



Answers Partitioning and Lazy Joins for Efficient Query Relaxation and Application to Similarity Search

Sébastien Ferré^(✉)

Univ Rennes, CNRS, IRISA, Campus de Beaulieu, 35042 Rennes, France
ferre@irisa.fr

Abstract. Query relaxation has been studied as a way to find approximate answers when user queries are too specific or do not align well with the data schema. We are here interested in the application of query relaxation to similarity search of RDF nodes based on their description. However, this is challenging because existing approaches have a complexity that grows in a combinatorial way with the size of the query and the number of relaxation steps. We introduce two algorithms, answers partitioning and lazy join, that together significantly improve the efficiency of query relaxation. Our experiments show that our approach scales much better with the size of queries and the number of relaxation steps, to the point where it becomes possible to relax large node descriptions in order to find similar nodes. Moreover, the relaxed descriptions provide explanations for their semantic similarity.

1 Introduction

Query relaxation has been proposed as a way to find *approximate answers* to user queries [10]. It consists in applying transformations to the user query in order to relax constraints, and make it more general so that it produces more answers. In previous work on SPARQL queries over RDF(S) graphs [5, 6, 9, 14, 15], typical relaxation steps consist in removing a query element or replacing a class by a superclass or a node by a variable. The major limitation of query relaxation is that the number of relaxed queries grows in a combinatorial way with the number of relaxation steps and the size of the query.

A potential application of query relaxation is *similarity search*, i.e. the search for the nodes most similar to a given query node in an RDF graph. We propose to start with the query node description as an overly specific query, and to relax it progressively in order to find similar nodes as approximate answers. The relaxed queries can then be used as *explanations* of the similarity with each node, and for *ranking* similar nodes. However, existing algorithms for query relaxation are not efficient enough for that purpose because node descriptions make up for

This research is supported by ANR project PEGASE (ANR-16-CE23-0011-08).

large queries, and many relaxation steps are necessary in order to find approximate answers. Various definitions of semantic distance/similarity have been proposed, especially between concepts in ontologies [17] but also between structural symbolic descriptions [7, 13]. However, those distances/similarities are numerical measures, and the explanation of the similarity provided by the relaxed queries is lost. In this paper, we only consider symbolic forms of semantic similarity, i.e. similarities that can be represented with graph patterns [3, 8]. A major drawback of those approaches is that the query node has to be compared to every other node in the RDF graph, and that each comparison is costly because it consists in finding the largest overlap between two rooted graphs.

In this paper, we introduce two algorithms, *answers partitioning* and *lazy join*, that together improve the efficiency of query relaxation to the point where it can be applied effectively to semantic similarity search. Despite the higher efficiency, our approach does not trade quality for efficiency as it produces the same results as query relaxation and symbolic semantic similarity. The contribution is that our partitioning algorithm is driven by both relaxed queries and similar nodes so as to prune the space of relaxed queries while avoiding individual comparison to each node. The set of nodes is partitioned into more and more fine-grained clusters, such that at the end of the process each cluster is a set of approximate answers produced by the same relaxed query. That relaxed query is the common subsumer between the query and each approximate answer. *Lazy join* is an essential optimization of our approach to address similarity search because multi-valued properties (e.g., `hasActor`) combined with high numbers of relaxation steps imply an explosion of the size of joins.

Section 2 discusses related work. Section 3 gives preliminary definitions. Section 4 defines query relaxation, and introduces its application to similarity search. Section 5 details our answers partitioning algorithm, and Sect. 6 details its optimization with lazy joins. Section 7 report experimental studies of efficiency and effectiveness. Finally, Sect. 8 concludes and draws perspectives.

2 Related Work

The existing approaches for query relaxation consist in enumerating relaxed queries up to some edit distance, and to evaluate each relaxed query, from the more specific to the more general, in order to get new approximate answers [9, 14, 15]. The main problem is that the number of relaxed queries grows in a combinatorial way with the edit distance, and the size of the query. Moreover, many relaxed queries do not yield any new answer because they have the same answers as more specific relaxed queries. Huang *et al.* [14] use a similarity score in order to have a better ranking for the evaluation of relaxed queries. They also use a selectivity estimate to save the evaluation of some relaxed queries that are subsumed by a more general relaxed query. Hurtado *et al.* [15] optimize the evaluation of relaxed queries by directly computing their proper answer but the optimization only works for the relaxations that replace a triple pattern by a single other triple pattern. They also introduce new SPARQL clauses, RELAX

and APPROX [9], to restrict relaxation to a small subset of the query. However, this requires from the user to anticipate where relaxation can be useful. The above approaches [14, 15] put some limitations on the relaxation steps that can be applied. Triple patterns can be generalized by relaxation according to RDFS inference but generally can not be removed from the query. URI and literal nodes cannot always be replaced by variables, which may work in some querying use cases but is a dead-end for similarity search. Other approaches [5, 6] present powerful relaxation frameworks but they do not evaluate their efficiency or only generate a few relaxed queries.

Among the numerous similarity/distance measures that have been proposed [4], only a few work on complex relational representations, and can be applied directly to RDF graphs. RIBL (Relational Instance-Based Learning) [13] defines a distance based on the exploration of a graph, starting from a given node and up to a given depth. Ferilli *et al.* [7] define a similarity between Horn clauses, which are analogous to SPARQL queries. However, although the input of those measures are symbolic, their output is numeric. The drawbacks are that they do not provide intelligible explanations for the measured similarity, and also that when two nodes are at the same distance/similarity to a query node, it is not possible to say whether this is by chance or because they share the same explanation. Proposals for a symbolic similarity between graph nodes include Least Common Subsumers (LCS) [3], and intersections of Projected Graph Patterns [8].

3 Preliminary Definitions

For ease of comparison with previous work, we largely reuse the notations in [15]. The only addition is the inclusion of filters in conjunctive queries. We here consider RDF graphs under RDFS inference, although our work can work with no schema at all, and can be extended to other kinds of knowledge graphs, like Conceptual Graphs [1] or Graph-FCA [8]. An *RDF triple* is a triple of *RDF nodes* $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I is the set of IRIs, B is the set of blank nodes, and L is the set of literals. An *RDF graph* G is a set of RDF triples. Its set of nodes is noted $\text{nodes}(G)$. Given an RDF(S) graph G , we note $cl(G)$ the closure of the graph with all triples that can be inferred. A *triple pattern* is a triple $(t_1, t_2, t_3) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$, where V is an infinite set of variables. A *filter* is a Boolean expression on variables and RDF nodes. We here only consider equalities between a variable and a node: $x = n$. A *graph pattern* P is a set of triple patterns and filters, collectively called *query elements*. We note $\text{var}(P)$ the set of variables occurring in P . A *conjunctive query* Q is an expression $X \leftarrow P$, where P is a graph pattern, and $X = (x_1, \dots, x_n)$ is a tuple of variables. X is called the *head*, and P the *body*. We define $\text{var}(Q) = X \cup \text{var}(P)$ the set of variables occurring in query Q . A query is said *connected* if its body is a connected graph pattern, and contains at least one head variable. A *matching* of pattern P on graph G is a mapping $\mu \in \text{var}(P) \rightarrow \text{nodes}(G)$ from pattern variables to graph nodes such that $\mu(t) \in cl(G)$ for every triple pattern $t \in P$, and $\mu(f)$ evaluates to true for every filter $f \in P$, where $\mu(t)$ and $\mu(f)$ are obtained

from t and f by replacing every variable x by $\mu(x)$. The *match-set* $M(P, G)$ of pattern P on graph G is the set of all matchings of P on G . A match-set M can be seen as a relational table with $\text{dom}(M)$ as set of columns. Relational operations such as natural join (\bowtie), selection (σ), and projection (π) can then be applied to match-sets [2]. The *answer set* of query $Q = X \leftarrow P$ on graph G is the match-set of the body projected on head variables: $\text{ans}(Q, G) = \pi_X M(P, G)$.

4 From Query Relaxation to Similarity Search

Query Relaxation. We propose a definition of query relaxation in the line of Hurtado *et al.* [15] but in a slightly more general form to allow for additional kinds of relaxations. We start from a partial ordering \leq_e on query elements that is consistent with logical inference: e.g., $(x, \text{rdf:type}, \text{:SciFiFilm}) \leq_e (x, \text{rdf:type}, \text{:Film})$. From there, we define the relaxation of query elements.

Definition 1 (element relaxation). *The set of relaxed elements of a query element e is the set of immediate \leq_e -successors of e : $\text{relax}(e) = \{e' \mid e \prec_e e'\}$.*

If there is no ontological definition, we have $\text{relax}(e) = \emptyset$ but queries can still be relaxed by removing query elements. In fact, element relaxation is handled as an optional refinement of element removal.

Definition 2 (query relaxation). *Let $Q = X \leftarrow P$ be a (conjunctive) query. Query relaxation defines a partial ordering \leq_Q on queries. A relaxation step, noted $Q \prec_Q Q'$, transforms Q into a relaxed query Q' by replacing any element $e \in P$ by $\text{relax}(e)$: $Q' = X \leftarrow (P \setminus \{e\} \cup \text{relax}(e))$. Partial ordering \leq_Q is defined as the reflexive and transitive closure of \prec_Q . The set of relaxed queries is defined as $RQ(Q) = \{Q' \mid Q <_Q Q'\}$. The relaxation distance of relaxed query $Q' \in RQ(Q)$ is the minimal number of relaxation steps to reach it.*

A novelty compared to previous work is that an element is not replaced by *one of* its immediate successors but by its *set of* immediate successors. When that set is empty, it allows for the removal of a query element. When that set has several elements, it allows for a more fine-grained relaxation. For example, if class `:FullProfessor` is a subclass of `:Researcher` and `:Teacher`, then triple pattern $(x, \text{rdf:type}, \text{:FullProfessor})$ will first be relaxed into pattern $\{(x, \text{rdf:type}, \text{:Researcher}), (x, \text{rdf:type}, \text{:Teacher})\}$, which can be further relaxed into either $\{(x, \text{rdf:type}, \text{:Researcher})\}$ or $\{(x, \text{rdf:type}, \text{:Teacher})\}$.

Another relaxation that is often missing is the replacement of a graph node (URI or literal) by a variable. The difficulty is that a graph node may occur in several query elements, and its relaxation into a variable is therefore not captured by element-wise relaxation. A solution is to normalize the original query in the following way. For each RDF node n occurring as subject/object in the graph pattern, replace its occurrences by a fresh variable x_n , and add the filter $x_n = n$ to the query body. We note $\text{Norm}(P)$ the result of this normalization on pattern P . The node-to-variable relaxation then amounts to the removal of

one query element, filter $x_n = n$. For the sake of simplicity, we do not include here the relaxation of property paths [9]. Another kind of relaxation that we let to future work is the gradual relaxation of equality filters.

We define the *proper answer* of a relaxed query with the same meaning as the “new answers” in [15], and extend this notion to *proper relaxed queries*.

Definition 3 (proper relaxed queries). Let Q be a query, and $Q' \in RQ(Q)$ be a relaxed query, and G be a graph. The proper answers of Q' is the subset of answers of Q' that are not answers of more specific relaxed queries.

$$\text{properAns}(Q', G) := \text{ans}(Q', G) \setminus \bigcup\{\text{ans}(Q'', G) \mid Q'' \in RQ(Q), Q'' < Q'\}$$

A proper relaxed query Q' is a relaxed query whose proper answer is not empty. We note $\text{PRQ}(Q, G)$ the set of proper relaxed queries of Q in G :

$$\text{PRQ}(Q, G) := \{Q' \in RQ(Q) \mid \text{properAns}(Q', G) \neq \emptyset\}.$$

The objective of query relaxation is to compute as efficiently as possible the proper relaxed queries, their proper answers, and their partial ordering.

Similarity Search. As a contribution to previous work, we lay a bridge from query relaxation to similarity search. Whereas Huang *et al.* [14] use a similarity measure to rank the relaxed queries, we propose to use proper relaxed queries to define a *semantic similarity* that is symbolic rather than numeric. Query relaxation is usually presented as a way to go from a query Q to approximate answers. However, it can in principle be applied to go from an RDF node n to *semantically similar nodes*. Simply build the original query $Q(n) = x \leftarrow \text{Norm}(P)$, where x abstracts over node n , and P is the description of node n , up to some depth in the graph, where n is replaced by x . For every $Q' \in \text{PRQ}(Q(n))$, $\text{properAns}(Q')$ is a set of similar nodes, and Q' describes the semantic similarity as a relaxed description. Compared to similarity (or distance) measures, similarity by query relaxation provides articulate and intelligible descriptions of the similarity. It also provides a more subtle ranking with a partial ordering instead of a total ordering, and the definition of clusters of nodes (proper answers) that share the same relaxed description. Furthermore, several numerical measures can be derived from proper relaxed queries, if desired:

- *extensional distance*: the number of answers in $\text{ans}(Q', G)$, i.e. the number of nodes that match what n has in common with nodes in $\text{properAns}(Q', G)$;
- *intensional similarity*: the proportion of elements in $\text{cl}(Q)$ that are in $\text{cl}(Q')$;
- *relaxation distance*: the number of relaxation steps from Q to Q' .

The latter two may be parameterized with weights on query elements [14] and relaxation steps [6], respectively. Note that those derived measures are total orderings that are compatible with partial ordering \leq_Q on relaxed queries. Indeed, a relaxation step can only increase the relaxation distance, and produce a more general query: hence $Q_1 \leq_Q Q_2 \Rightarrow \text{cl}(Q_1) \supseteq \text{cl}(Q_2) \wedge \text{ans}(Q_1, G) \subseteq \text{ans}(Q_2, G)$.

Applying query relaxation to semantic similarity is very challenging to compute. The number of relaxed queries grows exponentially with the size of the original query. The description of nodes in real RDF graphs can easily reach 100 triples at depth 1, and grows exponentially with depth. Previous work on query relaxation has so far restricted its application in order to manage the complexity: (a) queries with a few triple patterns only, (b) a RELAX clause to limit relaxations to one or two chosen triple patterns, and (c) a maximum relaxation distance. Our objective is to lift those restrictions and limits to the point where query relaxation can be applied effectively on large queries like node descriptions. The expected benefits are to allow for parameter-free query relaxation, and for similarity search with explanations. We propose a solution that we demonstrate to be significantly more efficient, and that can behave as an anytime algorithm in the hard cases or when responsiveness is important.

5 Anytime Partitioning of Approximate Answers

We propose a novel approach of query relaxation and symbolic similarity that is based on the iterative partitioning of the set of possible approximate answers. The general idea is that, at any stage of the algorithm, the set of possible answers is partitioned in a set of clusters, where each cluster C is defined by a relaxed query Q' , and the subset of $\text{ans}(Q')$ that are not answers of other clusters. Initially, there is a single cluster defined by the fully relaxed query – the query with an empty body – and the set of all possible answers. Each cluster may be split in two parts by using a query element as discriminating criteria. The algorithm stops when no cluster can be partitioned further. The resulting partition is the set of PRQs along with their (non-empty) proper answer.

Definition 4. A cluster is a structure $C = (X, P', d, E, M, A)$, where:

- X is the head, and P' the body, of the relaxed query $Q'_C = X \leftarrow P'$;
- d is the relaxation distance of Q'_C ;
- E is the set of elements that can be used to split the cluster;
- M is the match-set of P' over graph G ;
- A is a subset of $\text{ans}(Q'_C, G) = \pi_X M$.

A cluster represents a collection of relaxed queries, namely the subset of $RQ(X \leftarrow P' \cup E)$ whose body contains P' . Relaxed query $Q'_C = X \leftarrow P'$ is the most general in the collection, and d is its relaxation distance.

Algorithm 1 details the partitioning algorithm, and Fig. 1 shows its first steps on an example query about *Science-Fiction films directed by Spielberg*. Given a graph and an original query, it normalizes the query body, and initializes the partitioning with a single cluster C_{init} (C0 in Fig. 1) that represents the set of all relaxed queries, and covers all possible answers ($\{A1, A2, A3, A4\}$ in Fig. 1). Every iteration picks a cluster C and an available query element $e \in E$ in order to split C into C_e , and $C_{\bar{e}}$. The picked element must be connected to the current relaxed query $X \leftarrow P'$ in order to avoid building disconnected queries (e.g. e3

Algorithm 1. *Partition(G, Q)*

Require: A graph G , and an original query $Q = X \leftarrow P$

Ensure: A partition \mathcal{C} of approximate answers of Q on G , clustered by PRQ

- 1: $\mathcal{C} \leftarrow \{C_{init} := (X, \emptyset, 0, Norm(P), A_{init}, A_{init})\}$ where $A_{init} = ans(X \leftarrow \emptyset, G)$
- 2: **while** there are a cluster $C = (X, P', d, E, M, A) \in \mathcal{C}$
and an element $e \in E$ connected with $X \leftarrow P'$ **do**
- 3: $M_e \leftarrow ext(M, e, G) := \begin{cases} M \bowtie ans(var(e) \leftarrow \{e\}, G) & \text{if } e \text{ is a triple pattern} \\ \sigma_e M & \text{if } e \text{ is a filter} \end{cases}$
- 4: $A_e = A \cap \pi_X M_e$
- 5: $C_e = (X, P' \cup \{e\}, d, E \setminus \{e\}, M_e, A_e)$
- 6: $C_{\bar{e}} = (X, P', d+1, E \setminus \{e\} \cup relax(e), M, A \setminus A_e)$
- 7: $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C\} \cup \{C_e \mid A_e \neq \emptyset\} \cup \{C_{\bar{e}} \mid (A \setminus A_e) \neq \emptyset\}$
- 8: **end while**

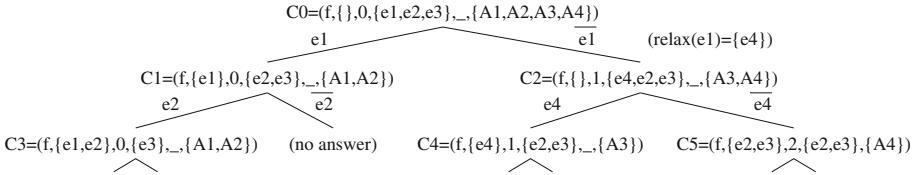


Fig. 1. First steps of the partitioning of query $Q = f \leftarrow \{e1, e2, e3\}$ where $e1 = (f, \text{rdf:type, SciFiFilm})$, $e2 = (f, \text{:director}, d)$, $e3 = (d = \text{:Spielberg})$, and $relax(e1) = \{e4\}$ where $e4 = (f, \text{rdf:type, :Film})$.

is not eligible at C0). C_e selects the subset of answers $A_e \subseteq A$ where e holds. Element e is moved from available elements to the body, thus restricting the collection of relaxed queries to those containing e . Each matching $\mu \in M$ is extended to the new element if possible, and removed otherwise: $ext(M, e, G)$ is defined as a join if e is a triple pattern, and as a selection if e is a filter. The set of answers is restricted to those for which there is an extended matching. $C_{\bar{e}}$ is called the complement of C_e because both the collection of relaxed queries and the set of answers are the complement of the respective parts of C_e . The relaxed query remains the same, and so does the match-set. The relaxation distance is increased by 1, and element e is replaced by its immediate relaxations $relax(e)$, if any ($e1$ is replaced by $e4$ at C2 in Fig. 1), so that they are considered in further partitioning. Clusters C_e and $C_{\bar{e}}$ replace their parent cluster C in the partition \mathcal{C} , unless their answer set is empty (e.g. the right child of C1 in Fig. 1). The number of clusters can only grow, and therefore converges.

Impact of Ontology. Ontological definitions such as `rdfs:subClassOf` axioms allow for more fine-grained relaxations, e.g. replacing query element $e1 = (f, \text{rdf:type, SciFiFilm})$ by $e4 = (f, \text{rdf:type, Film})$, instead of simply removing $e1$. However, the main relaxation work is done by the replacement of URIs and literals by variables, through the removal of filters. For instance, the removal of $e3 = (f = \text{:Spielberg})$ allows to relax the query to *films by any director*. By including the description of Spielberg in the query, and by relaxing that descrip-

tion, the query can be relaxed to *films by similar directors*, e.g. *directors born in the same place and/or having directed similar films*. Note that the efficiency of our algorithms is essential to the effective relaxation of such descriptions. In comparison to description relaxation, ontology-based relaxation has a welcome but limited impact.

Complexity. The enumeration-based algorithm evaluates $O(2^n)$ relaxed queries, given an original query containing n elements, considering element removal only. This number can be lowered by setting a maximal relaxation distance d_{max} , but it grows very quickly with increasing d_{max} . In contrast, assuming p is the number of PRQs, the partitioning algorithm evaluates $O(p)$ queries (binary tree with p leaves). p is bounded by $\min(N, 2^n)$, with N the number of nodes in the graph. As our experiments show (Sect. 7), p is much smaller than N in practice because many nodes match the same relaxed queries. Note that maximal relaxation distance can be applied in the partitioning algorithm by pruning clusters based on their relaxation distance.

Discussion. The partitioning algorithm does not find PRQs in increasing relaxation distance but it offers a lot of flexibility. The clusters can be split in any ordering, allowing for depth-first or breadth-first strategies or the use of heuristics [14]. The choice of the splitting element e is also free, and is independent from one cluster to another. Last but not least, the algorithm is *anytime* because a partition is defined at all time. If the algorithm is stopped before completion, the partition is simply coarser. Section 7.3 shows that the partitioning algorithm has less latency than enumeration-based algorithms for similarity search.

6 Optimization with Lazy Joins

The explicit computation of the match-set M of a relaxed query can be intractable, even in simple cases. For example, starting from the description of a film, the relaxed query $f \leftarrow (f, \text{rdf:type, Film}), (f, \text{:actor}, p_1), \dots, (f, \text{:actor}, p_n)$ will have in $O(Nn^n)$ matchings, assuming there are N films, and each film is related to n actors. For 1000 films related to 10 actors each, it amounts to 10^{13} matchings! It is possible to do better because the expected end result is the set of answers $A = \pi_X M$, whose size is bounded by the number of graph nodes in similarity search. We propose to represent M by a structure – called *match-tree* – containing several small local joins instead of one large global join.

Definition 5. A match-tree is a rooted n -ary tree T where each node n is labeled by a tuple $l(n) = (e, D, M, \Delta)$ where:

- e is a query element, and $\text{var}(e)$ is its set of variables;
- $D \subseteq \text{var}(e)$ is the set of variables introduced by e ;
- M is a match-set s.t. $\text{dom}(M) \supseteq \text{var}(e)$;
- $\Delta \subseteq \text{dom}(M)$ is the sub-domain that is useful to the node ancestors.

Algorithm 2. *LazyJoin*(T, n, n^*)

Require: a match-tree T , a current node n in T labeled with (e, D, M, Δ) , and a new node n^* labeled with $(e^*, D^*, M^*, \Delta^*)$

Ensure: two sets of variables Δ^+, Δ^-

- 1: $\Delta^+ \leftarrow \emptyset; \Delta^- \leftarrow \emptyset$
- 2: **for all** $n_c \in \text{children}(n)$, labeled with $(e_c, D_c, M_c, \Delta_c)$ **do**
- 3: $\Delta_c^+, \Delta_c^- \leftarrow \text{LazyJoin}(T, n_c, n^*)$ // recursive call on each child node
- 4: $\Delta^+ \leftarrow \Delta^+ \cup \Delta_c^+; \Delta^- \leftarrow \Delta^- \cup \Delta_c^-$
- 5: $M \leftarrow M \bowtie \pi_{\Delta_c} M_c$, if Δ_c or M_c was modified // update of local join
- 6: **end for**
- 7: **if** $D \cap \Delta^* \neq \emptyset$ **then** // if this node defines a variable of the new element
- 8: **if** n^* not yet inserted in T **then** // insert new node, unless already inserted
- 9: $\Delta^- \leftarrow \Delta^- \cup (\Delta^* \setminus D); M \leftarrow M \bowtie \pi_{\Delta^*} M^*; \text{parent}(n^*) \leftarrow n$
- 10: **else**
- 11: $\Delta^+ \leftarrow \Delta^+ \cup (\Delta^* \cap D)$
- 12: **end if**
- 13: **end if**
- 14: $\Delta^+, \Delta^- \leftarrow \Delta^+ \setminus \Delta^-, \Delta^- \setminus \Delta^+$
- 15: $\Delta \leftarrow \Delta \cup \Delta^+ \cup \Delta^-$ // update Δ
- 16: **return** Δ^+, Δ^-

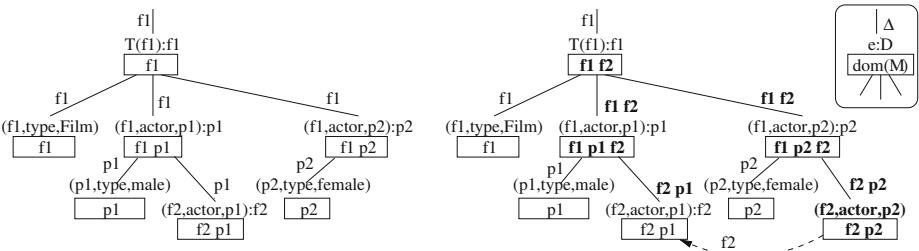


Fig. 2. Match-tree before and after lazy join with element $e^* = (f_2, \text{:actor}, p_2)$.

We replace the match-set M of a cluster $C = (X, P', d, E, M, A)$ by an equivalent match-tree T . The match-tree has one node for each element $e \in P'$. M is equal to the join of the local match-set M_n of all nodes n in T . The *initial match-tree* is used for the initial cluster C_{init} , and is defined by match-tree T_{init} that has a single node labeled with $(\top(X), X, ans(X \leftarrow \emptyset, G), X)$, where $\top(X)$ is a void query element. During the split of C with a query element e^* , the match-set extension $ext(M, e^*, G)$ is replaced by $\text{LazyJoin}(T, \text{root}(T), n^*)$ (see Algorithm 2), where T plays the role of M , and n^* is a new node labeled with $(e^*, var(e^*) \setminus var(P'), ans(var(e^*) \leftarrow \{e^*\}, G), var(e^*) \cap var(P'))$. This “lazy” join consists in inserting n^* as a leaf node into T , and in updating local joins M_n only as much as necessary to compute A correctly. Figure 2 illustrates lazy join on a richer version of the above example on films. It starts with the match-tree of query $f_1 \leftarrow (f_1, \text{rdf:type, Film}), (f_1, \text{:actor}, p_1), (p_1, \text{:sex, male}), (f_2, \text{:actor}, p_1), (f_1, \text{:actor}, p_2), (p_2, \text{:sex, female})$, and inserts element $e^* = (f_2, \text{:actor}, p_2)$ that

forms a cycle through f_1 and p_1 . Changes (shown in bold) are propagated from the two nodes that define p_2 and f_2 (see D), and the new leaf is inserted under one of the two nodes as a child (here, under the node defining p_2). The insertions of other elements, which led to the first tree in Fig. 2, change only one path in the match-tree because they do not introduce a cycle. In Algorithm 2, the computation of Δ^+ , Δ^- is used to correctly handle cycles. They are sets of variables of the new query element e^* that are not in the match-set of its parent (f_2 in the example), and hence need to be joined with distant nodes in the match-tree. Δ^- propagates up the tree branch of the parent (node (f_1, actor, p_2) in Fig. 2), and Δ^+ propagates up the tree branch of distant nodes (node (f_2, actor, p_1)). When they meet at their common ancestor (node $\top(f_1)$), the distant join can be done, and their propagation stops.

Complexity. The dominant factor is the project-and-join operation in lines 5, 9 of Algorithm 2. In the worst case, there is such an operation for each node of the match-tree, i.e. one for each element of the relaxed query (m). The cost of the project-and-join depends heavily on the topology of the relaxed query and of the RDF graph. Assuming a tree-shape graph pattern (no cycle) and R triples per predicate, the match-set at each node is a table with at most 2 columns and at most R rows. The space complexity of a match-tree is therefore in $O(mR)$, and the time complexity of each lazy join is in $O(mR\log R)$. In the simpler example on films, $m = n + 1$, $R = Nn$ for property `:actor`, and $R = N$ for class `:Film`. The total size is therefore in $O(Nn^2)$ local matchings instead of $O(Nn^n)$. For 1000 films related to 10 actors each, it amounts to 10^5 instead of 10^{13} !

7 Experiments

We have implemented Algorithms 1 and 2, and integrated them to SEWELIS¹ as an improvement of previous work on the guided edition of RDF graphs [12]. We here report extensive experiments to evaluate their efficiency, in the absolute and relative to other approaches, as well as their effectiveness.

7.1 Methodology

We ran all experiments on a Fedora 25 PC with i7-6600U CPU and 16 GB RAM.

Algorithms. We compare four algorithms: two baseline algorithms, and two variants of our algorithm. RELAXENUM is the classical approach that enumerates relaxed queries, and computes their answers. NODEENUM does the opposite by enumerating nodes, and computing the associated PRQ as the least common subsumer between the query and the node description [3]. PARTITION and PARTITIONLJ are our two variants, and differ in that only the latter uses lazy joins.

¹ Source code at <https://bitbucket.org/sebferre/sewelis>, branch `dev`, files `cnn*`.

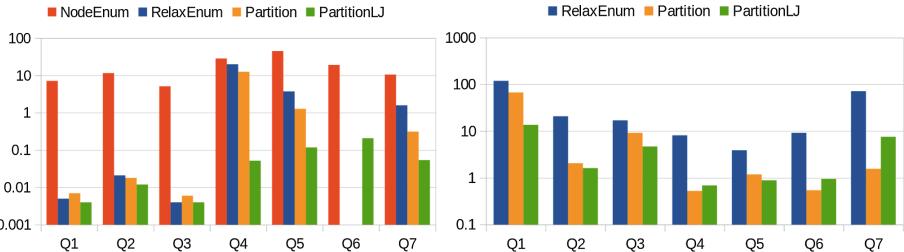


Fig. 3. Runtime (seconds, log scale) per algorithm and per query for MONDIAL (left) and LUBM10 (right).

Parameters. We consider three execution modes. In the NoLIMIT mode, all PRQs are computed. It is the best way to compare algorithms with different parameters. The MAXRELAX mode sets a limit to relaxation distance (not applicable to NODEENUM). The TIMEOUT mode sets a limit to computation time.

Datasets. We use three different datasets, MONDIAL, LUBM10, and LUBM100. MONDIAL is a simplification of the MONDIAL dataset [16] where numbers and reified nodes have been removed because they are not yet well handled by query relaxation. It contains a rich graph of real geographical entities (e.g., *countries containing mountains and rivers that flow into lakes*), and no ontological definitions. It contains 10k nodes and 12k triples. LUBM10 and LUBM100 are synthetic datasets about universities, and are based on an ontology introducing class and property hierarchies [11]. LUBM10 contains 315k nodes and 1.3M triples, and LUBM100 is 10 times bigger (3M nodes, 13M triples). Both datasets are interesting in our work because they are rich in multi-valued properties, and hence are a challenge for query relaxation.

Queries. For each dataset, we have two sets of queries. The first set is made of queries typically used in query relaxation, having less than 10 triple patterns and having different shapes. The second set is made of object descriptions, having up to hundreds of elements. More details are given below.

7.2 Efficiency of Query Relaxation

We here compare all algorithms with (small) queries². On MONDIAL we defined 7 queries having different sizes and shapes. They have between 5 and 21 elements after normalization. On LUBM10/100 we reused the 7 queries of Huang *et al.* [14]. They have between 3 and 7 elements.

NoLimit Mode. Figure 3 compares the runtime (log scale) of all algorithms on all queries of MONDIAL and LUBM10 in NoLIMIT mode. A few runtimes are missing on Q6 of MONDIAL because they are much higher than other runtimes. RELAXENUM is sensitive to the query complexity, in particular to multi-valued

² Visit <http://www.irisa.fr/LIS/ferre/pub/eswc2018/queries.txt>.

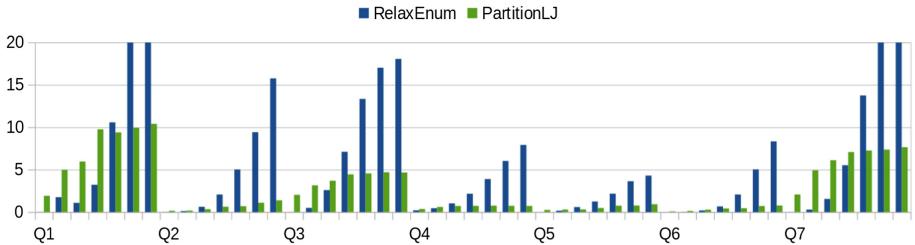


Fig. 4. Runtime (seconds) per algorithm and per LUBM10 query for increasing maximum relaxation distance (1 to 7). Full height bars represent runtimes over 20 s.

properties. NODEENUM looks unsensitive to the query complexity but does not scale well with the number of nodes. PARTITION and PARTITIONLJ are always more efficient – or equally efficient – and the use of lazy joins generally makes it even more efficient. On LUBM10, PARTITIONLJ is typically one order of magnitude more efficient than RELAXENUM. It can also be observed that PARTITIONLJ scales well with data size because from MONDIAL to LUBM10, a 100-fold increase in number of triples, the median runtime also follows a 100-fold increase, from 0.01–0.1 s to 1–10 s. This linear scaling is verified on LUBM100 (not shown) where the runtimes are all 10 times higher. We want to emphasize that it is encouraging that the full relaxation of a query over a 1.3M-triples dataset can be done in 4 s on average.

MaxRelax Mode. In practice, one is generally interested in the most similar approximate answers, and therefore in the relaxed queries with the smaller relaxation distances. It is therefore interesting to compare algorithms when the relaxation distance is bounded. Figure 4 compares RELAXENUM and PARTITIONLJ on LUBM10 queries for increasing values of maximal relaxation distance, here from 1 to 7 relaxations. As expected, the runtime of RELAXENUM grows in a combinatorial way, like the number of relaxed queries, with the relaxation distance. On the contrary, the runtime of PARTITIONLJ grows more quickly for 1–4 relaxations, and then almost flattens in most cases. The flattening can be explained by several factors. The main factor is that the relaxed queries that are not proper are pruned. Another factor is that query evaluation is partially shared between different queries. That sharing is also the cause for the higher cost with 1–4 relaxations. In summary, PARTITIONLJ scales very well with the number of relaxations, while RELAXENUM does not. This is a crucial property for similarity search where many relaxation steps are necessary.

Number of Relaxed Queries. The efficiency of PARTITION and PARTITIONLJ is better understood when comparing the number of PRQs to the number of relaxed queries (RQ). Over the 7 LUBM10 queries, there are in total 59 PRQs out of 2899 RQs, hence a 50-fold decrease.

7.3 Efficiency of Similarity Search

We now consider the large queries that are formed by node descriptions at depth 1 (including ingoing triples), and that include many multi-valued properties. On MONDIAL we used 10 nodes, one for each class: e.g., country *Peru*, language *Spanish*. The obtained queries have 5 to 248 elements (average = 61). On LUBM10 we used 12 nodes, one for each class: e.g., university, publication. The obtained queries have 5 to 1505 elements (average = 201). Main facts and measures are summarized in Table 1.

NoLimit Mode. PARTITIONLJ is the only algorithm that terminates under 600 s in NoLIMIT mode, and it does terminate for all queries of both datasets³. The full runtime has a high variance and a low median, between 0.03 s and 269 s (average = 30 s, median = 0.13 s) for MONDIAL, and between 0.13 s and 535 s (average = 179 s, median = 110 s) for LUBM10. The average number of PRQs is 32 (max = 117) for MONDIAL and 28 (max = 52) for LUBM10. It is noteworthy that both the runtime and the number of PRQs are relatively close for different datasets with very different sizes. It is also noteworthy that the number of PRQs is very small w.r.t. the size of queries and the number of nodes. It is therefore possible to manually inspect all PRQs, i.e. semantic similarities with all other nodes.

MaxRelax Mode. On LUBM10, RELAXENUM does not scale beyond 3 relaxations on node descriptions, except for the 3 smallest descriptions. Moreover, using PARTITIONLJ with a maximum of 10 relaxations, we have observed that only 75 PRQs are produced out of 340 over the 12 queries (0 PRQs for 3 queries), and computation takes 40 s on average. To limit the average runtime to 2 s, the maximum relaxation distance has to be 1, and then only 3 PRQs are produced out of 340. RELAXENUM and the MAXRELAX mode are therefore not effective for similarity search because of their high latency in generating the first PRQs.

TimeOut Mode. Although PARTITIONLJ can compute all PRQs in a reasonable time, it may still be too long in practice. Our anytime algorithm allows to control runtime by timeout. An important question is the latency of the generation of PRQs. Figure 5 shows that most PRQs are generated in a small fraction of the total runtime. For example, 50% PRQs are generated in 30 s, and 20% in 2 s. This superlinear production of PRQs is confirmed when looking at what happens when moving from LUBM10 to LUBM100. At timeout = 64 s, the number of computed PRQs is only divided by 2.6 although the data is 10 times bigger.

7.4 Effectiveness of Similarity Search

We assess the effectiveness of PRQs for similarity search with an empirical study that evaluates their use for the inference of property values. The null hypothesis

³ The ranked list of PRQs and their proper answers is available for the 10 MONDIAL nodes at http://www.irisa.fr/LIS/ferre/pub/eswc2018/mondial_PRQs.txt.

Table 1. Measures about the relaxation of node descriptions. The measures are given in the form *min - max (average)*: number of query elements, number of PRQs, number of PRQs within maximum 3 relaxation steps, and runtime of PARTITIONLJ.

Dataset	MONDIAL	LUBM10
nb. elts	5–248 (61)	5–1505 (201)
nb. PRQs	2–117 (32)	7–52 (28)
nb. PRQs (1–3 steps)	0–9 (3.2)	0–7 (1.3)
Runtime	0.03–269 (30)	0.13–535 (179)

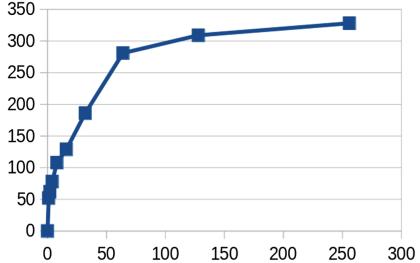


Fig. 5. Number of computed PRQs over lubm10 queries (out of 340 PRQs) as a function of timeout(s).

is that using the most similar nodes makes no difference compared to using random nodes. We do not claim here that this value inference method is better than other methods in machine learning, only that PRQs effectively capture semantic similarities. We do not compare our results to other query relaxation algorithms because we have shown in the previous section that they are not efficient enough for similarity search, and because, if given enough time, they would compute the same PRQs. Our experiment consisted in performing value inference for each property p (e.g., `:continent`) of each country c (e.g., *Peru*) of the MONDIAL dataset. Note that a property may have several values (e.g., `:language`). For each pair (c, p) we computed the PRQs starting with the description of c minus triples (c, p, v) for any value v . Then, we selected the 3 most similar nodes S (e.g., *Bolivia*, *Colombia*, and *Argentina* for *Peru*) as the proper answers of the most similar PRQs according to intensional similarity (size of the relaxed query). We used S to infer a set of values V for property p using a majority vote. V is compared to the actual set of values to determine precision and recall measures. Table 2 shows the obtained measures for each property and for all properties together, averaged over the countries. The baseline F1-score is obtained by using 3 random countries instead of the most similar nodes. It shows a significant improvement over the baseline, and good results for properties *continent* and *religion*. The result is poor for *government* because values are noisy strings, and for *dependentOf* because few countries have this property. Overall, 75% of inferred values are correct, and 48% of correct values are inferred. We obtained very close results when using extensional distance (number of answers) for ranking PRQs.

Table 2. Average precision/recall/F1-score over countries of value inference for each property and for all properties together. The last line gives the baseline F1-score.

	Continent	Religion	was Dependent Of	Language	Neighbor	Ethnic Group	Government	Dependent Of	All
Precision	0.97	0.80	0.77	0.75	0.43	0.65	0.82	0.50	0.75
Recall	0.93	0.76	0.47	0.34	0.33	0.25	0.19	0.00	0.48
F1-score	0.93	0.74	0.47	0.36	0.30	0.28	0.19	0.00	0.56
Baseline F1-score	0.12	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.05

8 Conclusion and Perspectives

We have proposed two algorithms that together significantly improve the efficiency of query relaxation and symbolic semantic similarity. In fact, they are efficient enough to tackle the challenging problem of applying many relaxation steps to large queries with hundreds of elements. For a 1M-triples dataset like LUBM10 it is possible to explore the entire relaxation space of queries having up to 1500 elements in a matter of minutes. The computation can be stopped at any time while still having a fair approximation of the final result.

Many perspectives can be drawn from this work. The partitioning algorithm offers opportunities for strategies and heuristics for choosing the cluster to split and the splitting element. New kinds of element relaxation can be explored, e.g. for the gradual relaxation of URIs and literals. There are potential applications in classification, case-based reasoning, concept discovery, or recommendation.

References

- Chein, M., Mugnier, M.L.: Graph-Based Knowledge Representation: Computational Foundations of Conceptual Graphs. Advanced Information and Knowledge Processing. Springer, Heidelberg (2008). <https://doi.org/10.1007/978-1-84800-286-9>
- Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
- Colucci, S., Donini, F.M., Di Sciascio, E.: Common subsumers in RDF. In: Baldoni, M., Baroglio, C., Boella, G., Micalizio, R. (eds.) AI*IA 2013. LNCS (LNAI), vol. 8249, pp. 348–359. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03524-6_30
- Cunningham, P.: A taxonomy of similarity mechanisms for case-based reasoning. IEEE Trans. Knowl. Data Eng. **21**(11), 1532–1543 (2009)
- Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing RDF queries based on user and domain preferences. J. Intell. Inf. Syst. **33**(3), 239 (2009)
- Elbassuoni, S., Ramanath, M., Weikum, G.: Query relaxation for entity-relationship search. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011. LNCS, vol. 6644, pp. 62–76. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21064-8_5

7. Ferilli, S., Basile, T.M.A., Biba, M., Di Mauro, N., Esposito, F.: A general similarity framework for horn clause logic. *Fundamenta Informaticae* **90**(1–2), 43–66 (2009)
8. Ferré, S., Cellier, P.: Graph-FCA in practice. In: Haemmerlé, O., Stapleton, G., Faron Zucker, C. (eds.) ICCS 2016. LNCS (LNAI), vol. 9717, pp. 107–121. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40985-6_9
9. Frosini, R., Calì, A., Poulovassilis, A., Wood, P.T.: Flexible query processing for SPARQL. *Semant. Web* **8**(4), 533–563 (2017)
10. Gaasterland, T.: Cooperative answering through controlled query relaxation. *IEEE Expert* **12**(5), 48–59 (1997)
11. Guo, Y., Pan, Z., Hefflin, J.: LUBM: a benchmark for OWL knowledge base systems. *J. Web Semant.: Sci. Serv. Agents* **3**, 158–182 (2005)
12. Hermann, A., Ferré, S., Ducassé, M.: An interactive guidance process supporting consistent updates of RDFS graphs. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS (LNAI), vol. 7603, pp. 185–199. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33876-2_18
13. Horváth, T., Wrobel, S., Bohnebeck, U.: Relational instance-based learning with lists and terms. *Mach. Learn.* **43**(1–2), 53–80 (2001)
14. Huang, H., Liu, C., Zhou, X.: Approximating query answering on RDF databases. *World Wide Web* **15**(1), 89–114 (2012)
15. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query relaxation in RDF. In: Spaccapietra, S. (ed.) *Journal on Data Semantics X*. LNCS, vol. 4900, pp. 31–61. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77688-8_2
16. May, W.: Information extraction and integration with FLORID: the MONDIAL case study. Technical report 131, Universität Freiburg, Institut für Informatik (1999). <http://dbis.informatik.uni-goettingen.de/Mondial>
17. Rodríguez, M.A., Egenhofer, M.J.: Determining semantic similarity among entity classes from different ontologies. *IEEE Trans. Knowl. Data Eng.* **15**(2), 442–456 (2003)