Personalized Best Answer Computation in Graph Databases

Michael Ovelgönne[1], Noseong Park[2], V.S. Subrahmanian[1,2], Elizabeth K. Bowman[3], and Kirk A. Ogaard[3]

[1] UMIACS, University of Maryland, College Park, MD, USA
[2] Department of Computer Science, University of Maryland, College Park, MD, USA
[3] Tactical Information Fusion Branch, Computational and Information Sciences, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, USA
{mov,npark,vs}@umiacs.umd.edu
{elizabeth.k.bowman.civ,kirk.a.ogaard.ctr}@mail.mil

Abstract. Though subgraph matching has been extensively studied as a query paradigm in semantic web and social network data environments, a user can get a large number of answers in response to a query. Just like Google does, these answers can be shown to the user in accordance with an importance ranking. In this paper, we present scalable algorithms to find the top-K answers to a practically important subset of SPARQL-queries, denoted as *importance queries*, via a suite of pruning techniques. We test our algorithms on multiple real-world graph data sets, showing that our algorithms are efficient even on networks with up to 6M vertices and 15M edges and far more efficient than popular triple stores.

1 Introduction

Facebook recently introduced a new feature called "graph search"[1] that enables users to search Facebook's social graph. This graph contains entities like persons, media items, companies, events, and associated data of these entities like name or age. For example, users can search for *cities that friends of their parents like* or *restaurants their friends have been to*. Such queries are a special case of SPARQL queries and of the class of subgraph matching queries (for the first example the pattern is the path graph *user* $\leftrightarrow$ *parent* $\leftrightarrow$ *friend* $\leftrightarrow$ *city*) with additional constraints on the vertex properties.

In this paper, we go beyond subgraph matching and consider the case of subgraph queries augmented with "importance" metrics that are specified by the user in his query. Such queries can be easily expressed in SPARQL using FILTER and ORDER BY clauses. In classical subgraph queries, the user specifies a query subgraph – and all matches of that subgraph with subgraphs of the graph database are considered equally important. However, when the nodes in the graph have associated semantic labels, then there are cases where the user may specify an importance measure that marks some matches as being "more important" than others.

[1] https://www.facebook.com/about/graphsearch

A query for restaurants a person's friends have been to can return hundreds or thousands of answers, as many Facebook users have hundreds of friends. However, users will prefer a short list of the *most relevant* restaurants. We want to provide users a tool to find the most relevant answers from their perspective. An *importance query* is an extended subgraph query with a scoring mechanism, and as such they are a subset of SPARQL queries. With importance queries users can, e.g. search for restaurants with the *highest star ranking* their friends like or the *largest* cities friends of the parents have been to.

Figure 1 shows a Facebook-style graph with four types of edges (friend of, resident of, located in, likes). Each vertex in this graph has different kinds of properties such as the type of the vertex (person, restaurant, city), the age and gender of persons, and the star rating of restaurants. A possible query on this graph is: Which are the restaurants with the highest star ranking in London that my friends who live in London like?

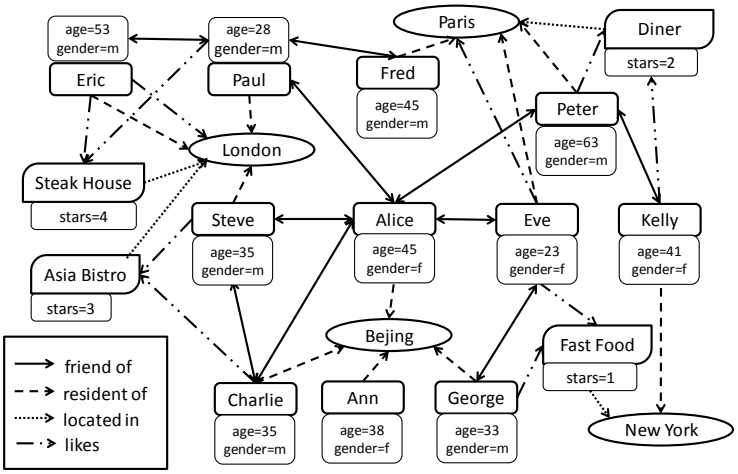


Fig. 1: Example of a data graph.

Our technical solution for finding the most important pattern matches in a graph is based on classic subgraph matching algorithms rather than join-based techniques for RDF database engines. Especially because of expensive join operations, finding patterns in large graph-structured datasets stored as triples is inefficient [16]. Our experimental evaluation compares the newly developed algorithms to RDF triples stores, and shows that our algorithms beat them on importance queries.

An important point to note is that in this paper we only consider *anchored* queries, i.e. subgraph queries where we already know the mapping for at least one of the vertices in the query. This is often a more realistic problem setting compared to arbitrary, non-anchored queries, because people usually create

searches with themselves as an anchor (queries containing terms like *my* friends, *my* company, cities *I* like).

In this paper we extend the subgraph matching problem and try to find the *most important* matches (according to a user provided definition) in attributed graphs (i.e. graphs with edge labels and where vertices may have associated properties). We make the following contributions.

- First, we formally define importance queries and define answers (and the top-k answers) to such queries (Section 2).
- We then define a simple baseline algorithm to solve such queries, followed by our more sophisticated OptIQ algorithm that can efficiently prune part of the search space and scale our top-k algorithms to find answers to importance queries (Sections 3 and 4).
- We present the results of experiments to analyze the influence of query properties on the performance of query algorithms (Section 5). Our experiments – on CiteSeerX, YouTube, Flickr and GovTrack data, show that our algorithms scale well to data sets containing up to 6.2M vertices and 15.2M edges. We also show that popular triple stores are much slower in answering importance queries.

2 Importance Queries on Graphs

In this section, we formalize the concept of *importance queries* – the query type we developed fast answering algorithms for. We use the following notation in this paper. $\mathbb{R}$ denotes the set of all non-negative real numbers. VP, EP, and VAR are arbitrary but fixed mutually disjoint sets of symbols for *vertex predicates*, *edge labels* and *variables*, respectively. Variable symbols start with a "?" (e.g. ?x). Every vertex predicate $p \in \mathsf{VP}$ has a *domain* $dom(p)$ which is some set disjoint from each of $\mathsf{VP}, \mathsf{EP}, \mathsf{VAR}$.

Definition 1 (Graph Database). *A* graph database *(GDB) is a triple* $\mathcal{G} = (V, E, \wp)$ *with* V *a finite set of* vertices, $E \subseteq V \times \mathsf{EP} \times V$ *a finite set of labeled edges and* $\wp : V \times \mathsf{VP} \to \bigcup_{p \in \mathsf{VP}} dom(p)$ *a* property function. *We assume that for all* $v \in V, p \in \mathsf{VP}$, $\wp(v, p) \in dom(p)$.

$V_\mathcal{G}, E_\mathcal{G}, \wp_\mathcal{G}$ denote the vertices, edges, and property functions of a GDB $\mathcal{G}$. Throughout this paper, we assume that $\mathcal{G} = (V, E, \wp)$ is an arbitrary but fixed graph. Figure 1 shows a sample of such a GDB.

Definition 2 (Term; Numeric Term). *(i) Every member of* $\bigcup_{p \in \mathsf{VP}} dom(p)$ *is a term. If* $nt \in \bigcup_{p \in \mathsf{VP}} dom(p) \cap \mathbb{R}$, *then* nt *is a* numeric term.
(ii) If $?x \in \mathsf{VAR}$ *and* $p \in \mathsf{VP}$, *then* $?x.p$ *is a term. If* $dom(p) \subseteq \mathbb{R}$, *then* $?x.p$ *is a* numeric term.
(iii) If nt_1, nt_2 *are numeric terms, then* $nt_1 + nt_2$ *and* $nt_1 * nt_2$ *are numeric terms.*
A term is ground *if no variables occur in it. We say a term* t *is* solely about variable $?x$ *if* $?x$ *is the only variable occurring in* t.

The importance query (definition follows) shown in Figure 2 contains the numeric term $?r.stars$. We assume that all *ground* numeric terms are evaluated, e.g. the numeric term $2 + 3$ is evaluated to 5.

Definition 3 (Constraint). *(i) If t_1, t_2 are terms, then $t_1 = t_2$ and $t_1 \neq t_2$ are constraints.*
(ii) If nt_1, nt_2 are numeric terms, then $nt_1 < nt_2, nt_1 \leq nt_2, nt_1 > nt_2, nt_1 \geq nt_2$ are constraints.
(iii) If c_1, c_2 are constraints, then $c_1 \wedge c_2$ is a constraint.
We say constraint C is solely about variable $?x$ *if $?x$ is the only variable occurring in C.*

The example importance query in Figure 2 contains for variable $?r$ the constraint $?r.type = restaurant$.

Definition 4 (Importance Query). *An* importance query *is a 4-tuple $PQ = (SQ, \chi, \varrho, agg)$ where:*

1. *SQ is a pair $SQ = (QV, QE)$ where $QV \subseteq V \cup \mathsf{VAR}$ and $QE \subseteq (V \cup \mathsf{VAR}, \mathsf{EP}, V \cup \mathsf{VAR})$. Because V and VAR are finite sets, QV and QE are finite sets as well. SQ is called a* subgraph query*.*
2. *χ associates a constraint that is solely about $?x$ with each variable $?x \in QV \cap \mathsf{VAR}$.*[2]
3. *ϱ is a partial function from $QV \cap VAR$ to numeric terms s.t. there is at least one $?x \in QV \cap VAR$ which is mapped to a numeric term with $?x$ occurring in it.*
4. *agg is one of four aggregation function MIN, MAX, SUM or AVG.*[3]

Suppose $SQ = (QV, QE)$ is a subgraph query. A *substitution* is a mapping $\theta : QV \cap \mathsf{VAR} \to V$. Thus, substitutions assign vertices in a GDB $\mathcal{G}$ to variables in QV. The *application* of a substitution θ to a term t, denoted $t\theta$, is the result of replacing all variables $?x$ in t by $\theta(?x)$. When t contains no variables, then $t\theta = t$.

If we consider the sample query Q shown in Figure 2, it has two answers w.r.t. the graph database shown in Figure 1:

$$\theta_1 \equiv ?p = Steve, ?r = AsiaBistro$$
$$\theta_2 \equiv ?p = Paul, ?r = SteakHouse$$

Definition 5 (Answer; Answer Value). *Suppose $\mathcal{G}$ is a GDB, $PQ = (SQ, \chi, \varrho, agg)$ is an importance query, and θ is a substitution w.r.t. SQ. θ is an* answer *of PQ w.r.t. $\mathcal{G}$ if:*

[2] If we do not wish to associate a constraint with a particular variable $?x$, then $\chi(?x)$ can simply be set to a tautologous constraint like $2 = 2$.
[3] These functions map multisets of reals to the reals and are defined in the usual way.

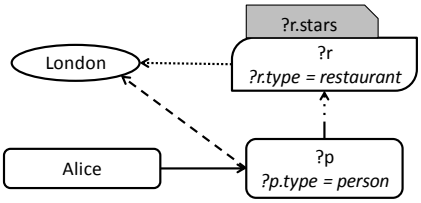


Fig. 2: Example of an importance query described by a subgraph query, constraints (italic) and an IQ-term (gray box).

(i) for every edge $(v_1, ep, v_2) \in QE$*, it is the case that* $(v_1\theta, ep, v_2\theta) \in E$ *and (ii) for each vertex* $?x \in QV \cap \mathsf{VAR}$*, the constraint* $\chi(?x)\theta$ *is true.*

The answer value *of a substitution* θ*, denoted* $\mathsf{Aval}(\theta, PQ, \mathcal{G}) = agg(\{(\varrho(?x))\theta \mid ?x \in dom(\varrho)\})$*. When the set on the right hand side is empty,* $\mathsf{Aval}(\theta, PQ, \mathcal{G}) = 0$*.*

We use $\mathsf{ANS}(PQ, \mathcal{G})$ *to denote the set of all answers of importance query PQ w.r.t. GDB* $\mathcal{G}$*.*

In our example ϱ assigns the very simple IQ-term $?r.stars$ to the variable $?r$. So the answer value of θ_1 is 3 and the answer value of θ_2 is 4.

3 Baseline Best Answer Algorithm

In Section 2 we defined importance queries. Depending on the size of the data graph and the constrained IQ-query, the number of results can be very large. We defined the notion of importance queries as users are usually only interested in the most important query answers. Consequently, we will discuss top-k query answer algorithms.

A straightforward algorithm to compute the answers of an importance query follows the definition of importance queries and first computes all subgraph query answers, filters the set of answers to those answers that satisfy the constraints and then computes the IQ-values. Any subgraph matching algorithm could be used here (see Sec. 6). However, we use an implementation that considers our specific problem situation (queries with anchors and large disk-residing graphs). Subgraph matching algorithms are branch-and-bound algorithms that follow a search tree. In our case, first, an anchor is selected. Then an unmapped neighbor of an anchor or a mapped variable in the query graph gets selected, and all candidates for this variable in the data graph are determined. For every candidate the variable is mapped to the candidate, and the search with the next unmapped variable is continued recursively. We only use the I/O- efficient pruning on vertex degrees because determining vertex degrees does not require one to read extra data. Loading index data for advanced indexes from disk usually does not pay off.

Algorithm 1: Optimized Importance Query (OptIQ) Algorithm

1 FUNCTION `AnswerQuery`
Input: Data Graph G, Importance Query $q = (SQ = (QV, QE), \chi, \varrho, agg)$, partial substitution θ, result size k
Output: answered stored in global variable A: set of tuples $(vertex, score)$
2 **if** θ *maps every variable to a ground term* **then**
3 $A \leftarrow A \cup \{\theta\}$
4 **if** $|A| > k$ **then**
5 $A \leftarrow A \setminus \{\theta \in A$ with minimal `score(`θ`)` $\}$
6 $nextvars \leftarrow \{(c, ?v) | ?v \rightarrow c$ or $c \rightarrow ?v \in QE\}$ //edges with one mapped endpoint
7 **foreach** $(c, ?v) \in nextvars$ **do**
8 $R_{c,?v} \leftarrow$ `getNeighborNum(`G, c, `getEdgeLabel((`$c, ?v$`)))`
9 $B_{c,?v} \leftarrow$ `getExpBenefit(`G, q, $(c, ?v), \theta$`)` // for WCOST only
10 **if** $R_{c,?v} = 0$ **then return**
11 $(c, ?w) \leftarrow (c, ?v) \in nextvars$ with max $B_{c,?v}$ // for WCOST
12 with min $R_{c,?v}$ // otherwise
13 $N_{?w} \leftarrow$`GetValidNeighbors(`G, q, $(c, ?w)$`)`
14 **foreach** $m \in N_{?w}$ *in decreasing order of score* $\varrho(m)$ **do**
15 $\theta' \leftarrow \theta \cup (?w \rightarrow m)$
16 $s \leftarrow$ `calculateMaxScore` (G, θ')
17 **if** $|A| > k$ *and* $s <$ *lowest score of any* $\theta \in A$ **then** continue
18 `AnswerQuery(`G, $q\theta'$,θ',k`)`
19 FUNCTION `GetValidNeighbors`
Input: Data Graph G, query q, tuple (vertex c, variable $?w$)
Output: vertices that can be mapped to $?w$ among all c's neighbors
20 FUNCTION `getNeighborNum`
Input: Data Graph G, vertex c, edge label l
Output: The number of c's neighbors which are connected through an edge of the label l
21 FUNCTION `getExpBenefit`
Input: Data Graph G, query q, tuple (vertex c, variable $?w$), partial mapping θ
Output: `getExpScore` $(G, q, (c, ?w), \theta)$ / `getCost` $(G, (c, ?w))$
22 FUNCTION `getExpScore`
Input: Data Graph G, query q, tuple (vertex c, variable $?w$), partial mapping θ
Output: $agg(\{?v \in QV \cap VAR : value(?v)\})$, where $value(?v) = \varrho(?v)\theta$ if θ maps $?v$, $value(?v) =$ `localAvg`$(?v)$ if all candidates for $?v$ are in a known, cached subgraph of the subgraph index of G, and $value(?v) = 0$ otherwise
23 FUNCTION `getCost`
Input: Data Graph G, tuple (vertex c, variable $?w$)
Output: $n \log n$, where n=`getNeighborNum` $(G, (c, ?w))$, i.e. sorting time in l. 14
24 FUNCTION `calculateMaxScore`
Input: Data Graph G, partial mapping θ
Output: $agg(\{?v \in QV \cap VAR : value(?v)\})$, where $value(?v) = \varrho(?v)\theta$ if θ maps $?v$, $value(?v) =$ `localMax`$(?v)$ if all candidates for $?v$ are in a known, cached subgraph of the subgraph index of G, and $value(?v) =$ `globalMax`$(?v)$ otherwise

4 Optimized (OptIQ) Algorithm

The baseline algorithm (Sec. 3) performs the 4 steps (1) Subgraph Matching, (2) Constraint Checking, (3) Scoring and (4) Top-k Selection sequentially and independently. An obvious improvement is the integration of the uncoupled steps. If we check the constraints in the subgraph matching step, then we do not have to create a possibly large list of subgraph matches that needs to be checked for meeting the constraints. Likewise, we can maintain a sorted list of top-k substitutions. Every time a new substitution with a score greater than the lowest score in the top-k list has been identified, we update the list.

Algorithm 1 shows the integrated algorithm. All blue code segments are extensions to improve the performance, but are not necessary to compute answers to IQ-queries *per se*. We will discuss these improvements in Sections 4.2–4.4.

Lines 2–5 check whether a complete substitution has been generated, and add a complete substitution to the answer set if its score is among the top-k. Lines 6–10 inspect every edge of the query graph whose one end is mapped to a vertex of the data graph and whose other end is not. In $R_{c,?v}$, we store the number of c's neighbors in the data graph that are connected through an edge with the same label between c and $?v$, i.e. $R_{c,?v}$ is the number of candidates for $?v$. In line 11 we select the query graph edge with the lowest number of candidates. `GetValidNeighbors`() returns the set of all valid vertices that can be mapped to $?w$. Here, we use DOGMA's pruning technique based on IPD values (see Section 4.1) to filter neighbors that cannot be part of a valid answer. Other pruning strategies (see e.g. [4, 13, 14]) could be used as well. In line 14–18, we substitute $?w$ with each candidate m and recursively continue the assembly of answers.

Before we can discuss the performance improving techniques shown in the blue code segments of Algorithm 1, we need to introduce our graph database index.

4.1 Database Index

To efficiently answer importance queries on large graphs, we use a disk-based index inspired by the DOGMA index [3]. We decompose the data graph into a large number of small, densely connected subgraphs and store them in an index. Our partitioning algorithm follows the multi-level graph partitioning scheme [7]. Like DOGMA, we iteratively halve the number of vertices by merging randomly selected vertices with all of their neighbors. When the resulting graph has less than 100 vertices, we iteratively expand the graph using the GGGP algorithm from the METIS algorithm package [5] to bisect the graph components at each level.[4] For every block of the partition we extract the subgraph it induces from the graph database and store it as one block of data to disk.

The objective of the DOGMA index is two-fold: (1) to increase the I/O-efficiency by exploiting data locality – only those parts of the graph that are

[4] We also conducted preliminary experiments with other partitioning algorithms but they showed no significant difference for the query processing performance.

necessary to answer a query have to be retrieved from disk (2) DOGMA stores for every vertex the internal partition distance (IPD), i.e. the number of hops from a vertex to the nearest other vertex outside the subgraph. Using the IPD, we can quickly compute a minimum distance between vertices, and prune candidates if their distance is higher than the distance between their respective query graph variables.

We extend this concept and store additional information in the index for advanced top-k pruning strategies. First, we store global maximum values for every vertex property. Additionally, we store together with each induced subgraph G_s the edges that connect it to other subgraphs (inter-subgraph edges) and aggregated information (maximum and average) of the predicate values of the vertices in G_s and of those vertices not in G_s but adjacent to a vertex in G_s (denoted as the *boundary* of G_s).

4.2 Simple Top-k Pruning on Scores

The optimized baseline algorithm does not exploit the fact that we are only interested in the top-k answers. During the stepwise assembly of substitutions, there will be partial substitutions which cannot make it into the top-k given the scores of the full substitutions that are already in the answer set. If we identify them, we can prune the respective branch of the search tree and save computation time.

First, the set $N_{?w}$ should be sorted by score in line 14. I.e., if $?w$ is scored by an IQ-term, $N_{?w}$ is sorted in decreasing order of the value of the term. This ensures that we evaluate the most promising candidates first.

The IQ-score of a substitution is the value of the aggregation function agg on the values assigned by the IQ-terms to the variables (see Def. 5). For a partial substitution θ, we can compute an upper bound of its answer value Aval by using upper bounds for $\varrho(?v)\theta$ of all unmapped variables. That means, we calculate an upper bound of the answer value by using the exact term score for every previously mapped variable and upper bounds for currently unmapped variables. This is performed by `calculateMaxScore`(). Our simple top-k pruning strategy uses precomputed global upper bounds, i.e. $max_{x \in V} \wp(x, p_i)$, for each vertex property p_i.

When the variance of vertex property values is high, using the global upper bound of a vertex property will not allow us to prune many branches of the search tree. A tighter upper bound is desirable. The mappings of the partial substitutions restrict the set of valid candidates for the currently unmapped variables. What we need is a fast way to find tight upper bounds for vertex property values given the mappings in the partial substitutions.

4.3 Advanced Top-*k* Pruning on Scores

In Section 4.2 we presented a simple top-k pruning strategy using upper bounds for the reachable substitution scores. Using the proposed database index, we can find tighter upper bounds that provide a higher pruning power.

For the candidate set $N_{?v}$ of $?v$, we can compute the upper bound for $\varrho(?v)$ using $max_{x \in N_{?v}} \wp(x, p_i)$. But computing the upper bound in this way would require us to read the property scores of all vertices in $N_{?v}$. This is prohibitively expensive because of the high costs of reading from disk. However, if we store the maximal property scores of a subgraph in the index, we can find a good upper bound in a reasonable amount of time.

In `calculateMaxScore`(), we compute the upper bound of the answer value of a partial mapping θ by computing the upper bound for each variable $?v$'s $\varrho(?v)$ (denoted as `value`()). If θ maps a variable $?v$ to a vertex of the data graph, we know the exact value of $\varrho(?v)\theta$. For a currently unmapped $?v$, we look at its distance to already mapped variables c, $dist(c, ?v)$. If $dist(c, ?v) < IPD(c)$ for some c, we know that $?v$ has to be mapped to the same subgraph as c. Then, we use `localMax` to compute `value` using the local maximum values of the subgraph of c. If $dist(c, ?v) < IPD(c) + 1$, we do the same but using the maximum values of the subgraph and its boundary. However, if $dist(c, ?v) > IPD(c) + 1$ for all c, we have no local information and `globalMax` computes `value` using global maximum values.

4.4 Processing Order

The baseline algorithm iteratively selects the unmapped variable with the smallest candidate set for processing. However, for importance queries this strategy sometimes leads to the late discovery of top-k answers. Selecting a variable with a higher number of candidates might not be bad when most candidates can be pruned very early. To weigh the different objectives (low number of branches to follow, following more promising paths first) we compute the benefit score $B_{c,?v}$ in line 9 and process candidates in decreasing order of their benefits. We define the benefit of substituting a variable $?w$ with n candidates in a partial substitution θ' as $wexp(\theta')/f(n)$, where $f(n)$ is the cost to process n candidates and $wexp(\theta')$ is the expected score of θ'. In Algorithm 1, `getExpBenefit`() calculates this score.

As in the case of computing upper bounds for substitution scores for pruning (Section 4.3), we compute $wexp(\theta')$ with the precomputed property scores of subgraphs. But additionally we weigh the expected term scores using the indegree of a candidate. The indegree is a simple heuristic for the probability that the variable will be mapped to a vertex. As I/O-efficiency is the primary problem of our algorithm, we use only information already read from disk to determine the expected score. Unavailable vertex property score estimates are replaced by 0.

To compute the expected value we proceed as follows. We classify unmapped variables in the query graph in two groups.

- If a variable $?v$ has no vertex c whose $hop(?v, c)$ is less than or equal to c's IPD value, we assume the property scores are 0 (or ∞ if the MIN aggregation is used). Computing an expected value would require reading many additional disk pages (which we want to avoid) or using global averages. But

underestimating the real expected score is in this case favorable because it puts variables whose mapping requires additional disk access at the end of the priority list.
– Otherwise, we use the weight expected value of the subgraph c is residing in. We know that all query variables whose distance from c is less than or equal to c's IPD value will be mapped in the same subgraph. So, we can use the precomputed weighted average property values of the subgraph as the expected value.

5 Experiments

In the following, we present an evaluation of the previously introduced top-k algorithms. We conducted experiments with 5 algorithm variants: the non-integrated baseline algorithm Base, the optimized importance query (OptIQ), the extension of OptIQ by simple top-k pruning (GMax), the extension of OptIQ by advanced top-k pruning (LMax), and the extension of LMax by the improved processing order (WCOST).

To see how our algorithms perform in relation to triple stores, we ran additional experiments using Apache Jena TDB 2.10.0 [1] and OWLIM-SE 5.3.5925 [8]. We considered using RDF-3x [10] as well, but had to omit RDF-3x because it cannot answer queries with cross-products because of a bug that still exists in the latest release. Importance queries can be easily written in SPARQL with its FILTER and ORDER BY clauses.

5.1 Experimental Setup

To evaluate our algorithms, we use four real-world datasets. Basic properties of these datasets are shown in Table 1.

Name	#Vertices	#Edges	#V.Prop.	#E.Labels
CiteSeerX	0.93M	2.9M	5	4
YouTube	4.6M	14.9M	8	3
Flickr	6.2M	15.2M	4	3
GovTrack	120K	1.1M	5	6

Table 1: Evaluation datasets

We analyze the performance of the algorithms with randomly generated importance queries. We created the queries by selecting random subgraphs of the data graph with n vertices and m edges. Random subgraphs are created by starting with a random vertex of the data graph. We iteratively add a randomly selected vertex from the neighborhood of any previously selected vertices. From the random subgraphs we created IQ-queries in the following way. We randomly selected c vertices of the subgraph, defining them as anchors, and mapped to

the respective vertices of the data graph. The remaining $n - c$ vertices of the randomly selected subgraph are defined as variables. The edges (including the edge labels) of the subgraph are edges in the query. With a probability p, a constraint is created from a numeric property of a vertex in the random subgraph. With a equal probability a constraint is a $>$ or $<$ constraint. The reference value of a $>$ constraint is the property value in the subgraph - 1, and the reference value of a $<$ constraint is the property value in the subgraph + 1. Scoring terms are created similarly to constraints. With a probability t, a numeric property of a vertex is select to be included in an IQ-term. If an IQ-term consists of more than one property, the properties are concatenated with a +. The aggregation function is MAX or SUM with equal probability.

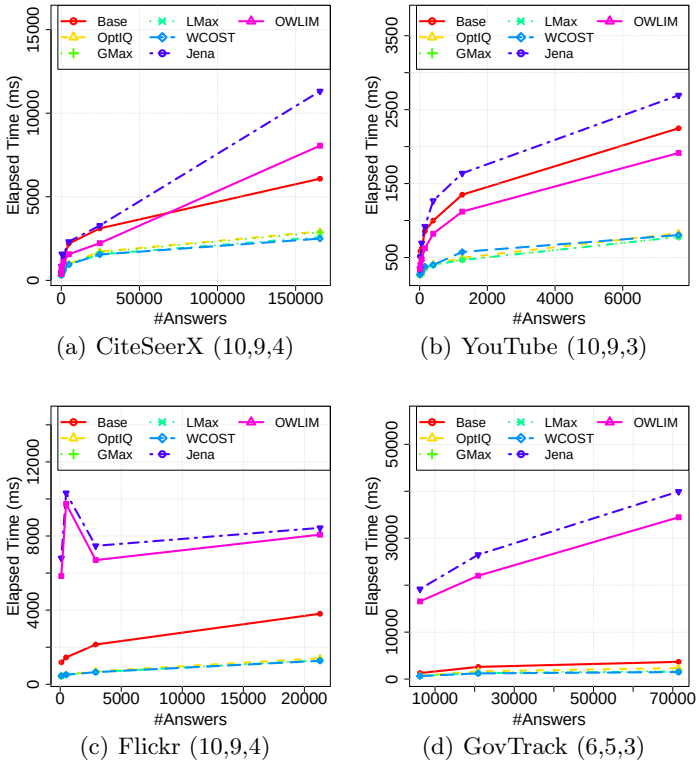
This query generation process ensures that all queries have at least one solution (which is the random subgraph the query has been generated from) and that (in probability) the distribution of structural patterns and properties used in constraints and query terms in a set of random queries resembles the respective distributions in the data graph.

5.2 Experimental Results

We evaluated our system by the selectivity of a query (i.e. the number of answers a query has), the size of the query (i.e. the number of vertices and edges the subgraph query has) and the number of desired answers. We used a set of 1000 random queries for the experiments.

Results by selectivity Figure 3 shows the runtime in relation to the answer size of the subgraph query. All algorithms show a sub-linear increase in the runtime with an increasing answer size. Reading subgraphs from disk is a dominating factor of the total runtime. The number of answers to the subgraph query increases much faster than the required number of subgraph reads because usually many answers lie in the same subgraphs. Compared to the baseline more sophisticated algorithms like WCOST and LMax can receive good speed-ups in some but not all settings - especially when the answer size is high. For non-selective queries our algorithms are up to one order of magnitude faster than the evaluated triple stores. For some datasets (Flickr, GovTrack) triple stores perform considerably worse even for very selective queries.

Results by subgraph query size For experiments on the subgraph query size, we use 2 pairs of query types (1000 random queries each) where each pair differs in the number of edges. The results of Figure 4 show that our algorithms scale very well in the number of vertices. As we increase the number of edges in a query, the runtime usually decreases as the query gets more selective (i.e. has fewer answers). Once again we see that our algorithms perform much better than the triple stores, and the performance difference is especially high for complex queries.



(a) CiteSeerX (10,9,4) (b) YouTube (10,9,3)

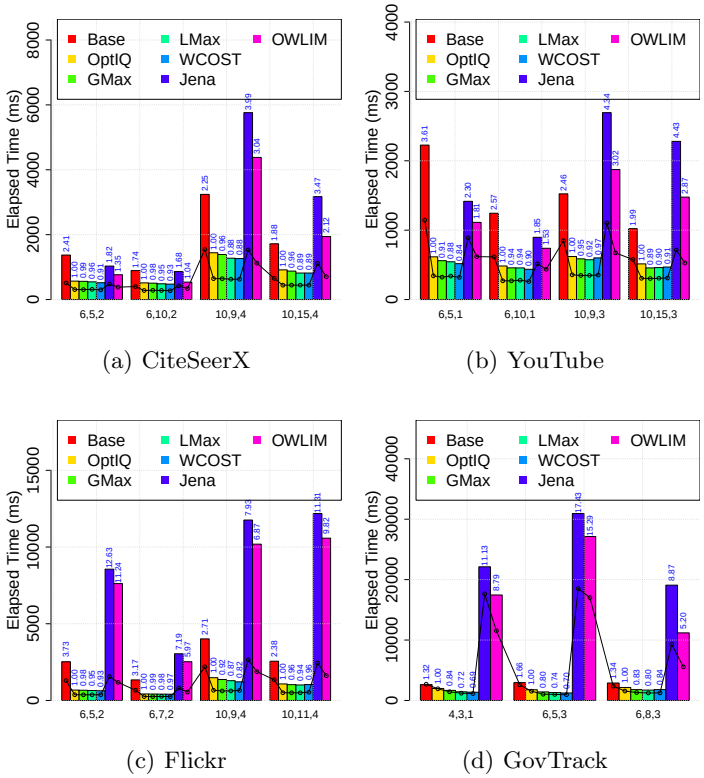
(c) Flickr (10,9,4) (d) GovTrack (6,5,3)

Fig. 3: Results by selectivity. Each caption shows the query size, e.g. 6,5,2 means 6 vertices, 5 edges, and 2 anchors.

Results by parameter k We analyzed the impact of the desired number of answers on runtime. Overall the scaling of our algorithms with respect to the number of answers is very good (see Figure 5), and is much less marked than for the triple stores. The almost constant runtime of our algorithms for low values of k is once again the result of the domination of the total runtime by the time needed to read a subgraph from disk. When most subgraphs in the neighborhood of the anchors have to be read to find the top-1 answer then the time to create a few additional solutions is low.

6 Related Work

We presented algorithms to identify the best answers to importance queries on attributed graphs. We extended subgraph matching algorithms to answer these queries, as non-specialist database systems (SQL as well as RDF databases) have a bad performance on complex subgraph queries. Subgraph queries on relational



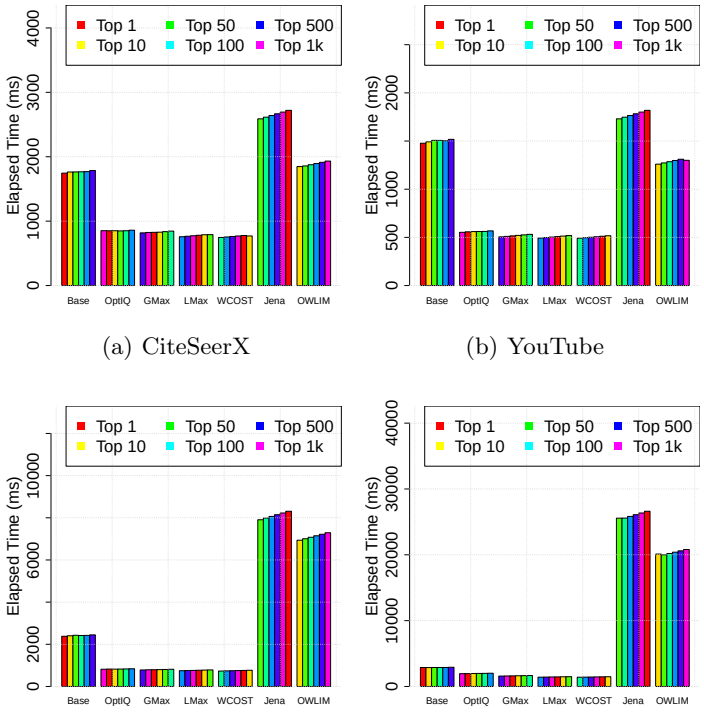
(a) CiteSeerX (b) YouTube

(c) Flickr (d) GovTrack

Fig. 4: Results by query size (Top 10, bar: mean, line: median).

databases require many expensive self-joins on a potentially very large edge table. RDF databases built on top of relational databases suffer from the same problem. Only RDF databases that store their data as graphs and not as triples could potentially provide a good performance. Recently Zou et al. [16] showed the performance advantage of subgraph matching algorithms compared to join-based algorithms used by triple stores for answering SPARQL queries. A way to improve the performance of triple stores for top-k SPARQL queries has been proposed by Magliacane et al. [9]. They presented a rank-aware join algorithm for top-k SPARQL queries. However, SPARQLRank supports only limited ranking functions and in particular does not support aggregation functions in the ranking term as we do.

So answering importance queries via subgraph matching algorithms is the best available approach, and with the Base algorithm we presented the straightforward way to answer importance queries by calling a subgraph matching algorithm. But we also showed that we can do much better than the simple Base approach by using sophisticated pruning techniques. Pruning strategies for subgraph matching have been discussed for decades. A considerable amount of lit-



(a) CiteSeerX (b) YouTube

(c) Flickr (d) GovTrack

Fig. 5: Results by parameter k of top-k queries.

erature has been published on subgraph matching on a single large graph. Since the early work of Ullman [12] most work on subgraph matching has been conducted on finding better ways to prune the search space of branch-and-bound algorithms. State-of-the-art algorithms store sophisticated graph invariants in precomputed indexes [4, 13, 14] to speed up the search. Invariants (in the simplest case the degree of a vertex) can be used to determine whether a vertex in the graph database *cannot* be a mapping for a variable in the query graph in an answer. Good overviews on different algorithms and pruning techniques are in [6, 11].

Our problem setting is different to this classical problem in two very important ways. First, classic subgraph matching searches only for structural patterns without anchors. This makes the overall computational effort much higher and is usually – depending on the dataset – in the range of hours. Second, the data graph is stored in memory. In our problem setting with anchored queries, the computational effort is much lower. However, our objective is to answer queries in interactive settings within seconds on large, disk-resident datasets. So I/O effi-

ciency is an important issue for us. As we showed, we can answer most anchored queries in less than a second. This is less time than an in-memory algorithm spends loading the data graph into memory.

We defined importance queries as an extension of standard subgraph queries, but deriving it from approximate [15] or probabilistic [2] subgraph matching definitions is straightforward. Approximate matching algorithms do not search for exact matches, but for a subgraph similar to the query graph. Probabilistic matching algorithms work on probabilistic graphs, i.e. graphs that model the probability of the existence of an edge. The intent is to address the problem of errors in the data or limited knowledge of the system that is modeled in the graph. The techniques we developed for the advanced WCOST and GMAX algorithms could be transfered to related problems. Join-based algorithms for triple stores lack this flexibility.

7 Conclusion and Future Work

In this paper, we motivated and defined the problem of importance queries on graph databases. Such queries can also be expressed in SPARQL through the FILTER and ORDER BY constructs. We designed query algorithms for efficient retrieval of top-k answers to importance queries and evaluated the performance of the algorithms on large real-world.

By computing upper-bounds for the IQ-scores of partial substitutions, our most advanced algorithms are able to prune branches of the search tree that will not lead to a top-k answer. Thus, these algorithms achieve a significantly better performance than naive implementations. Our best algorithms need less than a second to answer the majority of our random test queries on graphs with up to about 15 million edges.

We believe importance queries are an important next step to personalized graph queries. As a next step, we plan to extend the concept to probabilistic subgraph matching. Then we can extend the search to probabilistic graph databases. For example, image probabilistic "acquaintance" edges in Facebook-style graph databases inferred from the co-occurrence of people in images.

Acknowledgements. We thank the anonymous reviewers and Barry Bishop and Nikolay Krustev from Ontotext for their helpful remarks. Parts of this work have been funded by the US Army Research Office under grant W911NF0910206.

Bibliography

[1] Apache Software Foundation: Apache Jena, `http://jena.apache.org`

[2] Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: Probabilistic subgraph matching on huge social networks. In: 2011 International Conference on Advances in Social Networks Analysis and Mining (ASONAM). pp. 271–278 (2011)

[3] Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: DOGMA: A disk-oriented graph matching algorithm for RDF databases. In: The Semantic Web – ISWC 2009, Lecture Notes in Computer Science, vol. 5823, pp. 97–113. Springer Berlin Heidelberg (2009)

[4] Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10), 1367–1372 (2004)

[5] G. Karypis, V.K.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing 20(1), 359–392 (1998)

[6] Gallagher, B.: Matching structure and semantics: A survey on graph-based pattern matching. In: 2006 AAAI Fall Symposium on Capturing and Using Patterns for Evidence Detection. pp. 45–53 (2006)

[7] Hendrickson, B., Leland, R.: A multi-level algorithm for partitioning graphs. Proceedings of the 1995 ACM/IEEE Conference on Supercomputing p. 28 (1995)

[8] Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM - a pragmatic semantic repository for OWL. In: WISE Workshops. Lecture Notes in Computer Science, vol. 3807, pp. 182–192. Springer Berlin Heidelberg (2005)

[9] Magliacane, S., Bozzon, A., Della Valle, E.: Efficient execution of top-k sparql queries. In: The Semantic Web – ISWC 2012. Lecture Notes in Computer Science, vol. 7649, pp. 344–360. Springer Berlin Heidelberg (2012)

[10] Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proc. VLDB Endow. 1(1), 647–659 (Aug 2008)

[11] Solnon, C.: All different-based filtering for subgraph isomorphism. Artificial Intelligence 174(12–13), 850 – 864 (2010)

[12] Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM 23(1), 31–42 (1976)

[13] Washio, T., Motoda, H.: State of the art of graph-based data mining. SIGKDD Exploration Newsletter 5(1), 59–68 (2003)

[14] Zhang, S., Li, S., Yang, J.: Gaddi: distance index based subgraph matching in biological networks. In: Proceedings of the 12th International Conference on Extending Database Technology. pp. 192–203. EDBT '09, ACM (2009)

[15] Zhang, S., Yang, J., Jin, W.: Sapper: subgraph indexing and approximate matching in large graphs. Proc. VLDB Endow. 3(1-2), 1185–1194 (2010)

[16] Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: answering SPARQL queries via subgraph matching. Proc. VLDB Endow. 4(8), 482–493 (2011)