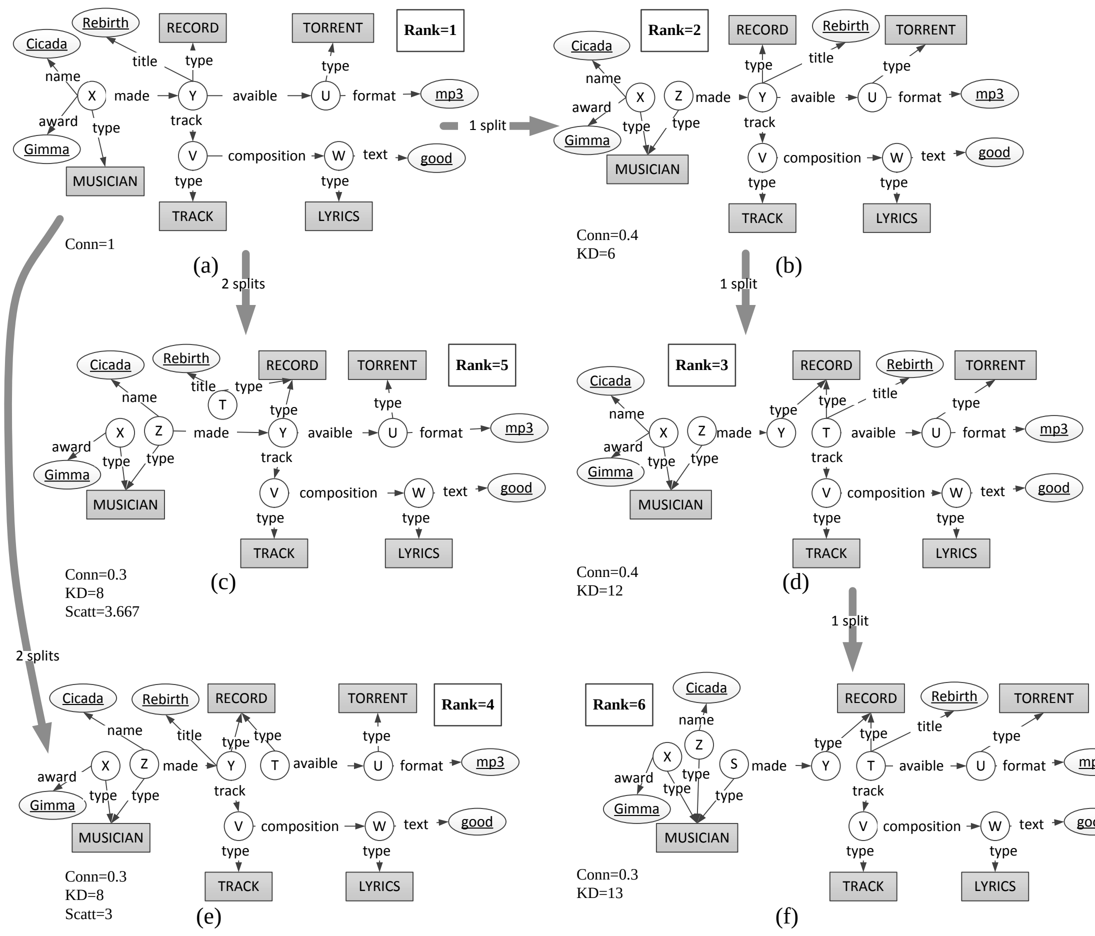


Fig. 11. The graphs show an original pattern graph (a) and its relaxed pattern graphs (b), (c), (d), (e) and (f).

graph, reflecting a weaker semantic connection between these keywords.

Definition 13 (Pattern graph keyword dispersion) The *keyword dispersion* of a pattern graph P is the sum of the split distances of all unordered pairs of keyword instances in P.

A relaxed pattern graph with smaller keyword dispersion is preferred over a pattern graph of the same keyword connectivity but higher keyword dispersion since its keywords are assumed to be more closely related. For example, the pattern graphs of Figures 11(b) and (d) have the same keyword connectivity of 0.4 whereas their keyword dispersion is 6 and 12, respectively. Hence, the pattern graph of Figure 11(b) will be ranked higher than that of Figure 11(d). Similarly, the keyword connectivity of the pattern graphs of Figures 11(c), (e) and (f) is 0.3. However the keyword dispersion of the pattern graphs of Figures 11(c) and (e) is 8 and that of Figure 11(f) is 13. Hence, the pattern graphs of Figures 11(c) and (e) will be ranked higher than that of Figure 11(f).

Nevertheless, it is possible that multiple relaxed patterns of the same original pattern graph have not only the same keyword connectivity but also the same keyword dispersion. In order to differentiate between the degree of relaxation of such pattern graphs, we employ a third metric called *scatteredness* of a pattern graph. We first define the *distance* between two tightly connected keyword instances in a pattern graph as the number of vertices in a shortest path between them. The distance between a keyword instance which is the label of a value vertex and a keyword instance which is the label of a property edge incident to this vertex is 0. In the pattern graph of Figure 11(c) the distance between the tightly connected keyword instances of `Cicada` and `mp3` is 3 while the distance between the tightly connected keyword instances of `mp3` and `good` is 4.

A relaxed pattern graph partitions its keyword instances into sets of tightly connected keyword instances such that any two keyword instances which are tightly connected belong to the same set. The scatteredness of a pattern graph measures how sparsely are positioned the keyword instances within the sets of the partition.

Definition 14 (Scatteredness of a pattern graph) Let N be the sum of the distances between all the unordered keyword instance pairs that are tightly connected, and S be the total number of tightly connected unordered keyword pairs in a pattern graph P. The *scatteredness* of the tightly connected keyword instances of P (scatteredness of P for short) is N/S.

In the example of Figure 11, the pattern graphs (c) and (e) have the same keyword connectivity of 0.3 and the same keyword dispersion of 8. However, the scatteredness of the pattern graph of Figure 11(c) is 3.67 and that of the pattern graph of Figure 11(e) is 3. We use the pattern graph scatteredness to rank the relaxed pattern graphs of a pattern graph having the same keyword connectivity and keyword dispersion. In our running example, the pattern graph of Figure 11(e) is ranked before the pattern graph of Figure 11(c), since the tightly connected keyword instances of the latter pattern graph are more sparsely positioned than that of the tightly connected keyword instances of the former pattern graph.

As another example, consider the pattern graph of Figure 2(a) and also shown in Figure 12(a) and two relaxations of it shown in Figures 12(b) and (c). The relaxed pattern graphs of Figures 12(b) and (c) have the same keyword connectivity of and keyword dispersion but the scatteredness of graph Figure 12(b) is 2 and that of Figure 12(c) is 3. Therefore, the graph of Figure 12(b) should precede that of Figure 12(c) in a ranking.

5.4. *Relaxation Order*

Given two pattern graphs P_1 and P_2, we say that, P_2 is "equally relaxed as" or "more relaxed than" P_1, and we write $P_1 \leq_r P_2$, if: (a) connectivity(P_1) $\geq$ connectivity(P_2), or (b) connectivity(P_1) $=$ connectivity(P_2) and dispersion(P_1) $\leq$ dispersion(P_2), or (c) connectivity(P_1) $=$ connectivity(P_2) and dispersion(P_1) $=$ dispersion(P_2) and scatteredness(P_1) $\leq$ scatteredness(P_2). Clearly, $\leq_r$ is reflexive and transitive and any two pattern graphs are comparable w.r.t. $\leq_r$. If a set of pattern graphs is ranked with respect to $\leq_r$, with the less relaxed pattern graphs ranked first, we say that it is ranked in *relaxation*

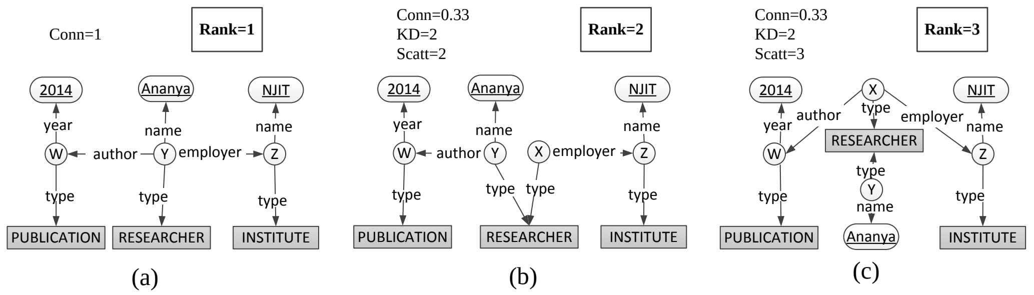


Fig. 12. (a) Original pattern graph (b) and (c) relaxed pattern graphs

order.

Since split operations introduce additional type edges in the pattern graph it is not difficult to see that the following statement holds.

Proposition 2 *Given two pattern graphs P_1 and P_2, if $P_1 \prec P_2$ then $P_1 \leq_r P_2$.*

Proof: By Proposition 1, if $P_1 \prec P_2$, there is a sequence of split operations which produce P_2 from P_1. Let now, P' be a pattern graph produced by applying split operation s to another pattern P. Since, s cannot not increase the number of tightly connected keyword instance pairs in P, connectivity(P') $\leq$ connectivity(P). Similarly, since it cannot reduce the split distance of any pair of keyword instance in P, dispersion(P) $\leq$ dispersion(P'). Finally, if s does not change the keyword connectivity and keyword dispersion of P, P can only be a cyclic pattern graph, and P' is produced by applying a vertex split operation to an entity variable vertex which lies on a cycle in P. Hence, the sets of tightly connected keywords instances of P are not affected by s. Further, the distance between two tightly connected keyword instances within a set in P can only increase or remain the same when s is applied. Therefore, scatteredness(P) $\leq$ scatteredness(P'). Consequently, $P \leq_r P'$. Since this is true for all the split operations in the sequence that produced P_2 from P_1, and $\leq_r$ is transitive, $P_1 \leq_r P_2$. $\square$

Proposition 2 states that $P_1 \prec P_2$ is compatible with $P_1 \leq_r P_2$. If P_2 is produced by applying a vertex split operation to P_1, $P_1 \leq_r P_2$.

6. Computing Relaxed Pattern Graphs

In this section, we elaborate on the reasons for a pattern graph having an empty answer. Then, we design an algorithm which computes relaxed pattern graphs with non-empty answers ranked in ascending order of their degree of relaxation. Finally, we show how view materialization and multiquery optimization techniques can be exploited to support the computation of relaxed pattern graphs and their evaluation on the RDF graph.

6.1. *Identifying Empty Vertices for Relaxation*

If an original pattern graph for a query has an empty answer on an RDF graph, we would like to identify vertices in the pattern graph which if not split, the relaxed pattern graph will keep producing an empty answer. Splitting these vertices does not guarantee that the relaxed query does have a non-empty answer. However if we omit splitting any one of these vertices, the relaxed pattern graph will not return any results. We call these vertices *empty vertices*.

Definition 15 (Empty vertex) An entity variable vertex X in a pattern graph P on a data graph G is an *empty vertex* iff P or any relaxed version of P where X is not split has an empty answer on G.

The following proposition characterizes empty vertices in a pattern graph. Let X be an entity variable vertex of type c in a pattern graph P, $p'_1(X, Z'_1), \ldots, p'_m(X, Z'_m)$ be the property edges incident to X whose value vertices $Z'_1, \ldots, Z'_m$ are variables, $p_1(X, v_1), \ldots, p_n(X, v_n)$ be the property edges incident to X whose value vertices $v_1, \ldots, v_n$ are not variables (they are keyword instances), $r_1(X, Y_1), \ldots, r_k(X, Y_k)$ be the relationship edges from X to some other entity variable vertices $Y_1, \ldots, Y_k$ of type $c_1 \ldots c_k$, respectively, and $r'_1(X, Y'_1), \ldots, r'_l(X, Y'_l)$ be the relationship edges to X from some other entity variable vertices $Y'_1, \ldots, Y'_l$ of type $c'_1, \ldots, c'_l$, respectively (see Figure 13). We call the graph of Figure 13 the *star-join view* of the entity variable vertex X in P.

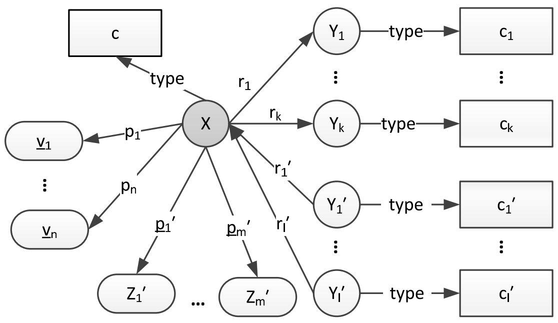


Fig. 13. Star-join view of entity variable vertex X

Proposition 3 *An entity variable vertex X is an empty vertex of pattern graph P on an RDF graph G iff the star-join view for X in P has an empty answer on G.*

Proof:*If part*: If the star-join view V_X of X has an empty answer, then P or any relaxed version of P where X is not split has an empty answer since V_X is a subgraph of this graph. Therefore, X is an empty vertex.

Only-if part: Let's assume that X is empty and the star-join view of X is non-empty. We will show that this is a contradiction. Since X is empty, the pattern graph P or any relaxed version of it where X is not split do not have an answer on G. Let P' be a pattern graph obtained from P by splitting all entity variable vertices except X until no more split operations can be applied. Since the star-view join of X is non-empty, P' has an answer on G. This is a contradiction since we assumed that X is empty. □

All empty vertices need to be split when relaxing a query in order to possibly get a nonempty answer for the query.

6.2. *An Algorithm for Computing Relaxed Patterns*

We provide now an algorithm which, given the pattern graph P chosen by the user (for instance, by navigating through the clustering hierarchy discussed in Section 4), gradually generates relaxed pattern graphs of P having non-empty answers. The algorithm returns these pattern graphs and their answers in ascending relaxation order. The number of relaxed pattern graphs returned is controlled by the user.

We provide now the intuition behind the algorithm. The chosen pattern graph might have an empty answer on the RDF data graph. For example, for the keyword query $Q = \{$`Gimma, Cicada, Rebirth, mp3, good`$\}$, the user chosen pattern graph shown in Figure 11(a) does not have a match on the RDF data graph of Figure 4. Hence, it needs to be relaxed. One can see that this pattern graph has an empty entity variable vertex (vertex X) since the star join view of this vertex is empty (Proposition 3). All the empty vertices of a pattern graph need to be split in order for the resulting pattern graph to have a non-empty answer (Definition 15). Therefore, vertex X needs to be split. Nevertheless, splitting all the empty vertices of a pattern graph does not guarantee that the resulting pattern graph will not have empty vertices. For instance, the pattern graphs 11(b) and (d) which are obtained from the pattern graph 11(a) by splitting its only empty vertex X, still have an empty vertex. Further, even if a relaxed pattern graph does not have empty vertices, it might still have an empty answer. In our running example of Figure 11 the relaxed pattern graphs of Figures 11(c) and (e) do not have empty vertices but have an empty answer on the RDF graph of Figure 4. In both the above cases, additional split operations need to be applied to candidate split vertices in order to reach a pattern graph with non-empty answer. On the other hand, splitting the empty vertices of a graph might result on a pattern graph which has a non-empty answer and this is the case of the pattern graph of Figure 11(f) which is obtained by applying a split operation to the only empty vertex X of the pattern graph of Figure 11(d). Since we want to return to the user relaxed pattern graphs with higher rank (w.r.t. $\leq_r$) first, we chose for relaxation a pattern graph with the highest rank at every iteration of the algorithm. For the same reason, if the pattern graph chosen for relaxation does not have empty nodes, we apply a split operation (in all possible ways) to all candidate split vertices separately. By choosing a pattern graph with the highest rank for relaxation at every iteration of the algorithm, we also avoid the redundant generation of relaxed pattern graphs.

6.2.1. *Algorithm Description*

The outline of our algorithm is shown in Algorithm 1. The input of this algorithm is an original pattern graph P. The algorithm generates as output a list of relaxed pattern graphs and their corresponding result graphs on the data graph in increasing order of relaxation. The data structure R is a list used to store pattern graphs (both original and relaxed). The variable $MoreResults$ reflects the user's choice of fetching more answers by further relaxing the pattern graphs in R. The algorithm first chooses a pattern graph P_{Top} with the highest rank from R (line 5). The pattern graph P_{Top} is then checked for empty vertices (line 7). If P_{Top} has non-empty vertices, they are marked (line 8) and they (and their split images) remain marked in the relaxations of P_{Top}. If EV, the set of empty vertices in P_{Top} is non-empty, the function $GetRelaxedFromEmptyVertices(P_{Top}, EV)$ is called (line 10). This function relaxes P_{Top} by applying one vertex split operation to all of its empty vertices in all possible

Algorithm 1

Input: P: An original pattern graph.

Output: A list of relaxed pattern graphs of P with non-empty answers in ascending relaxation order. Every pattern graph is returned along with its answer.

```
1: $R = \{P\}$;
2: $MoreResults = True$;
3: $Ans = \emptyset$;
4: while $R \neq \emptyset$ and $MoreResults$ do
5:     $P_{Top} \leftarrow$ the pattern graph in $R$ with the highest rank;
6:     $R \leftarrow R - \{P_{Top}\}$;
7:     $EV \leftarrow ComputeEmptyVertices(P_{Top})$;
8:     Mark the new non-empty vertices in $P_{Top}$;
9:     if $EV \neq \emptyset$ then
10:        $NewR \leftarrow GetRelaxedFromEmptyVertices(P_{Top}, EV)$;
11:        Rank the pattern graphs in $NewR$ in ascending relaxation order;
12:        $R \leftarrow$ merge $R$ and $NewR$ into one list of patterns ranked in ascending relaxation
               order;
13:    else
14:        $Ans \leftarrow Evaluate(P_{Top})$;
15:        if $Ans \neq \emptyset$ then
16:            Output $(P_{Top}, Ans)$;
17:            $MoreResults \leftarrow$ input from the user on whether more results are needed;
18:        if $Ans = \emptyset$ or $MoreResults$ then
19:            $MoreR \leftarrow GetRelaxed(P_{Top})$;
20:            Rank the pattern graphs in $R$ in ascending relaxation order;
21:            $R \leftarrow$ merge $R$ and $MoreR$ into one list of patterns ranked in ascending
                   relaxation order;
22: function GetRelaxed($P$)
23:     $R = \emptyset$
24:     for every candidate split vertex $X$ in $P$ do
25:         $R_X =$ {pattern graphs obtained by applying one vertex split operation to $X$ in all
                possible ways}
26:         $R = R \cup R_X$
27:     return $R$
28: function GetRelaxedFromEmptyVertices($P, EV$)
29:     $R = \{P\}$
30:     for every vertex $X$ in $EV$ do
31:         $R_X = \emptyset$
32:         for every pattern graph $P'$ in $R$ do
33:             $R_X = R_X \cup$ {pattern graphs obtained by applying one vertex split operation
                    to $X$ in $P'$ in all possible ways}
34:             $R = R - P'$
35:         $R = R_X$
36:     return $R$
```

ways (lines 30-35). The resulting relaxed pattern graphs form a new list $NewR$ of relaxed pattern graphs, which is then ranked in ascending relaxation order and is merged with the list R (lines 11-12). Otherwise, if P_{Top} does not have empty vertices, it is evaluated over the data graph and if the set Ans of result graphs is non-empty, P_{Top} is returned to the user along with Ans (lines 14-16). In case the user wants more results, or the pattern graph P_{Top} produces an empty answer when evaluated over the data graph, the function $GetRelaxed(P_{Top})$ is evoked (lines 18-19). This function relaxes P_{Top} by applying one vertex split operation to all of its candidate split vertices in all possible ways (lines 24-26). The list of relaxed pattern graphs returned by $GetRelaxed(P_{Top})$ is stored in a list $MoreR$. The relaxed pattern graphs in $MoreR$ are then ranked and merged with the list R of pattern graphs (lines 20-21). The whole process, as described in lines 5-21, continues until the user is satisfied with the results or no more pattern graphs are left in R. The above discussion suggests the next proposition.

Proposition 4 *Algorithm 1 correctly computes in relaxation order the relaxed pattern graphs with non-empty answers of the pattern graph given as input.*

Note that during the execution of the algorithm, the user can provide input on how to split empty or non-empty vertices when a pattern graph comes up for relaxation either because it has empty vertices or because it does not have empty vertices but has an empty answer. In this case, the number of split operations applied in this iteration of the algorithm is reduced since only the alternative dictated by the user is applied to the relevant vertex. We have omitted this feature in the outline of the algorithm, showing only the fully automated version, for simplicity of presentation.

The execution cost of our pattern graph relaxation algorithm depends on: (a) the cost for determining the empty vertices (by evaluating star-join views over the data graph), (b) the evaluation cost of relaxed pattern graphs over the data graph, and (c) the cost for generating relaxed pattern graphs using the functions $GetRelaxed(P)$ and $GetRelaxedFromEmptyVertices(P, EV)$. The star-join views can be computed efficiently by exploiting the indexes defined on entity attributes of the relations for properties and relationships. For the efficient evaluation of the relaxed pattern graphs we devise and discuss in the next section evaluation plans that exploit answering queries using materialized views and multiquery optimization techniques. Functions $GetRelaxed(P)$ and $GetRelaxedFromEmptyVertices(P, EV)$ can produce up to $(C_1^n + \ldots + C_{n-1}^n)/2 = 2^{n-1} - 1$ relaxed pattern graphs by applying vertex split operations on one vertex X, where n is the number of keywords. The worst case scenario can happen when each one of the n keyword instances in the pattern graph is linked to X through a different non-overlapping path. Since every pattern graph is a Steiner graph, it can have up to $nr + 1$ entity variable vertices, where r is the radius of the Steiner graph. In the worst case, all of them are need to be split. Nevertheless, even though in the worst case scenario an exponentially large number of relaxed pattern graphs can be produced, in practice only few of them are produced. Further, only a tiny portion of those produced are evaluated for empty vertices and empty results since otherwise the produced relaxed pattern graphs would be very irrelevant to the original pattern graph and not of interest to the user. This intuition is also confirmed in our experimental results.

6.3. *Optimization Techniques to Support Query Relaxation and Evaluation*

Materializing views in the main or secondary memory is a well-known technique for improving the performance of queries. This technique has been studied extensively over the years for queries on relational databases [22, 23], but the contributions for queries over RDF databases are limited [24, 25]. Queries to be evaluated are rewritten (inclusively or exclusively) using the stored views [26] in order to produce a query evaluation plan involving the materialized views which is more efficient than a plan involving exclusively the base relations. The technique is useful both in a horizontal and in a vertical setting. In a vertical setting (query caching) queries and subqueries are cached on the assumption that they will be useful for evaluating subsequent queries. A new query to be evaluated is rewritten equivalently using previously cached views. The expectation is that the produced evaluation plan will be cheaper and the savings will amortize the cost for deciding what subqueries to cash and for finding a rewriting of the query using the materialized views. In a horizontal setting (multiquery optimization) multiple queries need to be evaluated concurrently. To this end, common subexpressions among the queries in a given workload are detected on the fly and a global evaluation plan for all the queries is derived, which might be more efficient to evaluate than evaluating each query in the workload separately. A global evaluation plan reflects rewritings of the given queries over the views (common subexpressions) which remain materialized until all the queries in the workload that use them are evaluated. The expectation is that the savings produced from the global evaluation plan will amortize the cost for detecting the common subexpressions and producing the alternative global evaluation plans. The multiquery optimization problem is a complex one. Not surprisingly, it has been shown to be NP-hard even for conjunctive relational queries [27]. There are different sources of complexity to these problems in the general relational context: (a) deciding whether a query Q can be answered using a set of materialized views and producing an equivalent rewriting of Q using the views, (b) detecting common subexpressions among queries in a query set, and (c) deciding which views/common subexpressions to materialize in order to produce an efficient evaluation plan.

Our pattern graph relaxation algorithm generates multiple queries to be evaluated sequentially or concurrently. We discuss in this section how the techniques discussed above can be leveraged to design a global evaluation plan for all the queries that need to be computed. The goal is to exploit extensively common subexpressions among the generated queries. We consider a relational setting where the base relations are property and relationship relations. Initially, the value matching construct views of a given original pattern graph are evaluated and cached. Subsequently the star-join views are evaluated using the value matching constructs cached. Some of the generated queries are to be evaluated sequentially (when the pattern graphs with the highest rank is chosen for evaluation) while others are to be evaluated concurrently (when value matching construct views or star-join views are evaluated). Fortunately, these queries are not random queries but subgraphs of the original pattern graph or of its relaxations. As a consequence, common subexpressions among different queries can be detected easily based on the overlapping of the corresponding graphs because they are subgraphs of these graphs. Query rewritings can also be produced easily by simply joining the materialized subqueries (graphs) on their common entity variable vertices. Finally, common subexpressions among queries are selected so as to maximize the number of common entity variable vertices.

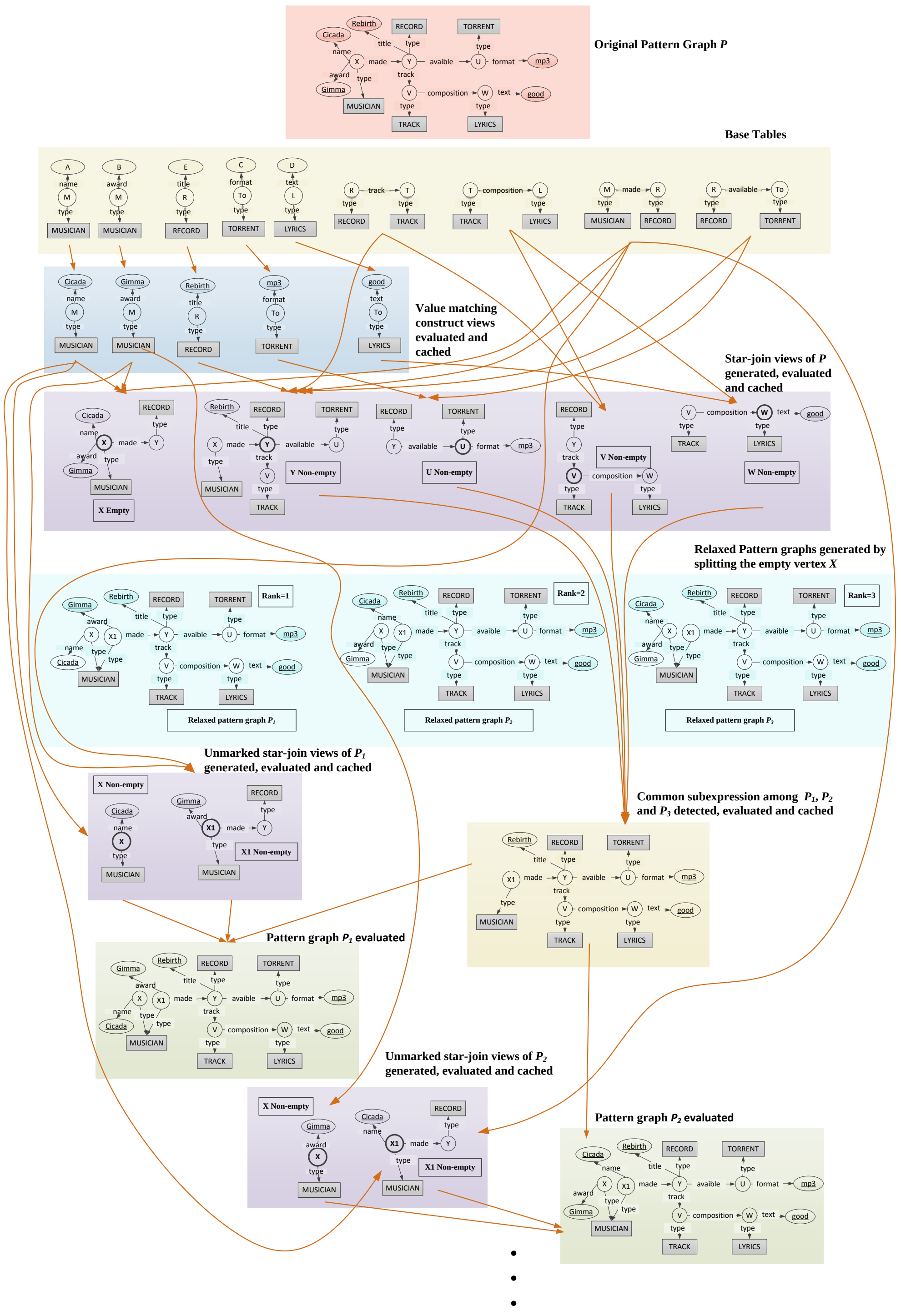

Original Pattern Graph P
Base Tables
Value matching construct views evaluated and cached
Star-join views of P generated, evaluated and cached
X Empty
Y Non-empty
U Non-empty
V Non-empty
W Non-empty
Relaxed Pattern graphs generated by splitting the empty vertex X
Rank=1
Rank=2
Rank=3
Relaxed pattern graph P_1
Relaxed pattern graph P_2
Relaxed pattern graph P_3
Unmarked star-join views of P_1 generated, evaluated and cached
X Non-empty
X1 Non-empty
Common subexpression among P_1, P_2 and P_3 detected, evaluated and cached
Pattern graph P_1 evaluated
Unmarked star-join views of P_2 generated, evaluated and cached
Pattern graph P_2 evaluated

Fig. 14. A global evaluation plan for relaxed queries

Figure 14 shows an example of caching and utilization of different subqueries for the successive evaluation of relaxed pattern graphs. The flow, from top to bottom, follows the execution of our algorithm. On the top of the figure, the input original query is shown. Next follow the based tables needed to compute relaxations of this query. The global evaluation plan involves computing and caching the value matching constructs of the original query and its star-join views which are shown in the next two layers. The fifth layer displays a ranked list of relaxed pattern graphs produced by splitting the empty vertices of the original pattern graph. The first relaxed pattern graph is considered and checked for empty vertices by evaluating the star-join views of the unmarked entity variable vertices which are also cached. As no empty vertex is found, this relaxed pattern graph is evaluated. Its evaluation involves the computation of the maximal common subexpression of the relaxed pattern graphs and the cached star-join views of the entity variable vertices. The process continues with the next relaxed pattern graph.

7. Experimental Evaluation

We implemented our approach and run experiments to evaluate our system. The goal of our experiments is to assess: (a) the effectiveness of the metrics introduced in ranking the relaxed pattern graphs, and (b) the feasibility of our system in producing and presenting to the user the relaxed pattern graphs and their answers in real time.

7.1. *Datasets and queries*

We used two real datasets Jamendo[b] and YAGO 1.0.0[c][28]. Jamendo is a large repository of Creative Commons licensed music. It consists of 1.1M triples and its size is 85MB. It contains information about musicians, records, music tracks and their licenses, music categories, track lyrics and many other details related to music production. This dataset has nearly 300,000 entities belonging to 12 classes. Jamendo has 14 properties and 10 relationships. Much larger, YAGO is an open domain dataset combining information about resources from different aspects of life extracted from Wordnet[d] and Wikipedia[e]. YAGO contains nearly 20 million triples about approximately 2 million entities belonging to over 180,000 classes. The entities in the YAGO dataset are characterized by 32 properties. The entities are associated to each other with 58 relationships.

The structural summary of each dataset was stored in a relational database which contained tables for classes, properties and relationships. The database also stored in a table the set of values associated with each property of the dataset. The experiments are conducted on a standalone machine with an Intel i7-5600U@2.60GHz processors and 8GB memory.

For the experiments, we employed three users who were computer science students able to understand RDF graph notation and were not involved in the research for this paper. For each of the two datasets we select queries which have no results or very few results. Users were provided with different queries on those two datasets and in every instance they selected the most relevant pattern graph among those provided by the system. We report on 20 queries

[b] http://dbtune.org/jamendo/
[c] http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/
[d] https://wordnet.princeton.edu/
[e] https://www.wikipedia.org/

(10 queries for each dataset). The queries cover a broad range of cases. They involve from 3 to 7 keywords, while the selected relevant pattern graphs form a star or a chain or a combination of them and in the case of YAGO dataset, they also form a cycle. Table 1 shows the keyword queries and information about their relevant pattern graph on both datasets.

Table 1. The keyword Queries in the two datasets

Query #	**Keywords**	**Structure of PG**	**Empty Vertex**	**Empty Result**
	Jamendo Dataset			
J1	teenage, text, fantasie, Document	star-chain	N	Y
J2	signal, onTimeLine, 10002, recorded_as, sweet	chain	N	Y
J3	kouki, recorded_as, knees	star-chain	N	Y
J4	briareus, reflectin, cool, girl	star	N	Y
J5	kouki, revolution, electro, good	star	N	Y
J6	nuts, spy4, chillout, track	star	Y	Y
J7	biography, guitarist, track, lemonade	chain	N	N
J8	divergence, track, obssession, format, mp32	star-chain	N	N
J9	fantasie, performance, recorded_as, slipstream	chain	N	Y
J10	signal, recorded_as, fantasie, onTimeLine, 10001	chain	N	Y
	YAGO Dataset			
Y1	sonai, influences, poet, 1414, born, discovers, whirling	chain	Y	Y
Y2	dunderberg, interested, nyc, industrialist, influences, victor	star	Y	Y
Y3	richard, louis, pulitzer, award, american, book	star-chain	Y	Y
Y4	delhi, actor, shahrukh, acted, produced, india, films	cyclic	Y	Y
Y5	ridley, directed, gladiator, douglas, prize	cyclic	Y	Y
Y6	married, actor, wrestler, produced, directed, movie, tripper	cyclic	N	Y
Y7	aristotle, influences, heliocentrism, astronomer, cambridge	star-chain	N	Y
Y8	yoko, artist, grammy, huckleberry	star	Y	Y
Y9	neal, world, interface, cover, jensen	star-chain	Y	Y
Y10	grammy, sonny, produced, howard, created, westlife, songs	cyclic	Y	Y

7.2. *Effectiveness in ranking relaxed pattern graphs*

For our effectiveness experiments, we used three expert users to determine the ground truth. For each query, the system produced the candidate pattern graphs. A user selected among them the pattern graph which is most relevant to the query (refer to Section 4). This is the original pattern graph. We then run our pattern graph relaxation algorithm until the third relaxed pattern graph with a non-empty answer is produced and collected the relaxed pattern graphs generated (which are many more). The generated relaxed pattern graphs are ranked by our system in relaxation order as described in Section 5.3. In order to measure the effectiveness of our technique in generating a ranked list of relaxed pattern graphs, we provided the generated relaxed pattern graphs of each original pattern graph to the expert users who rank them based on their closeness to the original pattern graph. A ground truth ranking of the relaxed pattern graphs is produced from different user rankings based on the sum of the ranks of the relaxed patterns in each ranking. For the comparison, we are using two metrics: (a) *normalized Discounted Cumulative Gain* (nDCG) [29], and (b) *Kendall tau-b rank correlation coefficient* [30]. Both of them allow comparing two ranked lists of items. Note that the ground truth list and the one produced by our system might not form a strict total order. That is, there might be ties (relaxed pattern graphs with the same rank). We call a set of relaxed pattern graphs that have the same rank in a ranked list equivalence class of relaxed pattern graphs. Equivalence classes need to be taken into account in measuring the similarity of the ranked lists.

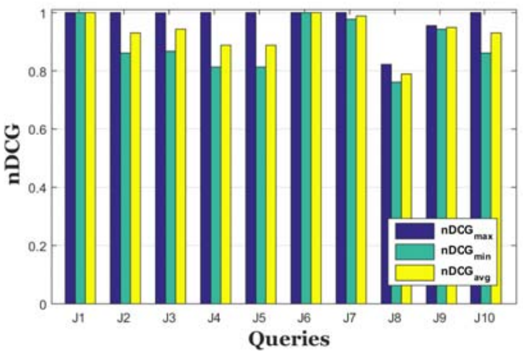
The nDCG metric was first introduced in [31] based on two key arguments: (a) highly important items are more valuable than marginally relevant items, and (b) the lower the position of the relevant item in the ranked list, the less valuable it is for the user because the less likely it is that the user will ever examine it. The first argument suggests that the relevance score of an item in the ranked list be used as a gained value measure. Then, the *cumulative gain* (CG) for position n in the ranked list is the sum of the relevance scores of the items in the ranked positions 1 to n. The second argument emphasizes that an item appearing at a lower position in the list should have a smaller share of its relevance score added to the cumulative gain. Hence, a discounting function is used over cumulative gain to measure *discounted cumulative gain* (DCG) for position n, which is defined as the sum of the relevance scores of all the items at positions 1 to n, each divided by the logarithm of its respective position in the ranked list. The DCG value of a ranked list is the DCG value at position n of the list where n is the size of the list. The *normalized discounted cumulative gain* (nDCG) is the result of normalizing DCG with the DCG of the list that is correctly ranked (the ground truth list produced by the expert user), by dividing the DCG value of the system's ranked list by the DCG value of the correct ranked list. Thus, nDCG favors a ranked list which is similar to the correct ranked list. The DCG at n is given by the following formula:

$$DCG_n = \sum_{i=1}^{n} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{1}$$

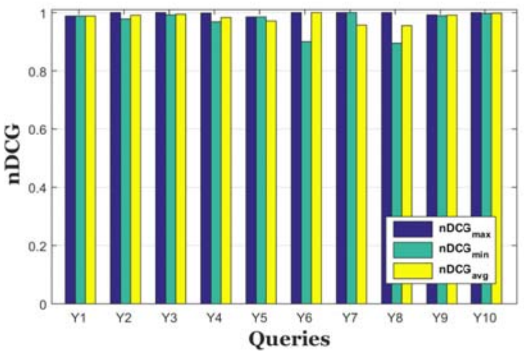
where rel_i, the relevance score of the item at position i in the ranked list, is the rank of this item's equivalence class in the inverse ground truth equivalence class list. For instance, if an item belongs to the 2nd equivalence class in a ground truth list of 5 equivalent classes, its relevance score is 3.

In order to take into account equivalent classes of pattern graphs in the system's ranked lists, we have extended nDCG by introducing minimum, maximum and average values for it. The nDCG$_{max}$ value of a ranked list RL_e with equivalence classes corresponds to the nDCG value of a strictly ranked (that is, without equivalence classes) list obtained from RL_e by ranking the pattern graphs in the every equivalence classes correctly (that is, in compliance with their ranking in the ground truth list). The nDCG$_{min}$ value of RL_e corresponds to the nDCG value of a strictly ranked list obtained from RL_e by ranking the pattern graphs in every equivalence classes in reverse correct order. The nDCG$_{avg}$ value of RL_e is the average nDCG value over all strictly ranked lists obtained from RL_e by ranking the pattern graphs in every equivalence classes in all possible ways. The nDCG values range between 0 and 1.

Figure 15 shows the nDCG$_{min}$, nDCG$_{max}$ and nDCG$_{avg}$ values for the queries of Table 1 on the Jamendo and YAGO datasets. As one can see, all the values are very close to 1 and the min and max values of $nDCG$ are close to each other. Specifically, the nDCG$_{avg}$ ranges between 1 and 0.789 in the Jamendo dataset and between 1 and 0.955 in the Yago dataset. The Yago dataset displays slightly better nDCG values. This is due to the fact that in the Yago dataset, the original pattern graphs have empty vertices in most cases while the opposite is true in the Jamendo dataset. The empty vertices guide the relaxation process narrowing the relaxation choices while in the absence of empty vertices, relaxed pattern graphs are produced by splitting all the candidate split vertices. When multiple vertices are split, the system creates larger equivalence classes and this negatively affects the nDCG values.



(a) Jamendo dataset



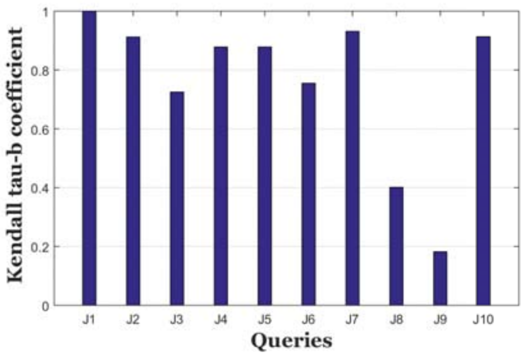
(b) YAGO dataset

Fig. 15. nDCG$_{max}$, nDCG$_{min}$ and nDCG$_{avg}$ for the queries on the Jamendo and YAGO datasets

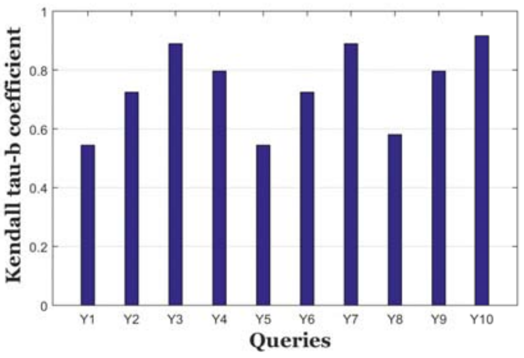
The Kendall tau rank correlation coefficient [32] was proposed to address the problem of measuring the association between two different rankings of the same set of items. For example, suppose that a set of items is given an order A which is correctly defined with reference to some quality q. An observer ranks the same set of items in an order B. A characteristic question that arises here is if the comparison of the orders B and A suggests that the observer possesses a reliable judgment of the quality q. In our context, we want to see if the comparison of the ranked list produced by our system (the relaxation order) with the correctly ranked list which is defined by the user suggests that the former possesses a reliable judgment of the closeness of the relaxed pattern graphs to the original pattern graph (which expresses the user's intention). However, the Kendall tau coefficient is useful when the ranked lists to be compared are strictly ranked. For this reason, we adopt here a variant called *Kendall tau-b coefficient* [30], which can deal with equivalent classes of items in the ranked lists. The Kendall tau-b coefficient is given by the following formula:

$$\tau_b = \frac{(\text{number of concordant pairs}) - (\text{number of discordant pairs})}{\sqrt{N_g} \times \sqrt{N_s}} \tag{2}$$

where N_g and N_s are the number of pairs of items which do not belong to the same equivalence class in the ground truth list and the system generated list, respectively. The value of τ_b ranges from -1 to 1. 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. If two items are in an equivalence class in at least one of the lists then the pair is neither concordant nor discordant. 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, there is no association between the lists. Figure 16 shows the Kendall tau-b rank correlation coefficient for the queries of Table 1 on the Jamendo and YAGO datasets. As we can see, all the values are positive and in most cases are 0.8 or higher (in the -1 to 1 scale).



(a) Jamendo dataset

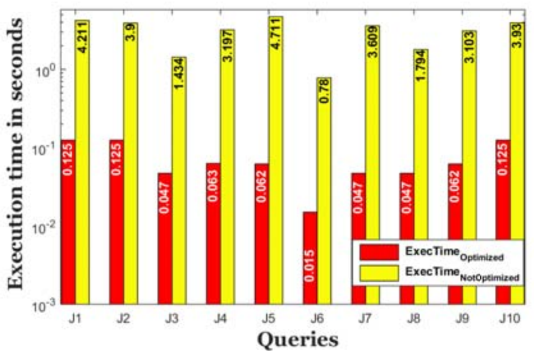


(b) YAGO dataset

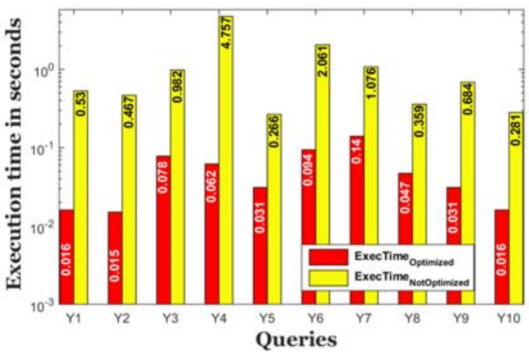
Fig. 16. Kendall tau-b coefficient for the queries on the Jamendo and YAGO datasets

7.3. *Efficiency of the system in producing relaxed results*

In order to asses the feasibility of our system, we ran our algorithm on the pattern graphs selected by the user for the queries of Table 1, and we measured the time needed to produce the first three consecutive nonempty relaxed pattern graphs and their answers. Many more relaxed pattern graphs are typically produced and ranked in the background, and a number of them are checked for empty answers. The queries were selected so that the original pattern graph for almost all of them has an empty answer. The Yago and the Jamendo datasets are stored in a relational database with one fully indexed relation for every distinct property and relationship in the datasets. To assess the efficiency of the system we evaluated the queries: (a) over the base relations with a cold cache, and (b) using the multi-query optimization and caching techniques presented in Section 6.3. Figure 17 shows the measured times.



(a) Jamendo dataset



(b) YAGO dataset

Fig. 17. Efficiency improvement achieved my Multiquery optimization on the Jamendo and YAGO datasets

One can see that the displayed times for all the queries are interactive. Further the optimization techniques are shown to substantially improve the execution time of the algorithm, in most cases by more than one order of magnitude. The time for constructing and ranking the relaxed pattern graphs are not significant compared to relaxed pattern graph evaluation time and they are not shown in Figure 17.

8. Related Work

Over the years, different approaches have been proposed and algorithms have been designed to deal with the challenges posed by keywords based search on graph data.

Results as Trees: A number of papers address keyword search on graph data and return answers which are trees [1, 2, 3, 4, 5]. In [1], a backward search algorithm, called BANK is presented for finding Steiner trees. The problem of finding Steiner trees is NP-complete. Different techniques are used to work around NP-completeness. In [3], a dynamic programming approach applicable to only few keywords and having an exponential time complexity is employed. In [5], a polynomial delay algorithm is introduced. The algorithm in [2] produced trees rooted at distinct vertices. This algorithm was supplemented by BLINK [4] with an efficient indexing structure.

Results as Graphs: Although, tree-based methods produce succinct answers, answers from graph-based methods are more informative. However, only few contributions [6, 7, 8] are there in the literature which return answers as graphs, subgraphs of the data graph. A recent graph-based approach [6] computes all possible r-radius Steiner graphs and indexes them. This method is prone to produce redundant results since it is possible that a highly ranked r-radius Steiner graph is included in another Steiner graph having a larger radius. The algorithm in [7] finds multi-centered subgraphs called communities containing all the keywords, such that there exists at least one path of distance less than or equal to R_{max} between every keyword instance and a center vertex. Later in [8], r-cliques containing all the keywords are found such that the distance between any two keywords matching vertices is no more than r. Finding r-clique with the minimum weight is an NP-hard problem. Hence, the authors provided an algorithm with polynomial delay to find the top-k r-cliques where r is an input to the algorithm. Predicting an optimal r for producing r-cliques is a challenge because it is possible that there exists no clique with that r or less.

Keyword Search on RDF Graphs: All the above approaches are proposed for generic graphs, and cannot be used directly for keyword search over RDF graph data. This is because, the edges of an RDF graph represents predicates, and predicates can be matched by the keywords of a keyword query. The approaches proposed for keyword search on RDF data can be classified into two categories: (a) data-based approaches, and (b) schema-based approaches.

Data-based approaches [12, 33] explore the data graph and retrieve subtrees/subgraphs connecting all the keywords of the query. It is to be noted that in contrast to structured databases, in a semi-structured database setting there is a lot of work which adopts schema-agnostic approaches. This is expected since unlike semi-structured data, the data stored in relational databases must adhere to a specific schema. Although these approaches generate precise answers, they are prone to produce a plethora of candidate results posing a challenge in: (a) identifying relevant results and (b) scaling satisfactorily with a growing size of the data and/or number of keywords in a query. Our approach avoids these problems by exploiting the structural summary of the data graph and user feedback in selecting a relevant pattern graph.

Schema-based approaches [10, 11, 14, 19, 20, 34] rely also on a reduced size structural summary graph extracted from the RDF data graph by mapping RDF classes to nodes and the properties between the entities of two classes to edges between these class nodes labeled by the property names. The process of keyword query evaluation with schema-based approaches involve two phases. The first phase involves finding the schema elements that match the query keywords and then generating the schema for all possible possible results of the query in the form of pattern graphs. Since the structural summary of an RDF graph is typically much smaller that its corresponding data graph, these approaches allow for an efficient exploration of the structural summary and generation of all possible pattern graphs. These pattern graphs can be expressed as structured queries (e.g., SPARQL). The second phase consists of evaluating these structured queries over the data graph to retrieve query answers. Given that keyword search is ambiguous these approaches often exploit relevance feedback from the users in order to identify users' intent [11, 14, 35]. A hierarchical clustering mechanism and user interaction at multiple levels of the hierarchy can be used to facilitate disambiguation of the keyword query and to support the computation of the relevant results. Such a mechanism is suggested in [36, 37] in the context of tree data and in [14] in the context of RDF data. Although summary based approaches proved to have better performance scalability compared to data-based approaches, they provide an approximate solution and they might miss relevant results for a given keyword query. As RDF data graphs are practically schema free, a summary graph extracted from an RDF graph cannot capture completely all the information in the RDF graph.

Result Space Expansion by Relaxation Techniques: In this paper we provide a pattern graph relaxation technique to address the issue related to the use of the structural summary. Relaxation techniques are studied in [38, 39, 40] in connection with XML data in order to expand the result space of a query. These techniques have been developed in a different context since they are applied to queries over tree data. Further, their goals and processes are different: reference [40] relaxes weighted tree pattern queries with descendant edges in order to permit approximate matching on XML data. Reference [39] provides a framework for generating similar satisfiable queries, when the user tree pattern query is unsatisfiable. Reference [38] relaxes the MaxMatch semantics [41] of keyword queries on XML data so that they also return LCA nodes which are not SLCA nodes. In contrast, given a user keyword query on an RDF graph, we use the structural summary to construct pattern graphs which represent different interpretations of the keyword query. Then, we gradually generate and rank relaxed versions of one of them selected by the user as the most relevant interpretation to her intent. The goal is to expand the (possibly empty) result space of the relevant pattern graph with results which are close to the user intent by evaluating relaxed pattern graphs according to their rank on the RDF graph. Relaxing conjunctions of attribute-value pairs has been considered in [42]. However, this work does not consider at all relaxing structural information in pattern graphs which is the focus of this paper. Relevant to our work are also the papers [43, 44] which focus on relaxing a conjunctive SPARQL query. Their approach involves substituting class or relationship labels in the query graph by related terms or variables. Instead, our approach is a relaxation approach for a pattern graph produced from a keyword query. As such it is not comparable to existing relaxation approaches like [43, 44]: our query evaluation process involves a disambiguation phase during which the user explicitly selects class and relationship labels and the pattern graph reflecting his intent. That is, class and relationship labels are invariants in our context. Our approach cannot be compared to previous RDF relaxation approaches as they attempt to relax what in our approach is an invariant. Reference [45] is a preliminary version of part of the work presented in the current paper.

9. Conclusion

Exploiting the structural summary has emerged in recent years as a promising technique for evaluating keyword queries over RDF graphs. Structural summary-based approaches compute pattern graphs (structured queries) as possible interpretations of the unstructured keyword query and often rely on user feedback to identify the pattern graph which is most relevant to the user intent. However, since summaries are approximate representations of the data, these approaches might return empty answers or miss results which are relevant to the user intent. To address the drawback while maintaining the advantages of these approaches, we have presented a novel approach that permits the relaxation of the most relevant pattern graph selected by the user and expands its result space with similar results. We used pattern graph homomorphisms to introduce relaxed pattern graphs. We then defined an operation on pattern graphs and we prove that it is sound and complete with respect to relaxed pattern graphs. In order to characterize the semantic closeness of relaxed pattern graphs to the original pattern graph, we introduced different syntax and semantic-based metrics that allow us to compare the degree of relaxation of relaxed pattern graphs. We provided results to identify the reasons for a pattern graph having an empty answer and we use them to design an algorithm which computes relaxed pattern graphs with non-empty answers in ascending relaxation order. We design optimization techniques that exploit subquery caching and multiquery optimization to support the computation of relaxed pattern graphs. Our experimental results demonstrate the effectiveness of our approach in ranking the relaxed pattern graphs and the efficiency of our system and optimization techniques in producing relaxed pattern graphs and their answers.

We are currently working on comparing results from different relaxed pattern graphs. The goal is to identify results which turn out to be more relevant to the original pattern graph due to additional connections between their vertices discovered in the result sets of other relaxed pattern graphs.

References

1. Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.
2. Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.
3. Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.
4. Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.
5. Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in com-

plex data graphs. In *SIGMOD*, pages 927–940, 2008.

6. 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 *SIGMOD*, pages 903–914, 2008.
7. Lu Qin, Jeffrey Xu Yu, Lijun Chang, and Yufei Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.
8. Mehdi Kargar and Aijun An. Keyword search in graphs: Finding r-cliques. *VLDB*, pages 681–692, 2011.
9. Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.
10. Haofen Wang, Kang Zhang, Qiaoling Liu, Thanh Tran, and Yong Yu. Q2semantic: A lightweight keyword interface to semantic search. In *ESWC*, pages 584–598, 2008.
11. Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. *ICDE*, pages 405–416, 2009.
12. Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching RDF graphs with sparql and keywords. *IEEE Data Eng. Bull.*, pages 16–24, 2010.
13. Ziyang Liu and Yi Chen. Processing keyword search on XML: a survey. *WWW*, pages 671–707, 2011.
14. Ananya Dass, Cem Aksoy, Aggeliki Dimitriou, and Dimitri Theodoratos. Exploiting semantic result clustering to support keyword search on linked data. In *WISE*, pages 448–463, 2014.
15. Ananya Dass, Aggeliki Dimitriou, Cem Aksoy, and Dimitri Theodoratos. Incorporating cohesiveness into keyword search on linked data. In *WISE*, pages 47–62. Springer, 2015.
16. Ananya Dass and Dimitri Theodoratos. Trading off popularity for diversity in the results sets of keyword queries on linked data. In *ICWE 2017*, page 18, 2017.
17. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*, pages 133–144, 2002.
18. Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.
19. Haizhou Fu, Sidan Gao, and Kemafor Anyanwu. Disambiguating keyword queries on RDF databases using "Deep" segmentation. In *ICSC*, pages 236–243, 2010.
20. Kaifeng Xu, Junquan Chen, Haofen Wang, and Yong Yu. Hybrid graph based keyword query interpretation on RDF. In *ISWC*, 2010.
21. Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *PVLDB*, 1(1):1189–1204, 2008.
22. Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.
23. Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.
24. Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on SPARQL views. In *WWW*, pages 655–664, 2011.
25. Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.
26. Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995.
27. Timos Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge & Data Engineering*, (2):262–266, 1990.
28. Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. Factorizing YAGO: scalable machine learning for linked data. In *WWW*, pages 271–280, 2012.
29. Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval - the con-*

cepts and technology behind search, Second edition. Pearson Education Ltd., Harlow, England, 2011.
30. Alan Agresti. *Analysis of ordinal categorical data*, volume 656. John Wiley & Sons, 2010.
31. Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.
32. Maurice G Kendall. A new measure of rank correlation. *Biometrika*, pages 81–93, 1938.
33. Shady Elbassuoni and Roi Blanco. Keyword search over RDF graphs. In *CIKM*, pages 237–242, 2011.
34. Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. Scalable keyword search on large RDF data. *IEEE Trans. Knowl. Data Eng.*, 26(11):2774–2788, 2014.
35. Mengxia Jiang, Yueguo Chen, Jinchuan Chen, and Xiaoyong Du. Interactive predicate suggestion for keyword search on RDF graphs. In *ADMA (2)*, pages 96–109, 2011.
36. Xiping Liu, Changxuan Wan, and Lei Chen. Returning clustered results for keyword search on XML documents. *IEEE Trans. Knowl. Data Eng.*, pages 1811–1825, 2011.
37. Cem Aksoy, Ananya Dass, Dimitri Theodoratos, and Xiaoying Wu. Clustering query results to support keyword search on tree data. In *WAIM*, pages 1–12, 2014.
38. Lingbo Kong, Rémi Gilleron, and Aurélien Lemay Mostrare. Retrieving meaningful relaxed tightest fragments for XML keyword search. In *EDBT*, pages 815–826, 2009.
39. Tali Brodianskiy and Sara Cohen. Self-correcting queries for xml. In *CIKM*, pages 11–20, 2007.
40. Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.
41. Ziyang Liu and Yi Cher. Reasoning and identifying relevant matches for xml keyword search. *Proceedings of the VLDB Endowment*, 1(1):921–932, 2008.
42. Davide Mottin, Alice Marascu, Senjuti Basu Roy, Gautam Das, Themis Palpanas, and Yannis Velegrakis. A probabilistic optimization framework for the empty-answer problem. *Proceedings of the VLDB Endowment*, 6(14):1762–1773, 2013.
43. Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. Query relaxation for entity-relationship search. In *ESWC*, pages 62–76, 2011.
44. Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum. Language-model-based ranking for queries on rdf-graphs. In *CIKM*, pages 977–986, 2009.
45. Ananya Dass, Cem Aksoy, Aggeliki Dimitriou, and Dimitri Theodoratos. Keyword pattern graph relaxation for selective result space expansion on linked data. In *ICWE*, pages 287–306, 2015.