Keyword Pattern Graph Relaxation for Selective Result Space Expansion on Linked Data

Ananya Dass[1], Cem Aksoy[1], Aggeliki Dimitriou[2], and Dimitri Theodoratos[1(✉)]

[1] New Jersey Institute of Technology, Newark, USA
dth@njit.edu
[2] National Technical University of Athens, Athens, Greece

Abstract. Keyword search is a popular technique for querying the ever growing repositories of RDF graph data. In recent years different approaches leverage a structural summary of the graph data to address the typical keyword search related problems. 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 show that 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. Finally, we run experiments to measure the effectiveness of our ranking of relaxed pattern graphs and the efficiency of our system.

1 Introduction

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 comes along with a drawback. Keyword queries are imprecise and ambiguous. For this reason, keyword queries return a very large number of results. This is a typical problem

P. Cimiano et al. (Eds.): ICWE 2015, LNCS 9114, pp. 287–306, 2015.
DOI: 10.1007/978-3-319-19890-3_19

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 [6,10,14,16,19,20,23,24]. 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 [12,15,25,26]. 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. Further, despite the size restriction of the candidate results, 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 increases.

Leveraging the Structural Summary. In order to address these challenges recent approaches to keyword search on RDF data developed techniques which exploit a structural summary of the RDF graph [9,25,26]. This is a concept similar to the 1-index or data guide 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 RDF 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 RDF graph, the candidate results for the keyword query can be produced. Interestingly, a pattern graph can be expressed as a SPARQL query, and all the machinery of query engines and optimization techniques developed for SPARQL can be leveraged to efficiently compute the results of the keyword query.

Benefits of the Structural Summary Approach. A structural summary approach can resolve the challenges mentioned above. Indeed, the pattern graphs can be ranked using a scoring function and the top-k of them be presented to the user. The user chooses one that best meets her intention, and only the corresponding structured query is evaluated against the data graph [25,26]. A more recent approach exploits semantic interpretations for the query keywords and a hierarchical clustering of the pattern graphs in order to select a relevant one [9]. Effectiveness studies show that the approaches based on the structural summary display good precision. 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 [6,19]. In contrast, since the structural summaries are typically much

smaller than the actual data, generating the relevant pattern graph 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 [8,9,25].

The Missing Relevant Result Problem. Despite its advantages, the structural summary-based approach for keyword search on RDF data has a drawback: since the structural summaries are approximate representations of the RDF graph data, the selected pattern graph might miss results which are relevant to the user intent. This might happen even if the user correctly selects the pattern graph which is relevant to her intent.

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 enables gradual relaxation of a relevant pattern graph so that additional results of possible interest to the user are retrieved from the RDF graph, if needed (for example, if the original pattern graph returns no result or if the user wants to extract more semantically similar results).

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

- We leverage pattern graph homomorphisms to define relaxed pattern graphs. Relaxed pattern graphs can expand the result space of an original pattern graph with semantically similar results (Section 3.1).
- 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 two split images of an entity variable vertex in a pattern graph and partitions its incident edges between the two vertices. We show that this operation is complete, that is, it can produce all the relaxed pattern graphs (Section 3.2).
- Since we want to relax a pattern graph so that the relaxed version is as close to the initial pattern graph as possible, we introduce three metrics to compare the degree of relaxation of relaxed pattern graphs and rank them. 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 (Section 3.3).
- 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 (Section 3.4).

- 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 graphs (and computes their answer in the RDF graph) in ascending order of relaxation (Section 3.5).
- We run experiments to measure the effectiveness of our ranking of relaxed pattern graphs and the efficiency of our system in computing relaxed pattern graphs and their answers. The results showed to be promising (Section 4).

2 Structural Summaries and Pattern Graphs

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. 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 which represent entity relationships), E_P (entity to value edges called *Property* edges which represent property assignments), E_T (entity to class edges called *type* edges which represent entity to class membership) and E_S (class to class edges called *subclass* edges which represent class-subclass relationship).
L is a finite set of labels that includes the labels "type" and "subclass".
l is a function from $V_C \cup V_V \cup E_R \cup E_P$ to L. That is, l assigns labels to class and value vertices and to relationship and property edges.

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. Fig. 1 shows an example RDF graph (inspired by the Jamendo dataset[1]). For simplicity, vertex and edge identifiers are not shown in this example graph.

Query Language Semantics. A *query* Q on an RDF graph G is a set of keywords. A *keyword instance* of a keyword k in Q is a vertex or edge label in G containing k. The *answer* of Q on G is a set of result graphs of Q on G. Each result graph is a minimal subgraph of G involving at least one instance of every keyword in Q and is formally defined below. In order to facilitate the interpretation of the semantics of the keyword instances, every instance of a keyword in Q is matched against a small subgraph of G which involves this

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

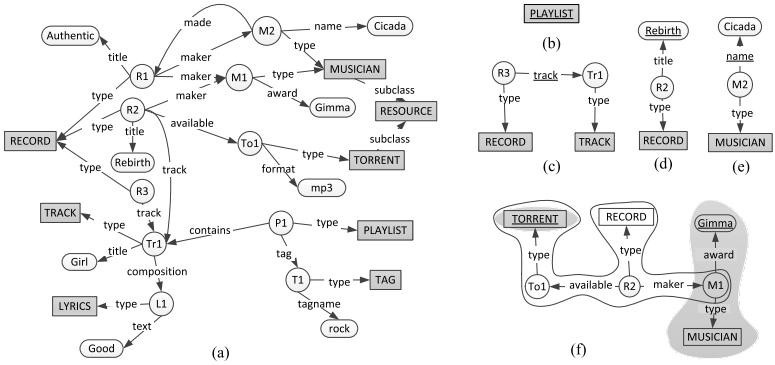

Fig. 1. (a) An RDF graph, (b), (c), (d) and (e) class, relationship, value and property matching constructs, respectively, (f) inter-construct connection and result graph

keyword instance and the corresponding class vertices. This subgraph is called *matching construct*. Figs. 1(b), (c), (d) and (e) show a class, relationship, value and property matching construct, respectively, for different keyword instances in the RDF graph of Fig. 1(a). Underlined labels in a matching construct denote the keyword instances. Each matching construct provides information about the semantic context of the keyword instance under consideration. For instance, the matching construct of Fig. 1(d) shows that `Rebirth` is the title of entity $R2$ of type Record.

A *signature* of Q is a function that matches every keyword k in Q to a matching construct of k in G. 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. Fig. 1(f) shows an inter-construct connection between the matching constructs for keywords `Torrent` and `Gimma` in the RDF graph of Fig. 1(a). The matching constructs are shaded and the inter-construct connection is circumscribed.

A subgraph of G is said to be *connection acyclic* if there is no cycle in the graph obtained by viewing its matching constructs as vertices and its inter-construct connections between them as edges. Given a signature S for Q on G, a *result graph* of S on G is a connected, connection acyclic subgraph of G which contains only the matching constructs in S and possibly inter-construct connections between them. A *result graph* for Q on G is a result graph for a signature of Q on G. Fig. 1(f) shows a result graph for the query {`Torrent, Cicada`} on the RDF graph of Fig. 1(a).

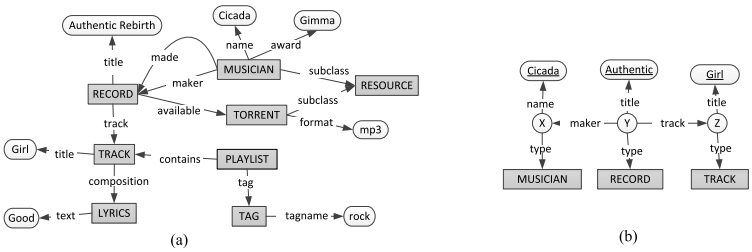

Fig. 2. (a) Structural Summary, (b) Query Pattern Graph

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 is a graph that summarizes the RDF graph.

Definition 2 (Structural Summary). The *structural summary* of an RDF graph G is a vertex and edge labeled graph constructed from G as follows:

1. 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.
2. 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. Merge also the corresponding edges into one edge labeled by their label.
3. Merge all the relationship edges between the same class vertices which are labeled by the same label into one edge with that label.

Fig. 2(a) shows the structural summary for the RDF graph G of Fig. 1. Similarly to matching constructs on the data graph we define matching constructs on the structural summary. Since the structural summary does not have entity vertices, a matching construct on a structural summary possess one distinct entity variable vertex for every class vertex labeled by a distinct variable.

Pattern graphs are the 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 3 (Pattern Graph). A *(result) 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 labels.
(b) The labels of the value vertices are replaced by distinct variables whenever these labels are not the keyword instances in the result graph. These variables are called *value variables* and they range over value labels in the RDF graph.

Fig. 2(b) shows an example of a pattern graph, for the keyword query $Q = \{\texttt{Cicada, Authentic, Girl}\}$ on the RDF graph of Fig. 1.

Computing and Selecting Pattern Graphs. For computing the pattern graphs of a query on the structural summary, we use an algorithm which computes r-radius Steiner graphs [9]. The user selects a pattern graph by navigation through a two-level semantic hierarchical clustering system [9]. Nevertheless, the way the pattern graph is selected by the user is orthogonal to the relaxation method we present in this paper. Any other approach like those in [13,25–27] can be used for selecting the relevant pattern graph which will be relaxed.

3 Computing Relaxed Pattern Graphs

In order to expand the result set of the 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. Relaxed pattern graphs which are semantically closer to the original pattern graph are preferable. Therefore, we introduced different metrics to compare the degree of relaxation of relaxed pattern graphs and characterize their closeness to the original pattern graph. Then, we elaborate on the reasons for a pattern graph having an empty answer. Finally, we design an algorithm which computes relaxed pattern graphs with non-empty answers ranked in ascending order of their degree of relaxation.

3.1 Relaxed Pattern Graphs

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

Definition 4 (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.

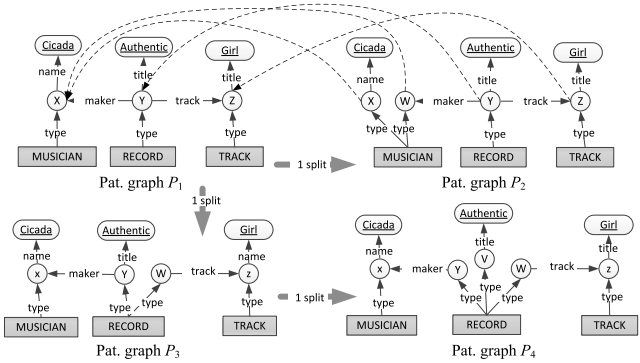

Fig. 3. An original pattern graph P_1 and relaxed pattern graphs P_2, P_3, P_4

(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.

Fig. 3 shows 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 P_3 and P_4 to P_1. However, there is no homomorphism from P_1 to any one of the other pattern graphs.

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

Definition 5 (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 Fig. 3, $P_1 \prec P_2$ and $P_1 \prec P_3 \prec P_4$. No other $\prec$ relationships hold 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. 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.

3.2 Vertex Splitting

A pattern graph is relaxed by applying the *vertex split* operation to one or more of its entity variable vertices. This operation "splits" an entity variable vertex in

a pattern graph into two entity variable vertices of the same type and partitions the incident edges between the two new vertices as indicated by the operation.

Definition 6 (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 Fig. 3. 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 $split(P_3, Y, \{title\})$ produces the pattern graph P_4.

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 operations. Therefore, one can see that if P_1 and P_2 are two pattern graphs, $P_1 \prec P_2$ iff P_2 can be produced from P_1 by applying a sequence of vertex split operations. In other words, the vertex split operation is sound and complete w.r.t. relaxed pattern graphs.

3.3 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 connectivity of the pattern graph. In order to define the 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 Fig. 4(b), the keyword instances `Rebirth` and `mp3` are tightly connected whereas the keyword instances `Cicada` and `Gimma` are not.

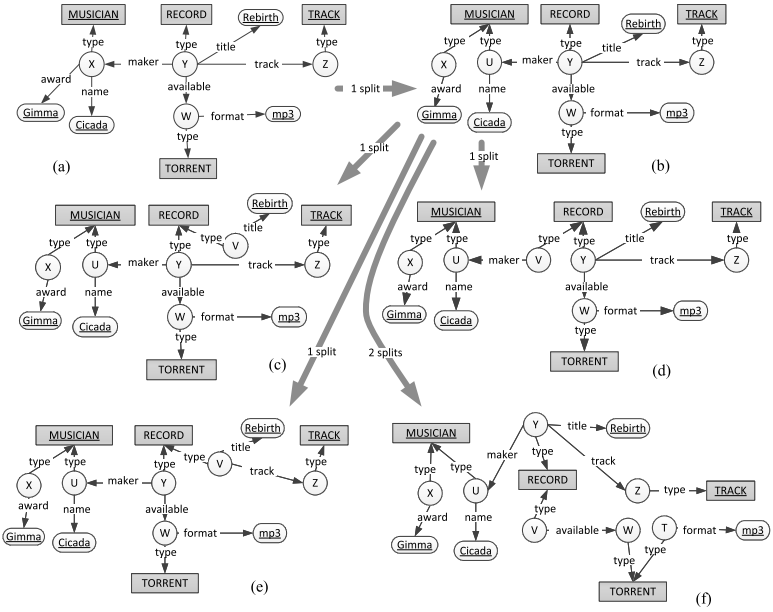

Fig. 4. (a) Original pattern graph (b), (c), (d), (e) and (f) relaxed pattern graphs

Definition 7 (Pattern graph connectivity). The *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 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 same connectivity. For instance, the connectivity of the pattern graph in Fig. 4(a) is 1. The connectivity of the relaxed pattern graph of Figs. 4(b) is 0.73, the connectivity of that of Fig. 4(c) is 0.47 and the connectivity of those of Figs. 4(d), (e) and (f) is 0.33.

In order to distinguish between relaxed pattern graphs of the same pattern graph which have the same connectivity, we introduce another 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. For instance, in the pattern graph of Fig. 4(d) the distance between the tightly connected keyword instances of `Rebirth` and `mp3` is 2 while the distance between `Gimma` and `Musician` is 1.

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 these sets.

Definition 8 (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.

A relaxed pattern graph with smaller scatteredness is preferred over a pattern graph of the same connectivity but higher scatteredness since its keywords are assumed to be more closely related. In the example of Fig. 4, the patterns (d) and (e) have the same connectivity 0.33. However, the scatteredness of the pattern of Fig. 4(d) is 1 and that of the pattern of Fig. 4(e) is 2. We use the pattern graph scatteredness to rank the relaxed patterns of a pattern graph having the same connectivity. In our running example, the pattern of Fig. 4(d) is ranked before the pattern of Fig. 4(e).

Nevertheless, it is possible that multiple relaxed patterns of the same pattern graph have not only the same connectivity but also the same scatteredness. In order to differentiate between the degree of relaxation of such pattern graphs, we employ a third 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. For instance, in the pattern graph of Fig. 4(d), the split distance of the keyword instances of `Gimma` and `mp3` is 2 and that of `Gimma` and `Musician` is 0.

Definition 9 (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.

For example, the pattern graphs of Fig. 4(e) and (f) have the same connectivity (0.33) and the same scatteredness (2) whereas their keyword dispersion is 10 and 14, respectively. A pattern graph with a lower keyword dispersion is preferred over a pattern graph with a higher keyword dispersion. Hence, the pattern graph of Fig. 4(e) will be ranked higher than that of Fig. 4(f).

Given two pattern graphs P_1 and P_2, we say that, P_2 is equally or more relaxed than P_1, and we write $P_1 \leq_r P_2$, if: (a) $\text{conn}(P_1) \geq \text{conn}(P_2)$, or (b) $\text{conn}(P_1) = \text{conn}(P_2)$ and $\text{scatt}(P_1) \leq \text{scatt}(P_2)$, or (c) $\text{conn}(P_1) = \text{conn}(P_2)$ and $\text{scatt}(P_1) = \text{scatt}(P_2)$ and $\text{disp}(P_1) \leq \text{disp}(P_2)$. 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 are ranked first, we say that it is ranked in *relaxation order*. Since split operations introduce additional type edges in the pattern graph it is not difficult to see that given two pattern graphs P_1 and P_2, if $P_1 \prec P_2$ then $P_1 \leq_r P_2$.

3.4 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 10 (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.

We provide next conditions to characterize 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 Fig. 5). We call the graph of Fig. 5 the *star-join view* of the entity variable vertex X in P. One can see 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.

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

3.5 An Algorithm for Computing Relaxed Patterns

We provide now an algorithm which, given the pattern graph P chosen by the user, gradually generates relaxed pattern graphs of P having non-empty answers. The algorithm returns these pattern graphs and their answers in relaxation order. The number of relaxed pattern graphs returned is controlled by the user.

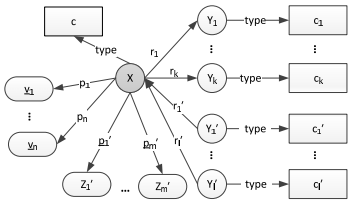

Fig. 5. Star-join view of entity variable vertex X

Algorithm 1.

Input: P: pattern graph.

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

1: $R = \{P\}$;
2: $MoreResults = True$;1
3: $Ans = \emptyset$;
4: **while** $R \neq \emptyset$ and $MoreResults$ **do**
5: $P_{Top} \leftarrow$ the pattern 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: $NewR \leftarrow GetRelaxedFromEmpty(P_{Top}, EV)$; ▷ all the relaxed patterns obtained by applying one vertex split operation to all empty vertices.
11: Rank the patterns 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: $MoreR \leftarrow GetRelaxed(P_{Top})$; ▷ all the new relaxed patterns obtained by applying one vertex split operation to a candidate split vertex.
20: Rank the patterns in R in ascending relaxation order;
21: $R \leftarrow$ merge R and $MoreR$ into one list of patterns ranked in ascending relaxation order;

The outline of our algorithm is shown in Algorithm 1. The input of this algorithm is an original pattern graph P. 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 ranks the pattern graphs in R in ascending relaxation order (line 5). The pattern graph P_{Top} with the highest rank is chosen from R and its unmarked vertices are checked for emptiness (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 P_{Top} has empty vertices, it is further relaxed by applying one vertex split operation to all of its empty vertices in all possible ways. The resulting relaxed pattern graphs form a new list $NewR$ of relaxed pattern graphs (lines 9-10), which is then ranked in ascending relaxation order and is merged with the list R (lines 11-12). In contrast, if P_{Top} does not have any empty vertex, it is evaluated over the data graph and the set Ans of result graphs is non-empty, it is returned to the user along with the corresponding pattern graph P_{Top} (lines 14-16). In case the user wants more results, or the

Table 1. The keyword queries

Query #	Keywords	Chosen Pattern Graph Structure
1	teenage, text, fantasie, document	star-chain
2	signal, onTimeLine, 10002, recorded_as, sweet	chain
3	kouki, recorded_as, knees	star-chain
4	briareus, reflection, cool, girl	star
5	kouki, revolution, electro, good	star
6	nuts, spy4, chillout, track	star
7	biography, guitarist, track, lemonade	chain
8	divergence, track, obssession, format, mp32	star-chain
9	fantasie, performance, recorded_as, slipstream	chain
10	signal, recorded_as, fantasie, onTimeLine, 10001	chain

pattern graph P_{Top} produces an empty answer when evaluated over the data graph, P_{Top} is relaxed by applying one vertex split operation to all of its candidate split vertices in all possible ways and the generated relaxed pattern graphs are stored in a list $MoreR$ (lines 18-19). All the elements of $MoreR$ are then ranked and merged with the list R of pattern graphs (line 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 we can deduce that Algorithm 1 correctly computes its relaxed pattern graphs with non-empty answers in relaxation order.

4 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.

Dataset and Queries. We use Jamendo, a large repository of Creative Commons licensed music. Jamendo is a real dataset of 1.1M triples and of 85MB size containing information about musicians, music tracks, records, licenses and many other details related to them. Experiments are conducted on a standalone machine with an Intel i5-3210M @ 2.5GHz processors and 8GB memory.

Users were provided with different queries on the Jamendo dataset and selected the most relevant pattern graph among those provided by the system. We report on 10 queries. The queries cover a broad range of cases. They involve

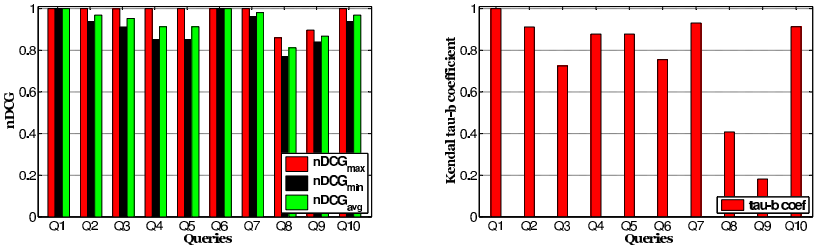

Fig. 6. (a) nDCG$_{max}$, nDCG$_{min}$ and nDCG$_{avg}$ (b) Kendall tau-b coefficient

from 3 to 5 keywords and their relevant pattern graph form a star or a chain or a combination of them. Table 4 shows the keyword queries and information about them.

Effectiveness in Ranking Relaxed Pattern Graphs. For our effectiveness experiments, we used expert users to determine the ground truth. For each query, a user selected, among those computed by the system, the pattern graph which is most relevant to the query. This is the original pattern graph. The user then is presented with all the relaxed pattern graphs that are generated by our algorithm (see Algorithm 1, Section 3.5) until the third relaxed pattern graph with a non-empty answer is produced and ranks them w.r.t to their semantic closeness to the original pattern graph. In order to measure the effectiveness of our technique in generating a ranked list of relaxed pattern graphs (ranked in relaxation order as described in Section 3.3), we are using two metrics: (a) *normalized discounted cumulative gain* (nDCG) [17], and (b) *Kendall tau-b rank correlation coefficient* [1]. Both of them allow comparing two ranked lists of items. Note that the list produced by the user (the correct ranked list) and one produced by our system might not form strict total orders. That is, there might be ties (relaxed pattern graphs with the same rank). We call the set of relaxed pattern graphs that have the same rank in a ranked list equivalence class. Equivalence classes need to be taken into account in measuring the similarity of the ranked lists.

The nDCG metric was first introduced in [17] 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 the item. 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. 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

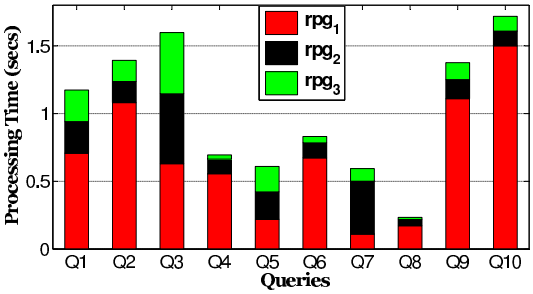

Fig. 7. Evaluation time for three consecutive relaxed pattern graphs

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 correct list 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.

In order to take into account equivalent classes of pattern graphs in the 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 equivalence classes correctly (that is, in compliance with their ranking in the correct 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 the 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 the equivalence classes in all possible ways. The nDCG values range between 0 and 1. Fig. 6(a) shows the nDCG_{min}, nDCG_{max} and nDCG_{avg} values for the queries of Table 1. As one can see, all the values are very close to 1.

The Kendall tau rank correlation coefficient [1] was proposed to address the problem of measuring the association between two different rankings of the same set of items. In our context, we want to see if the comparison of the ranked list produced by our system (the relaxation order) with the correct ranked list (defined by the expert 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* [1], which can deal with equivalent classes of items in the ranked lists. The value of the coefficient 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*). If two items are in an equivalence class in at least one of the lists then the pair is neither concordant nor discordant. When the number of concordant pairs is much larger (resp. much less) than the number of discordant pairs, then the two lists are positively (resp. negatively) correlated, and the coefficient is close to 1 (resp. -1). When the number of discordant and concordant pairs are about the same, then the two lists are weakly correlated (the coefficient is close to 0). Fig. 6(b) shows the Kendall tau-b rank correlation coefficient for the queries of Table 1. As we can see, all the values are positive and in most cases very close to 1.

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 4, 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. Fig. 7 shows the measured times. One can see that the displayed times for all the queries are interactive. The times needed to produce the second and third relaxed pattern graphs are usually shorter than for the previous relaxed pattern graphs since information about non-empty vertices is recorded and propagated down to the relaxations of the pattern graphs.

5 Related Work

In recent years, a number of papers address keyword search on graph data. Most of these approaches return answers which are trees [5,6,8,12,16,18,21]. Only few of them [20,23,24] return answers as graphs, subgraphs of the data graph. 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 represent predicates, which can also 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 [11,12] and (b) schema-based approaches [9,13,22,25–27]. Data-based approaches rely on the data graph to produce answers. Although these approaches generate precise answers, they fail to scale well when the size of the data increases. In contrast, summary-based approaches rely also on a reduced size structural summary extracted from the data. In order to compute answers, these approaches focus on capturing the interpretations of a keyword query by mapping the keywords to elements of the structural summary and constructing pattern graphs. Given that keyword search is ambiguous, these approaches often exploit relevance feedback from the users in order to identify the users' intent [9,18,25]. 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 [2] in the context

of tree data and in [9] 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. In this paper we provide a pattern graph relaxation technique to address this issue. Relaxation techniques are studied in [5,7,21]. These techniques are not directly related to our work since they are developed for XML and not RDF data and their goals and processes are different.

6 Conclusion

To address the drawback of structural summary-based approaches for keyword search on RDF graphs, while maintaining their advantages, 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 show that it is sound and complete w.r.t. 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 studied properties of pattern graphs with empty answers and we use them to design an algorithm which computes relaxed pattern graphs with non-empty answers in ascending relaxation order. Our experimental results demonstrate the effectiveness of our approach in ranking the relaxed pattern graphs and the efficiency of our system in producing relaxed pattern graphs and their answers.

We are currently working on exploiting common subexpressions between relaxed pattern graphs and applying multiquery optimization techniques to further improve the performance of our system in computing keyword queries over large RDF repositories.

References

1. Agresti, A.: Analysis of ordinal categorical data. John Wiley & Sons (2010)
2. Aksoy, C., Dass, A., Theodoratos, D., Wu, X.: Clustering query results to support keyword search on tree data. In: Li, F., Li, G., Hwang, S., Yao, B., Zhang, Z. (eds.) WAIM 2014. LNCS, vol. 8485, pp. 213–224. Springer, Heidelberg (2014)
3. Aksoy, C., Dimitriou, A., Theodoratos, D.: Reasoning with patterns to effectively answer XML keyword queries. The VLDB journal 24(3): 441–465 (2015)

4. Aksoy, C., Dimitriou, A., Theodoratos, D., Wu, X.: *XReason*: a semantic approach that reasons with patterns to answer XML keyword queries. In: Meng, W., Feng, L., Bressan, S., Winiwarter, W., Song, W. (eds.) DASFAA 2013, Part I. LNCS, vol. 7825, pp. 299–314. Springer, Heidelberg (2013)
5. Amer-Yahia, S., Cho, S.R., Srivastava, D.: Tree pattern relaxation. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 496–513. Springer, Heidelberg (2002)
6. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: ICDE, pp. 431–440 (2002)
7. Brodianskiy, T., Cohen, S.: Self-correcting queries for XML. In: CIKM (2007)
8. Dalvi, B.B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. PVLDB **1**(1), 1189–1204 (2008)
9. Dass, A., Aksoy, C., Dimitriou, A., Theodoratos, D.: Exploiting semantic result clustering to support keyword search on linked data. In: Benatallah, B., Bestavros, A., Manolopoulos, Y., Vakali, A., Zhang, Y. (eds.) WISE 2014, Part I. LNCS, vol. 8786, pp. 448–463. Springer, Heidelberg (2014)
10. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: ICDE, pp. 836–845 (2007)
11. Elbassuoni, S., Blanco, R.: Keyword search over RDF graphs. In: CIKM (2011)
12. Elbassuoni, S., Ramanath, M., Schenkel, R., Weikum, G.: Searching RDF graphs with sparql and keywords. IEEE Data Eng. Bull., 16–24 (2010)
13. Fu, H., Gao, S., Anyanwu, K.: Disambiguating keyword queries on RDF databases using "Deep" segmentation. In: ICSC, pp. 236–243 (2010)
14. Golenberg, K., Kimelfeld, B., Sagiv, Y.: Keyword proximity search in complex data graphs. In: SIGMOD, pp. 927–940 (2008)
15. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, pp. 16–27 (2003)
16. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD, pp. 305–316 (2007)
17. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. **20**(4), 422–446 (2002)
18. Jiang, M., Chen, Y., Chen, J., Du, X.: Interactive predicate suggestion for keyword search on RDF graphs. In: Tang, J., King, I., Chen, L., Wang, J. (eds.) ADMA 2011, Part II. LNCS, vol. 7121, pp. 96–109. Springer, Heidelberg (2011)
19. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)
20. Kargar, M., An, A.: Keyword search in graphs: Finding r-cliques. VLDB (2011)
21. Kong, L., Gilleron, R., Mostrare, A.L.: Retrieving meaningful relaxed tightest fragments for XML keyword search. In: EDBT, pp. 815–826 (2009)
22. Le, W., Li, F., Kementsietsidis, A., Duan, S.: Scalable keyword search on large RDF data. IEEE Trans. Knowl. Data Eng. **26**(11), 2774–2788 (2014)
23. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: SIGMOD, pp. 903–914 (2008)
24. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Querying communities in relational databases. In: ICDE, pp. 724–735 (2009)

25. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: ICDE (2009)
26. Wang, H., Zhang, K., Liu, Q., Tran, T., Yu, Y.: Q2Semantic: a lightweight keyword interface to semantic search. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 584–598. Springer, Heidelberg (2008)
27. Xu, K., Chen, J., Wang, H., Yu, Y.: Hybrid graph based keyword query interpretation on RDF. In: ISWC (2010)