Query-Based Entity Comparison in Knowledge Graphs Revisited

Alina Petrova, Egor V. Kostylev, Bernardo Cuenca Grau, and Ian Horrocks

Department of Computer Science, University of Oxford
{alina.petrova, egor.kostylev, bernardo.cuenca.grau, ian.horrocks}@cs.ox.ac.uk

Abstract. Large-scale knowledge graphs are increasingly being used in applications, and there is a growing need for tools that can effectively support users in analysis and exploration tasks. One such important task is entity comparison—to describe in an informative way the similarities between two given entities as described in a knowledge graph. In our previous work the result of entity comparison is modelled as a similarity query—that is, a SPARQL query having the input entities as part of the answer over the input graph; for instance, one can describe the similarity between two companies such as Telenor and Vodafone in the YAGO graph as a query asking for all telecom companies based in Europe. In this paper, we extend the results of our prior work in different ways. First, we expand the language of similarity queries to consider a richer fragment of SPARQL allowing for numeric filter expressions; this enables us to express that Telenor and Vodafone are also similar in that they both have at least 30,000 employees. We then propose algorithms for computing similarity queries satisfying certain additional desirable properties, such as being as specific as possible. Such algorithms are, however, impractical; hence, we also propose and implement a scalable algorithm that is guaranteed to compute a similarity query, but not necessarily a most specific one.

1 Introduction

Large-scale knowledge graphs are increasingly being used in applications, and there is a growing need for tools that can effectively support users in analysis and exploration tasks. One such important task is *entity comparison*—to describe in an informative way the similarities and differences between two given entities as outlined in a knowledge graph. This is in stark contrast to the computation of a similarity measure, where the output is a number indicating how similar the given entities are likely to be rather than a human-readable explanation.

To make our discussion concrete, consider a small excerpt from the YAGO knowledge graph [19] (in RDF format) about European companies that is depicted in Figure 1. We would like a tool to assist us in comparing Telenor with

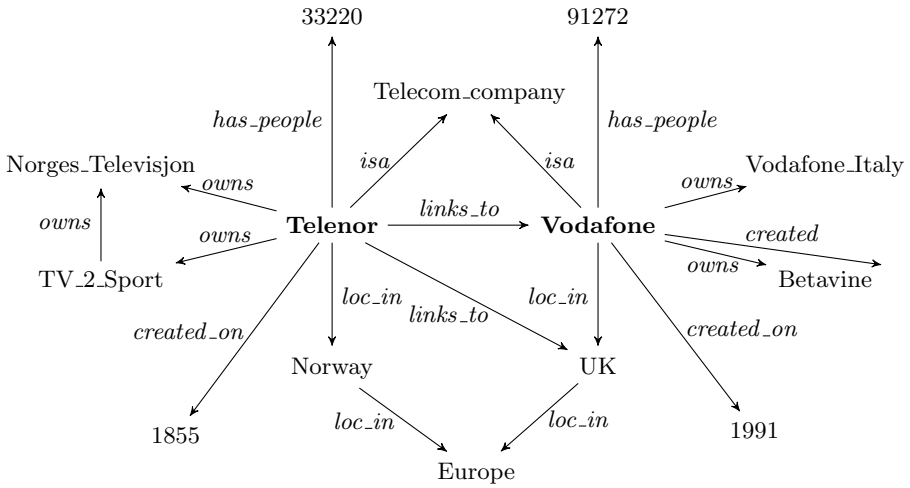

Fig. 1: An excerpt of the YAGO knowledge graph

Vodafone. In particular, the tool should be able to automatically report that Telenor and Vodafone are similar in that they are both telecom companies located in Europe which own other companies, have at least 25 years of operating experience and more than 30,000 employees on payroll; however, they are different in that Telenor is located in Norway whereas Vodafone is based in the UK.

Entity comparison is used routinely across multiple domains and applications, from online shopping to food and nutrition comparison widgets, to Facebook's 'see what you have in common' pages. Existing tools typically focus on a constrained application domain (e.g., used cars) and provide a side-by-side comparison of the given entities based on a fixed set of relevant attributes (e.g., price, engine size, or colour). We are, however, interested in the generic entity comparison support in knowledge graphs, in which case it is no longer possible to fix a relevant set of attributes or relationships upfront.

In our previous work, we proposed a logical framework for entity comparison in knowledge graphs represented in RDF format [17]. The description of similarities and differences is given in terms of SPARQL queries in the conjunctive fragment. In particular, a *similarity query* is a query containing the two given entities to compare as answers. A more specific such query is seen as more informative: for example, knowing that both Vodafone and Telenor are telecom companies is more informative than just knowing that they are both companies.

We previously showed that, for any given RDF graph and pair of entities to compare, there exists a unique most specific similarity query (MSSQ), which can be computed in polynomial time in the size of the input graph [17]. The algorithm in that work, however, has two important practical limitations. First, it was designed for a fragment of SPARQL without numeric filter expressions, which significantly limits its applicability to graphs containing numeric information;

for instance, the algorithm would not be able to report as a similarity that both Vodafone and Telenor have at least 30,000 employees. Second, the running time of the algorithm is quadratic in the size of the input graph (even in the best case), which makes it impractical even for moderately-sized inputs.

In this paper, we first extend the previously proposed framework and algorithms so as to produce more informative similarity queries. In particular, we consider a richer fragment of SPARQL allowing for numeric filter expressions, and also study a new type of similarity queries that we call *exact*. We then show that both most-specific and exact similarity queries can be computed using an extension of the algorithm proposed in [17]; this algorithm is, however, also impractical. To address this issue, we then propose a practical and scalable algorithm for computing similarity queries. Although our algorithm does not ensure that the computed similarity query is the most specific one, our empirical evaluation suggests that it is a reasonable approximation in many cases.

2 Preliminaries

Let $\mathbf{U}$, $\mathbf{L}$, and $\mathbf{B}$ be pairwise disjoint, countably infinite sets of *IRIs*, *literals*, and *blank nodes*, respectively. We assume that $\mathbf{L}$ includes all integers $\mathbb{Z}$. We will refer to IRIs and literals collectively as *entities*. An *RDF triple* (or simply a *triple*) is a tuple (s, p, o) from $(\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L} \cup \mathbf{B})$, where s is called the *subject*, p the *predicate*, and o the *object*. An (*RDF*) *graph* is a finite set of triples.

Let $\mathbf{X}$ be a countable infinite set of variables disjoint from $\mathbf{U}$, $\mathbf{B}$, and $\mathbf{L}$. A *term* is an element from $\mathbf{U} \cup \mathbf{L} \cup \mathbf{X}$. A *triple pattern* is a triple of terms from the set $(\mathbf{U} \cup \mathbf{X}) \times (\mathbf{U} \cup \mathbf{X}) \times (\mathbf{U} \cup \mathbf{L} \cup \mathbf{X})$. A *basic graph pattern* is a non-empty finite set of triple patterns. An *arithmetic comparison* is an expression of the form $(?Y \lhd n)$, where $?Y$ is a variable in $\mathbf{X}$, n is an integer (i.e., a literal), and $\lhd$ is a comparison symbol in $\{<, \leq, >, \geq\}$. A (*arithmetic*) *filter condition* is a finite (possibly empty) set of arithmetic comparisons. For E an expression such as a pattern or a filter condition we denote with $\mathsf{var}(E)$ and $\mathsf{term}(E)$ the sets of variables and terms, respectively, occurring in E. A basic graph pattern P is *connected* if for every pair $t, t' \in \mathsf{term}(P)$ there is a sequence of triple patterns $T_1, \ldots, T_m$ in P such that $t \in \mathsf{term}(T_1)$, $t' \in \mathsf{term}(T_m)$ and $\mathsf{term}(T_i) \cap \mathsf{term}(T_{i+1}) \neq \emptyset$ for all $i = 1, \ldots, m-1$.

In this paper, we concentrate on (SPARQL) queries of a very specific form. In particular, in the context of this paper, a *query* is an expression of the form

$$\textsc{Select}\ ?X\ \ \textsc{Where}\ P\ \ \textsc{Filter}\ C, \tag{1}$$

where P is a connected basic graph pattern, $?X \in \mathsf{var}(P)$ is the *answer variable* of the query and C is a filter condition satisfying $\mathsf{var}(C) \subseteq \mathsf{var}(P)$. Such queries essentially correspond to connected monadic conjunctive queries with arithmetic comparisons (CQACs) [15] restricted to signatures over a single ternary relation and using no comparisons between variables.

A *valuation* of a finite set of variables $?\bar{X}$ from $\mathbf{X}$ is a mapping from $?\bar{X}$ to $\mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$. An element from $\mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$ is an *answer* to a query Q of the form (1) over a graph G if there exists a valuation ν of $\mathsf{var}(P)$ so that $\nu(P) \subseteq G$ and $\nu(?Y) \lhd n$

holds for each comparison $(?Y \lhd n)$ in C. We denote by $[Q]_G$ the set of all answers to Q over G. A query Q_1 is *subsumed* by a query Q_2, written $Q_1 \subseteq Q_2$, if $[Q_1]_G \subseteq [Q_2]_G$ for every graph G. Query Q_1 is *strictly subsumed* by query Q_2, denoted by $Q_1 \subset Q_2$, if $Q_1 \subseteq Q_2$ and $Q_2 \not\subseteq Q_1$. Finally, Q_1 and Q_2 are *equivalent*, denoted by $Q_1 \equiv Q_2$, if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. Subsumption and equivalence allow us to compare queries relative to their specificity, so we sometimes say that Q_1 is *(strictly) more specific* than Q_2 if Q_1 is (strictly) subsumed by Q_2.

We conclude this section with an observation that we restrict the filter conditions to only arithmetic comparisons between variables and constants. This is justified by the fact that all other comparisons, such as general inequalities between variables and IRIs, have very little meaning in the context of entity comparisons, and moreover may flood similarity queries hiding the essential parts.

3 Entity Comparison Using Similarity Queries

There are two main proposals for capturing similarities between entities in the literature: either by queries [17] that have given entities as answers or by explicit paths in the graph originating in given entities and converging into the same node [6,11,14]. As discussed in our previous work [17], queries contain variables, which allow us to represent similarities at a higher level of abstraction, so we adopt the first approach. We start by extending the notions of similarity and most specific similarity queries of [17] to also consider filter conditions.

Definition 1. *A query Q is a* similarity query *for entities a and b in a graph G if $\{a, b\} \subseteq [Q]_G$. A similarity query Q for a and b in G is* most specific *(*MSSQ*) for a and b if there is no similarity query Q' for a and b in G such that $Q' \subset Q$.*

For example, the following query Q_{ex} asking for all telecom companies located in Europe is a similarity query for Vodafone and Telenor in the graph in Figure 1:

$$\text{SELECT } ?X \text{ WHERE } \{(?X, \mathit{isa}, \mathit{Telecom\_company}), (?X, \mathit{loc\_in}, ?Y),\ (?Y, \mathit{loc\_in}, \mathit{Europe}))\}.$$

Query Q_{ex} is, however, not an MSSQ since the following query Q'_{ex} is also a similarity query, and it is strictly more specific as it adds the information that both companies were created between 1855 and 1991:

$$\text{SELECT } ?X \text{ WHERE } \{(?X, \mathit{isa}, \mathit{Telecom\_company}), (?X, \mathit{loc\_in}, ?Y),\ (?Y, \mathit{loc\_in}, \mathit{Europe}), (?X, \mathit{created\_on}, ?Z)\} \text{ FILTER } \{(?Z \leq 1991),\ (?Z \geq 1855)\}.$$

It is not difficult to see that a similarity query exists, provided the input entities appear at the same position (i.e., subject, predicate, or object) in the input graph. Moreover, as the above example suggests, there may be multiple (even infinitely many) similarity queries for a pair of entities in a graph. We next show, however, that MSSQs are unique modulo equivalence. Intuitively, this is the case because the conjunction of similarity queries is also a similarity query.

Proposition 1. *MSSQs are unique up to equivalence.*

Proof. Let a and b be entities in a graph G. Consider two arbitrary MSSQs Q_i = SELECT $?X$ WHERE P_i FILTER C_i, $i \in \{1, 2\}$, for a and b in G. Then query

$$Q \quad = \quad \text{SELECT } ?X \ \text{ WHERE } P_1 \cup P_2 \ \text{ FILTER } C_1 \cup C_2$$

is a similarity query, which is also more specific than both Q_1 and Q_2. Note that $P_1 \cup P_2$ is connected because P_1 and P_2 are both connected and both mention $?X$. Therefore, Q_1, Q_2, and Q are all equivalent MSSQs. □

The notion of MSSQ relies on query subsumption, which is a data-independent relationship between queries. It would clearly also make sense to look for similarity queries that are as discriminating for input entities a and b as possible over the specific input graph G at hand—that is, those similarity queries that return only a and b as answers when evaluated over G.

Definition 2. *A query Q is an* exact similarity query (ESQ) *for entities a and b in a graph G if $\{a, b\} = [Q]_G$.*

For instance, our example query Q_{ex} is an ESQ for the example graph from Figure 1 because Vodafone and Telenor are the only telecom companies in Europe represented in the graph. However, as already discussed, Q_{ex} is not an MSSQ because it is not minimal with respect to subsumption. Furthermore, if we were to consider the whole of YAGO instead of our example excerpt, Q_{ex} would certainly no longer be an ESQ since YAGO contains many other European telecom companies. So, MSSQs and ESQs are incomparable in general. The following proposition, however, establishes a useful link between ESQs and MSSQs, which we exploit in the algorithms proposed in following sections.

Proposition 2. *If Q is an MSSQ for entities a and b in a graph G such that $[Q]_G \neq \{a, b\}$, then no ESQ for a and b in G exists.*

Proof. Let Q' be an ESQ for a and b in G—that is, Q' is a similarity query with $[Q']_G = \{a, b\}$. So, Q' is a similarity query that is not subsumed by the MSSQ Q, which contradicts Proposition 1. □

4 Computing Most Specific and Exact Similarity Queries

In this section, we present an algorithm that computes an MSSQ, if one exists, and reports failure otherwise. We also show how a simple modification of this algorithm can be used for computing an ESQ. Our algorithm for MSSQ extends the one in [17], where changes are needed to deal with filter conditions.

Our algorithm relies on the following notion of the (tensor) product graph.

Definition 3. *Given triples $\tau_1 = (s_1, p_1, o_1)$ and $\tau_2 = (s_2, p_2, o_2)$, let*

$$\tau_1 \times \tau_2 = (\langle s_1, s_2 \rangle, \langle p_1, p_2 \rangle, \langle o_1, o_2 \rangle).$$

The product $G_1 \times G_2$ *of graphs G_1 and G_2 is the set $\{\tau_1 \times \tau_2 \mid \tau_1 \in G_1, \tau_2 \in G_2\}$.*

Algorithm 1: COMPUTE_MSSQ

Input: graph G, entities a and b in G
Output: MSSQ for a and b in G, or *fail*

1 compute $G \times G$;
2 **if** $\langle a, b\rangle$ does not occur in a triple in $G \times G$ **then return** *fail*;
3 compute the connected component $G_\times$ of $\langle a, b\rangle$ in $G \times G$;
4 **let** P be the pattern obtained from $G_\times$ by replacing each pair $\langle c_1, c_2\rangle$ with either variable $?X_{c_1,c_2}$, if $c_1 \neq c_2$ or $c_1 \in \mathbf{B}$, or with c_1 otherwise;
5 **if** $a = b$ **then**
6 add to P all triple patterns obtained from triple patterns already in P by replacing at least one occurrence of a with $?X_{a,a}$;
7 **let** C be $\{(?X_{n_1,n_2} \leq \max(n_1, n_2)), (?X_{n_1,n_2} \geq \min(n_1, n_2)) \mid ?X_{n_1,n_2} \in \mathsf{var}(P);\ n_1, n_2 \in \mathbb{Z}\}$;
8 **return** SELECT $?X_{a,b}$ WHERE P FILTER C.

Algorithm COMPUTE_MSSQ (given in Algorithm 1) accepts as input a graph G, and entities a and b in G. In the first step, it computes the product graph $G \times G$ and checks whether the node $\langle a, b\rangle$ occurs in $G \times G$; if it does not, then the algorithm determines that a similarity query (and hence an MSSQ) for a and b in G does not exist, and reports failure. In contrast, if $\langle a, b\rangle$ occurs in the product graph $G \times G$, then the algorithm computes the connected component of $\langle a, b\rangle$ in the product graph and constructs the output query based on it. Specifically, the algorithm computes the pattern P in the query by replacing each element of a product triple in $G \times G$ with either a constant or a variable (uniquely associated with the element), and the filter condition C by adding suitable inequalities for those variables representing pairs of numeric literals in the product graph.

Since the size of $G \times G$ is quadratic in the size of G, the algorithm works in polynomial time. Correctness is established by the following theorem.

Theorem 1. COMPUTE_MSSQ *is a polynomial time procedure that returns an MSSQ for its input entities and graph, if it exists, or fail otherwise.*

Proof. First, recall that a similarity query of entities a and b in a graph G exists if and only if both a and b appear in the same position in triples in G, which happens precisely when $\langle a, b\rangle$ appears in a triple in $G \times G$ by construction. So, if COMPUTE_MSSQ returns *fail* in line 2 then there is no MSSQ for a and b.

Next, algorithm COMPUTE_MSSQ extends our previously proposed algorithm from [17] that computes a most specific similarity query without filter conditions for two entities in an RDF graph. So, if the MSSQ (in the extended language) does not contain integers, then COMPUTE_MSSQ returns this MSSQ, with the filter condition C being empty. If the MSSQ contains integers, then the algorithm first generates the most specific basic graph pattern P in lines 1–6 as before and then computes the filter condition C in line 7; moreover, C contains the arithmetic comparisons for all possible numeric variables in P, and these comparisons are constrained in the tightest way possible by the integer values.

Finally, as already mentioned, all steps can be done in polynomial time. □

The correctness of the algorithm implies that an MSSQ is always guaranteed to exist whenever a similarity query exists for the given input. Furthermore, checking whether a similarity query exists can be done efficiently.

Despite running in polynomial time, Algorithm 1 is impractical. Indeed, real-life graphs G of interest tend to contain millions of triples, and the algorithm explicitly constructs the product graph $G \times G$, which is of quadratic size in the size of G. Moreover, large MSSQs are often incomprehensible and practically useless for entity comparison. Hence, it makes sense to design *approximation algorithms*, which, on the one hand, construct reasonably specific similarity queries and, on the other hand, can scale to large input graphs. In Section 5 we devise one such algorithm. We next show, however, that checking whether a query (e.g., a query output by an approximation algorithm) is an MSSQ is computationally hard.

Theorem 2. *The problem of checking whether a query is an MSSQ for two entities in a graph is* Π_2^P*-complete.*

Proof (Sketch). To check whether a query Q is an MSSQ for entities a and b in a graph G, we proceed as follows. First, we apply Algorithm 1 to obtain (in polynomial time) an MSSQ Q' for a and b in G. By Proposition 1, Q is an MSSQ if and only if it is equivalent to Q'. So, second, we check equivalence of Q and Q'; since all MSSQs are essentially CQACs, the check is feasible in Π_2^P [15].

In turn, the lower bound is obtained by reduction of the equivalence problem for connected CQACs with a restricted form of comparisons, which can be shown to be Π_2^P-complete by a similar technique as in [15]. The idea of the reduction is to first construct a graph G with entities a and b using the first CQAC q_1 such that q_1 corresponds to an MSSQ for a and b in G; then, to rewrite the second CQAC q_2 into a query Q syntactically compatible with G; and finally to show that Q is the MSSQ for a and b in G if and only if q_1 and q_2 are equivalent. □

We next observe that Algorithm 1 can be easily modified to compute an ESQ, if one exists. Indeed, let algorithm COMPUTE_ESQ be the same as COMPUTE_MSSQ except that it additionally evaluates the constructed query at the end, and returns the query only if the result is precisely a, b, and *fail* otherwise.

Theorem 3. COMPUTE_ESQ *is a procedure that returns an ESQ for its input entities and graph if it exists, or fail otherwise.*

Proof. If the algorithm returns a query Q, then Q is an ESQ for the input entities a and b in the input graph G since this is explicitly checked in the last step. Assume now that the algorithm returns *fail*; we argue that no ESQ exists. If it returns *fail* in line 2, then by the correctness of Algorithm 1 we can conclude that no similarity query (and hence no ESQ) exists for a and b in G. In turn, if the algorithm returns *fail* in the last step, we know that the constructed query Q is not an ESQ. Furthermore, by the correctness of Algorithm 1, we know that Q is an MSSQ for a and b in G, so, by Proposition 2, no ESQ exists. □

Note that the evaluation step in COMPUTE_ESQ does not work in (deterministic) polynomial time. As the following proposition says, no ESQ can be computed in polynomial time (assuming P ≠ NP).

Theorem 4. *The problem of checking whether an ESQ for two entities in a graph exists is* coNP*-complete.*

Proof (Sketch). The upper bound follows from the algorithm: first it computes, in polynomial time, a candidate query Q and then universally guesses an entity different from a and b verifying that it is not an answer to Q. The last can be done in coNP by usual query evaluation algorithms.

The lower bound is obtained by reduction of the coNP-complete problem of checking whether there exists a *difference* comparison-free query Q for an entity a relative to an entity b in an RDF graph G—that is, such that Q has the empty filter condition and has a as an answer over G but not b [17]. In particular, given G, a, and b as instance to the difference existence problem, consider the graph $G' = G \cup \{(a, d, c), (b, d, c)\}$ for fresh entities d and c not occurring in G. Then it is not difficult to check that there exists a difference query for a relative to b in G if and only if there exists an ESQ for a and a in G'. □

We conclude the section with the complexity of checking if a query is an ESQ.

Theorem 5. *The problem of checking whether a query is an ESQ for two entities in a graph is* DP*-complete.*

Proof (Sketch). To establish the upper bound, consider the algorithm that checks in NP that both input entities are answers to the query on the input graph and checks in coNP that there are no other answers. For the lower bound, we first show that the verification problem for comparison-free difference queries is DP-hard, and then reduce this problem to ESQ verification in a way very similar to the one presented in the proof of Theorem 4. □

5 Computing Approximated MSSQs

As discussed in Section 4, algorithm Compute_MSSQ is impractical even for moderately-sized input graphs G since the algorithm computes upfront the product graph $G \times G$ of the input graph G with itself, which is of quadratic size. In this section, we propose a practical algorithm that computes a similarity query for two entities in a graph (if one exists). Although the query computed by the algorithm is not guaranteed to be an MSSQ, we will verify empirically in Section 6 that it is a reasonable approximation in practice. Before going to the details, we make two important observations. First, in the rest of the paper we concentrate on MSSQs leaving similar treatment of ESQs for future work. Second, the theoretical framework and the exact algorithm Compute_MSSQ treat subjects, predicates, and objects in the same way; however, in practice we would like to compare subject and object entities, considering predicates as relations, and hence our approximation algorithm assumes that the compared entities appear in the graph either both as subjects or both as objects at least once (and hence an MSSQ exists).

Our algorithm relies on the notion of a *similarity tree* for entities a and b in a graph G, which we define next. Roughly speaking, a similarity tree is a labelled

directed tree, where each node is labelled with a pair of sets of entities (appearing in subject and object positions in G), with the first set in a pair corresponding to a and the second to b; the root node is labelled with the pair $(\{a\}, \{b\})$. Each edge in the tree is labelled with two sets of entities (appearing in the predicate position in triples from G) and a direction of triples;. Furthermore, we require that the tree is consistent with the structure of G in that each edge in the tree is justified by corresponding triples in G.

Definition 4. *A* pair tree *is a rooted labelled directed tree such that*

- *each node v is labelled with a pair (V_1, V_2), where each V_i is a non-empty set of entities satisfying either $V_1 \cap V_2 = \emptyset$ or $V_1 = V_2 = \{c\}$ for an entity c;*
- *each edge e is labelled with a tuple (E_1, E_2, dir), where each E_i is a set of entities satisfying either $E_1 \cap E_2 = \emptyset$ or $E_1 = E_2 = \{c\}$ for an entity c, and where $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$.*

An edge $e = (v, v')$ in a pair tree $\mathcal{T}$ is justified *in a graph G if the following properties hold for both $i = 1, 2$, where (V_1, V_2), (E_1, E_2, dir), and (V_1', V_2') are labels of v, e, and v', respectively:*

- *for each entity $c \in V_i$ there is a triple* justifying *e in G for c—that is, a triple (s, p, o) such that $p \in E_i$ and either $s = c$ and $o \in V_i'$ when dir is $\rightarrow$, or $s \in V_i'$ and $o = c$ otherwise.*

Pair tree $\mathcal{T}$ is a similarity tree *for entities a and b in graph G if the root is labelled with $(\{a\}, \{b\})$ and all edges in $\mathcal{T}$ are justified in G.*

Consider Figure 2, where a graph G_{ex} and two pair trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are depicted (for brevity, g, f, and r in the trees abbreviate $(\{g\}, \{g\})$, $\{f\}, \{f\}$, and $\{r\}, \{r\}$, respectively). Note that the roots in both trees are labelled by $(\{a\}, \{b\})$. In $\mathcal{T}_1$ the edge between the root and the node labelled $(\{c\}, \{d, d'\})$ is justified: for both a and b there exists a triple in G that has this entity as the subject, f as the predicate, and c and d (or d'), respectively, as the object. However, neither of the other two edges in $\mathcal{T}_1$ is justified, because of the $\{d, d'\}$ component in the parent node label: there are no triples (g, r, d') and (d, f, d) in G. In contrast, every edge in $\mathcal{T}_2$ is justified, and hence $\mathcal{T}_2$ is a similarity tree.

Similarity trees are relevant since they have corresponding similarity queries.

Definition 5. *Let $\mathcal{T}$ be a similarity tree for entities a, b in a graph G. For each node or edge u in $\mathcal{T}$ labelled with (L_1, L_2) or (L_1, L_2, dir), respectively, let t_u be*

- *a variable $?X$ if u is the root of the tree;*
- *the entity c if $L_1 \cap L_2 = \{c\}$; and*
- *a fresh variable otherwise.*

The query corresponding *to $\mathcal{T}$ is* SELECT $?X$ WHERE P FILTER C *with*

- *P containing, for each edge $e = (v, v')$ in $\mathcal{T}$, the triple pattern $(t_v, t_e, t_{v'})$ or $(t_{v'}, t_e, t_v)$ if e is labelled with $\rightarrow$ or $\leftarrow$, respectively; and*
- *C containing, for each node v in $\mathcal{T}$ labelled (V_1, V_2) with each V_i consisting of only integers, the arithmetic comparisons $(t_v \geq \mathit{min})$ and $(t_v \leq \mathit{max})$, where min and max are the minimal and the maximal, respectively, values in $V_1 \cup V_2$.*

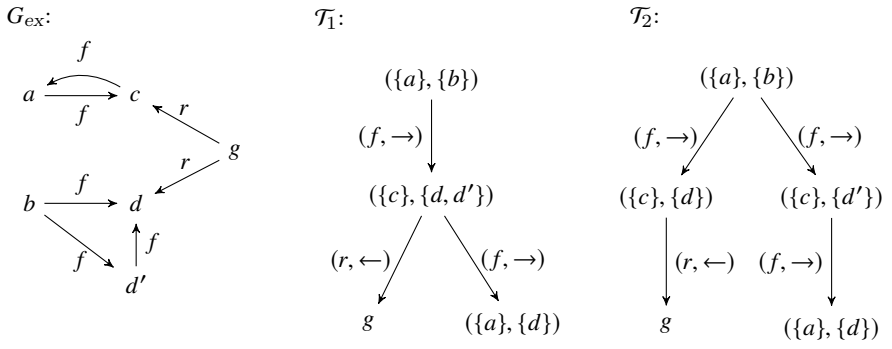

Fig. 2: An example graph G_{ex} and two pair trees $\mathcal{T}_1$ and $\mathcal{T}_2$

The query corresponding to the similarity tree $\mathcal{T}_2$ from Figure 2 is

$$Q_{sim} = \text{SELECT } ?X \text{ WHERE } \{(?X, f, ?Y_1), (g, r, ?Y_1), (?X, f, ?Y_2), (?Y_2, f, ?Y_3)\}.$$

The following proposition establishes that the query corresponding to a similarity tree is indeed a similarity query.

Proposition 3. *The query corresponding to a similarity tree for entities a and b in a graph G is a similarity query for a and b in G.*

Proof (Sketch). Given a similarity tree $\mathcal{T}$ for a and b in G, let us first traverse $\mathcal{T}$ from the root to the leaves and recursively associate each node and edge in $\mathcal{T}$ with a pair of entities such that the first is from the first component of the label of the node or edge and the second is from the second component, as well as the following holds:

- the root is associated with (a, b), and,
- for each edge $e = (v, v')$ with v associated with (c_a, c_b), e and v' are associated with pairs of entities (d_a, d_b) and (c'_a, c'_b), respectively, from the labels of e and v' such that the triples (c_a, d_a, c'_a) and (c_b, d_b, c'_b), if e is labelled by $\rightarrow$, or the triples (c'_a, d_a, c_a) and (c'_b, d_b, c_b) otherwise, justify e in G for c_a and c_b, respectively (such justifying triples exist by Definition 4).

Let Q be the query corresponding to similarity tree $\mathcal{T}$. Consider the valuations ν_a and ν_b that send $?X$ to a and b, respectively, and every other variable $?Y$ of Q to the entities c_a and c_b, respectively, in the pair (c_a, c_b) associated to the node or edge u such that t_u is $?Y$ according to Definition 5. It is immediate to check that valuations ν_a and ν_b justify a and b as answers to Q, as required. □

We are ready to present algorithm COMPUTE_APPROX_MSSQ (given in Algorithm 2), which computes a similarity query of a given depth `dep` (i.e., a natural number) for given entities a and b in a given graph G according to the three steps described next. In the first step (line 2), we create a preliminary pair

Algorithm 2: COMPUTE_APPROX_MSSQ

Input: graph G, entities a and b in G, depth dep
Output: similarity query for a and b in G

1 **let** $\mathcal{T}_0$ be pair tree with a single root node v_0 labelled $(\{a\}, \{b\})$;
2 **let** $\mathcal{T}_{gen}$:= GENERATE_TREE($\mathcal{T}_0$, v_0, G, dep);
3 **let** $\mathcal{T}_{sim}$:= UNCOUPLE_NODES($\mathcal{T}_{gen}$, G);
4 **return** the query corresponding to $\mathcal{T}_{sim}$.

tree $\mathcal{T}_{gen}$. For example, for the input graph G_{ex} from Figure 2, for the entities a and b in that graph and for depth 2 the pair tree $\mathcal{T}_{gen}$ is $\mathcal{T}_1$. As in this example, $\mathcal{T}_{gen}$ may not yet be a similarity tree. Hence, in the second step (line 3), we uncouple some of the nodes in $\mathcal{T}_{gen}$, making all edges in the tree justified, and thus creating a similarity tree $\mathcal{T}_{sim}$. For example, we uncouple the node from $\mathcal{T}_1$ labelled $(\{c\}, \{d, d'\})$ into two new nodes, labelled $(\{c\}, \{d\})$ and $(\{c\}, \{d'\})$, respectively. The former becomes the parent node for the node labelled g, while the latter becomes the parent node for the node labelled $(\{a\}, \{d\})$. As the result, in this example $\mathcal{T}_{sim}$ is $\mathcal{T}_2$. Finally (in step 4), we turn $\mathcal{T}_{sim}$ into a similarity query corresponding to this tree; for example we turn $\mathcal{T}_2$ into Q_{sim}.

Let us look at each of the steps in more detail. In the first step (line 2), the algorithm constructs, by means of the recursive subroutine GENERATE_TREE, a pair tree $\mathcal{T}_{gen}$ of depth at most dep. In particular, in lines 1–2 of COMPUTE_APPROX_MSSQ a root labelled $(\{a\}, \{b\})$ is created and passed to the recursion. When a node v in $\mathcal{T}$ labelled (V_1, V_2) is received in a recursive call of GENERATE_TREE, the following extensions are performed, where $(s, p, o)^{\rightarrow}$ and $(s, p, o)^{\leftarrow}$ denote (s, p, o) and (o, p, s), respectively:

- first, for each direction $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each pair of entities c, d such that, for both $i = 1, 2$, there are triples $(c_i, d, c)^{\mathsf{dir}} \in G$ with $c_i \in V_i$, a new edge labelled $(\{d\}, \{d\}, \mathsf{dir})$ from v to a new node labelled $(\{c\}, \{c\})$ is added to $\mathcal{T}$;
- second, for each $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each entity c such that, for both $i = 1, 2$, there exists $(c_i, d_i, c)^{\mathsf{dir}} \in G$ with $c_i \in V_i$ the sets

$$E_i = \{d_i \mid (c_i, d_i, c)^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$$

  are considered; if the sets $E_1 \setminus E_2$ and $E_2 \setminus E_1$ (i.e., the sets of edge entities not covered in the previous case) are both non-empty, then an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$ from v to a new node labelled $(\{c\}, \{c\})$ is added;
- third, for each $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ and each d such that, for both $i = 1, 2$, there are triples $(c_i, d, c_i')^{\mathsf{dir}} \in G$ with $c_i \in V_i$ the sets

$$V_i' = \{c_i' \mid (c_i, d, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\}$$

  are considered; if $V_1' \setminus V_2'$ and $V_2' \setminus V_1'$ (i.e., the sets of not covered node entities) are non-empty, then an edge labelled $(\{d\}, \{d\}, \mathsf{dir})$ from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$ is added; moreover, if the depth of v is non-zero, then GENERATE_TREE is recursively called for v';

– finally, for both $\mathsf{dir} \in \{\rightarrow, \leftarrow\}$ the sets

$$E_i = \{d_i \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i\} \text{ and}$$
$$V_i' = \{c_i' \mid (c_i, d_i, c_i')^{\mathsf{dir}} \in G \text{ for } c_i \in V_i \text{ and } d_i \in E_i \setminus E_{3-i}\}$$

are considered for both $i = 1, 2$; if the sets $E_1 \setminus E_2$, $E_2 \setminus E_1$, $V_1' \setminus V_2'$, and $V_2' \setminus V_1'$ are all non-empty, then an edge labelled $(E_1 \setminus E_2,\ E_2 \setminus E_1,\ \mathsf{dir})$ from v to a new node v' labelled $(V_1' \setminus V_2',\ V_2' \setminus V_1')$ is added; moreover, if the depth of v is non-zero, then GENERATE_TREE is called for v'.

After all these extensions, $\mathcal{T}$ is returned to the previous level of recursion.

As mentioned above, the resulting $\mathcal{T}_{gen}$ is a pair tree; however, it may not be a similarity tree for a and b, since some edges may not be justified in G. So, in the second step (line 3) of COMPUTE_APPROX_MSSQ, pair tree $\mathcal{T}_{gen}$ is refined from the leaves upwards using subroutine UNCOUPLE_NODES, which ensures that each edge in the tree is suitably justified, and hence yields a similarity tree $\mathcal{T}_{sim}$ for a and b in G. In particular, this subroutine considers nodes of its input pair tree $\mathcal{T}$ from leaves to the root, and for each node v under consideration and each child v' of v—that is, a node with an edge $e = (v, v')$—the following is performed, where (V_1, V_2), (E_1, E_2, dir), and (V_1', V_2') are labels of v, e, and v', respectively:

– a node v^* and an edge (v^*, v') labelled (V_1^*, V_2^*) and $(E_1^*, E_2^*, \mathsf{dir})$, respectively, are added to $\mathcal{T}$, for maximal sets $V_i^* \subseteq V_i$ and $E_i^* \subseteq E_i$, $i = 1, 2$, with (v^*, v') justified by G;
– if v is not the root then an edge (v_p, v^*) labelled as the incoming edge (v_p, v) to v is added to $\mathcal{T}$;
– when all children of v are processed, each group of children with the same label are merged to one, and v is removed.

Note that, by construction, the resulting $\mathcal{T}_{sim}$ is a pair tree as well; moreover, we will see that, contrary to $\mathcal{T}_{gen}$, it is a similarity tree.

Finally, in the last step (line 4), algorithm COMPUTE_APPROX_MSSQ constructs the query corresponding to the similarity tree according to Definition 5, which is guaranteed to be a similarity query by Proposition 3.

Overall, we arrive to the following correctness theorem.

Theorem 6. *For each positive integer* dep*,* COMPUTE_APPROX_MSSQ *computes a similarity query for entities* a *and* b *in a graph* G*.*

Proof (Sketch). The claim follows from the construction and Proposition 3. Indeed, the pair tree $\mathcal{T}_{sim}$ is a similarity tree for input G, a, and b because the root is labelled with $(\{a\}, \{b\})$, while all the edges are processed in UNCOUPLE_NODES in the bottom-up manner and explicitly verified to be justified by G. □

We next briefly discuss the running time of the algorithm. One execution of the GENERATE_TREE subroutine runs in $O(\rho \cdot |G|)$, where ρ is the number of different entities appearing in the predicate position in triples from G. GENERATE_TREE is recursively called at most $(2\rho)^{\mathsf{dep}-1}$ times, hence the full runtime of these calls is $O(\rho^{\mathsf{dep}} \cdot |G|)$. Then the subroutine UNCOUPLE_NODES performs a check on $O(\rho^{\mathsf{dep}} \cdot |G|)$ pair tree nodes, each check being in $O(|G|)$. Hence,

COMPUTE_APPROX_MSSQ runs in $O(\rho^{\mathsf{dep}} \cdot |G|^2)$ in the worst case. Note that ρ for a graph G is typically much smaller in practice than the number of triples in G (e.g., $\rho = 128$ for full YAGO), and the checks in UNCOUPLE_NODES are made for all triples in G containing the current entity, which usually constitute only a small fraction of G. This makes the algorithm suitable for real-case scenarios, which we will demonstrate in the next section.

Finally, we observe that it is possible to find an example where the approximating SQ has arbitrary many answers while the MSSQ has just two (i.e., the input entities). So, there is no constant approximation ratio for our algorithm. However, the same can be said about any approximation algorithm that outputs a SQ that is not an MSSQ, so we cannot hope for such theoretical guarantees. Instead, we evaluate the quality of our approximation empirically in Section 6.

6 Evaluation

We implemented our two similarity algorithms COMPUTE_MSSQ and COMPUTE_APPROX_MSSQ in Python. We then evaluated the performance of our implementations and estimated to what extent the similarity queries computed by algorithm COMPUTE_APPROX_MSSQ approximate MSSQs computed by algorithm COMPUTE_MSSQ in practical cases. We used the following three RDF graphs (datasets) in our experiments:

- the synthetic graph LUBM1 [12] consisting of 100,543 triples over 26,437 entities, out of which 17 appear in the predicate positions;
- a subset of the anonymised Twitter follower graph (TFG) [18] consisting of 713,319 triples over 404,719 entities, only one of which (i.e., entity *follows*) appears in the predicate positions; and
- a subset of YAGO graph [19] consisting of 1,069,072 triples over 604,905 entities, out of which 42 appear in the predicate positions.

The graphs are different in size and nature: YAGO has a rich set of property entities, while TFG uses only one; LUBM1 has a regular structure and resembles data typically encountered in databases, whereas YAGO is more heterogeneous.

All experiments were performed on a MacBook Air laptop with macOS 10.14, 1.6 GHz Intel Core i5 processor, and 16 GB 2133 MHz LPDDR3 memory.

6.1 Performance Analysis

We evaluated the runtime of our implementation of COMPUTE_APPROX_MSSQ for increasing values of the depth parameter. For this, we randomly selected 100 pairs of entities in each graph and, for each such pair, we ran the implemented algorithm for values of the depth parameter ranging from 1 to 4. For each graph and each depth value, we recorded the average, median and maximum runtime as well as the average number of triple patterns in a query amongst all the selected pairs of entities. We limited the maximum depth to 4, since queries beyond that depth are very difficult to comprehend due to their size and structure; indeed, psychologists established precise limitations in the human capacity to store and

		runtime				size
RDF graph	depth	avg	median	max	timeouts	avg
LUBM1	1	0.000851	0.000346	0.006910	–	1.88
	2	0.002690	0.000971	0.036051	–	11.25
	3	0.072132	0.001389	2.101702	–	463.00
	4	0.348439	0.002058	8.558924	–	3235.02
TFG	1	0.000811	0.000356	0.045334	–	0.75
	2	0.001115	0.000373	0.045334	–	3.54
	3	0.058080	0.000415	3.540030	–	592.86
	4	67.203592	11.308518	352.100547	–	35904.21
YAGO	1	0.000918	0.000327	0.056005	–	0.73
	2	0.006476	0.000338	0.175918	–	7.81
	3	8.318439	0.000347	461.952534	–	149.63
	4	84.950921	0.640530	488.342738	3	1287.67

Table 1: Runtime (in seconds) and output query size (in number of triples) of Compute_Approx_MSSQ on the LUBM1, TFG, and YAGO graphs

process information, where experiments show that most people would have trouble keeping in memory chains of related pieces of information longer than 4 [8].

Our results for LUBM1, TFG, and YAGO are summarised in Table 1. We can observe that our similarity queries can be computed efficiently with sub-second average running times in most cases; in contrast our implementation of exact Compute_MSSQ timed out in all cases. The average runtime becomes larger for depth 4 for larger datasets, such as TFG and YAGO; in case of YAGO the algorithm reached 3 timeouts for 500 seconds threshold. However, we can also observe that output queries tend to become very large (and hence difficult to interpret, verbalise, and comprehend) for depths greater than 3. Therefore, it is only practical to consider approximated MSSQs of depth up to 3, for which our algorithm can always compute a similarity query.

6.2 Query Specificity Analysis

In this section we report the results of an experiment that aims to estimate how different the similarity queries computed using Compute_Approx_MSSQ are from the actual MSSQs computed by the exact algorithm Compute_MSSQ. Unfortunately, our implementation of Compute_MSSQ timed out and hence failed to produce a query for all inputs in our datasets; thus, a direct comparison of the answers to the similarity queries produced by the algorithms is not feasible. To circumvent this limitation, we have designed an experiment consisting of the following steps for each of the LUBM1, TFG, and YAGO graphs:

1. we first created 40 random connected graphs, called *pattern graphs*, such that each of them consists of 4 triples, and exactly 20 are acyclic;
2. for each pattern graph G, we created its copy G' with all entities renamed to fresh entities;

RDF graph		MSSQs		Approximations					
				dep = 1		dep = 2		dep = 3	
		avg	%	avg	%	avg	%	avg	%
LUBM1	A	7983.15	30.20	12157.45	45.97	10360.35	39.19	10332.05	39.08
	C	33.65	0.13	6697.00	25.33	2960.45	11.20	2522.35	9.54
TFG	A	156566.47	38.69	161958.50	40.02	161345.60	39.87	156566.47	38.69
	C	42838.20	10.58	83284.95	20.59	82541.10	20.39	78122.65	19.30
YAGO	A	147284.37	24.51	207236.80	34.26	175541.00	29.02	169331.26	27.99
	C	7175.25	1.19	83641.85	13.83	44372.90	7.34	41518.15	6.86

Table 2: Average number of answers (avg) and average percentage of all entities in answers (%) to MSSQs and the approximating queries, computed over acyclic (A) and cyclic (C) pattern graphs and evaluated on the LUBM1, TFG, and YAGO graphs

3. we then picked an entity a from each such G at random and the corresponding a' in the copy G' and ran both algorithms on $G \cup G'$ as a graph and a, a' as input entities; the approximation algorithm was run for depths 1 to 3;
4. finally, we evaluated the resulting queries on the considered graph (LUBM1, TFG, or YAGO) and compared the answers.

Intuitively, each pattern graph G represents a 'pattern' that may occur in the real data (and hence a pattern that will be reflected in the MSSQ). The approximation algorithm Compute_Approx_MSSQ constructs a tree-like query where variables in the predicate positions of triple patterns occur at most once, and hence the query returned by Compute_Approx_MSSQ on a graph $G \cup G'$ may not faithfully reflect the data pattern encoded by G. By evaluating the resulting queries in step 4 we are also assessing how common each pattern is in the graph (based on the number of answers to the MSSQ) as well as how faithfully the approximated query reflects the pattern.

Our results are summarised in Table 2. As can be seen from the average percentage of entities contained in query answer sets, similarity queries computed by Compute_Approx_MSSQ become more specific and closer to MSSQs as the depth grows. Unsurprisingly, the approximating queries evaluated on the TFG graph are almost identical to MSSQs, since the graph contains a single relation. The approximation error consistently goes below 10% for both cyclic and acyclic pattern graphs for depth 3 on all datasets, as can be seen from the percentage for MSSQs and approximated queries of dep = 3. This makes Compute_Approx_MSSQ suitable for real-world applications of entity comparison.

7 Related Work

Exploring relationships between entities in RDF graphs is a recent and growing research topic. Some approaches focus on general relatedness and connectedness of entities. They explore paths connecting given entities together and analyse patterns in these paths [1,6,11,14,16]. More generic approaches look at patterns

that are common for several entities in a graph. An approach by El Hassad et al. [9,10] attempts to find commonalities between Web resources by computing the least general generalisation (lgg) of the RDF data containing these resources. The computation is based on the RDFS entailment rules, and an lgg is itself an RDF graph that entails subgraphs of the input RDF dataset that contain the target Web resources. To the best of our knowledge, our recent work [17] is the only one focussing not only on patterns common for input entities (see Sec. 3), but also on patterns that differentiate input entities from each other.

The problem of computing similarity and difference queries can be viewed as an instance of the query reverse engineering (QRE) problem; in case of exact similarities, the problem becomes an instance of the definability problem, a more restricted version of QRE. In particular, the QRE problem for a query language takes as input a dataset and two disjoint sets of positive and negative example tuples of constants, and decides whether there exists a query in the language whose answers over the dataset contain all the positive examples but none of the negative examples. The definability problem is the same except there are no negative examples, but the answers to the query should be exactly the positive examples. Both problems have been studied for various query languages [5, 13, 20, 22, 23], including SPARQL [2] and CQs [4, 21]. Hence, our work contributes to the field by setting complexity bounds for monadic unary CQACs.

Finally, computing MSSQs is related to the problem of finding the least common subsumer for description logic (DL) concepts [3, 7], which, for two individuals and a set of concept and role names, requires to compute a DL concept that contains both individuals and is most specific modulo concept subsumption.

8 Conclusion and Future Work

We investigated the problem of entity comparison in knoweldge graphs, taking as the basis our recently proposed framework [17], in which entity comparison is modelled via similarity queries. In particular, we extended the language of similarity queries to consider a richer fragment of SPARQL allowing for numeric filter expressions, and studied the complexity of computing various similarity queries in this fragment. We also proposed and implemented a scalable algorithm that is guaranteed to compute a similarity query and can be used on large knowledge graphs. An immediate step of future research is to study difference queries in the extended query language, and to create scalable algorithms for computing difference queries that are as generic as possible for the given entities in a graph. Another important problem is to present similarity and difference queries to the user in a comprehensible way, which is not trivial given their size and complicated structure. Possible solutions include splitting the queries into subqueries and ranking, visualising or verbalising them, and allowing the users to iteratively expand only the parts of queries they are interested in. Once these problems are solved, a comprehensive entity comparison tool would be possible.

Acknowledgements This research was supported by the SIRIUS Centre for Scalable Data Access and the EPSRC projects DBOnto, MaSI^3, and ED^3.

References

1. C. Aebeloe, G. Montoya, V. Setty, and K. Hose. Discovering diversified paths in knowledge bases. *Proceedings of the VLDB Endowment*, 11(12):2002–2005, 2018.
2. M. Arenas, G. I. Diaz, and E. V. Kostylev. Reverse engineering SPARQL queries. In *Proc. of WWW*, pages 239–249, 2016.
3. F. Baader and A.-Y. Turhan. On the problem of computing small representations of least common subsumers. In *Proc. of KI*, pages 99–113, 2002.
4. P. Barceló and M. Romero. The complexity of reverse engineering problems for conjunctive queries. In *Proc. of ICDT*, pages 7:1–7:17, 2017.
5. A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases. In *Proc. of EDBT*, pages 109–120, 2015.
6. G. Cheng, Y. Zhang, and Y. Qu. Explass: exploring associations between entities via top-K ontological patterns and facets. In *Proc. of ISWC*, pages 422–437, 2014.
7. S. Colucci, F. M. Donini, S. Giannini, and E. Di Sciascio. Defining and computing least common subsumers in RDF. *Web Semant.*, 39:62–80, 2016.
8. N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behav. Brain Sci.*, 24(1):87–114, 2001.
9. S. El Hassad, F. Goasdoué, and H. Jaudoin. Learning commonalities in RDF. In *Proc. of ESWC*, pages 502–517, 2017.
10. S. El Hassad, F. Goasdoué, and H. Jaudoin. Learning commonalities in SPARQL. In *Proc. of ISWC*, pages 278–295, 2017.
11. V. Fionda and G. Pirrò. Explaining and querying knowledge graphs by relatedness. *Proc. of the VLDB Endowment*, 10(12):1913–1916, 2017.
12. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semant.*, 3(2-3):158–182, 2005.
13. V. Gutiérrez-Basulto, J. C. Jung, and L. Sabellek. Reverse engineering queries in ontology-enriched systems: The case of expressive horn description logic ontologies. In *Proc. of IJCAI*, pages 1847–1853, 2018.
14. P. Heim, S. Hellmann, J. Lehmann, S. Lohmann, and T. Stegemann. RelFinder: Revealing relationships in RDF knowledge bases. In *Proc. of SAMT*, pages 182–187, 2009.
15. A. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.
16. J. Lehmann, J. Schüppel, and S. Auer. Discovering unknown connections - the DBpedia relationship finder. *Proc. of CSSW*, 113:99–110, 2007.
17. A. Petrova, E. Sherkhonov, B. Cuenca Grau, and I. Horrocks. Entity comparison in RDF graphs. In *Proc. of ISWC*, pages 526–541, 2017.
18. R. A. Rossi and D. F. Gleich. A dynamical system for PageRank with time-dependent teleportation. *Internet Math.*, 10(1), 2014.
19. F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A large ontology from Wikipedia and Wordnet. *Web Semant.*, 6(3):203–217, 2008.
20. W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.
21. B. ten Cate and V. Dalmau. The product homomorphism problem and applications. In *Proc. of ICDT*, pages 161–176, 2015.
22. Y. Y. Weiss and S. Cohen. Reverse engineering SPJ-queries from examples. In *Proc. of PODS*, pages 151–166, 2017.
23. M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *Proc. of SIGMOD*, pages 809–820, 2013.