Approximate and Incremental Processing of Complex Queries against the Web of Data

Thanh Tran, Günter Ladwig, and Andreas Wagner

Institute AIFB, Karlsruhe Institute of Technology, Germany
{ducthanh.tran,guenter.ladwig,a.wagner}@kit.edu

Abstract. The amount of data on the Web is increasing. Current exact and complete techniques for matching complex query pattern against graph-structured web data have limits. Considering web scale, exactness and completeness might have to be traded for responsiveness. We propose a new approach, allowing an affordable computation of an initial set of (possibly inexact) results, which can be incrementally refined as needed. It is based on approximate structure matching techniques, which leverage the notion of neighborhood overlap and structure index. For exact and complete result computation, evaluation results show that our incremental approach compares well with the state of the art. Moreover, approximative results can be computed in much lower response time, without compromising too much on precision.

1 Introduction

Recently, large amounts of semantic data has been made publicly available (e.g., data associated with Web pages as RDFa[1] or Linked Data[2]). The efficient management of semantic data at Web-scale bears novel challenges, which have attracted various research communities. Several RDF stores have been implemented, including *DB-based solutions* such as RDF-extensions for Oracle and DB2, Jena, Sesame, Virtuoso or *native solutions* for RDF like OWLIM, HStar, AllegroGraph, YARS [10], Hexastore [17] and RDF-3X [14]. Recently, also *IR technologies*, in particular the inverted index has been proposed for managing RDF data [18].

We observe that all these systems focus on computing complete and exact answers. Exact matching in a Web setting (with billions of RDF triples), however, results in unacceptable response times especially w.r.t. complex SPARQL[3] queries. The success of current Web search engines suggest that exact matching might be not needed. A more practical direction towards responsive and scalable solutions for Web-scale semantic data management is approximate matching equipped with sophisticated mechanisms for ranking. In this paper, we focus on the problem of *approximate matching* and how to refine matches *incrementally*.

Contribution. We propose an approach for matching complex query patterns against the Web of Data. Our approach allows an "affordable" computation of an initial set of *approximate* results, which can be *incrementally* refined as needed. Our main contributions are:

[1] `http://w3.org/TR/xhtml-rdfa-primer/`
[2] `http://www.w3.org/DesignIssues/LinkedData.html`
[3] `http://www.w3.org/TR/rdf-sparql-query/`

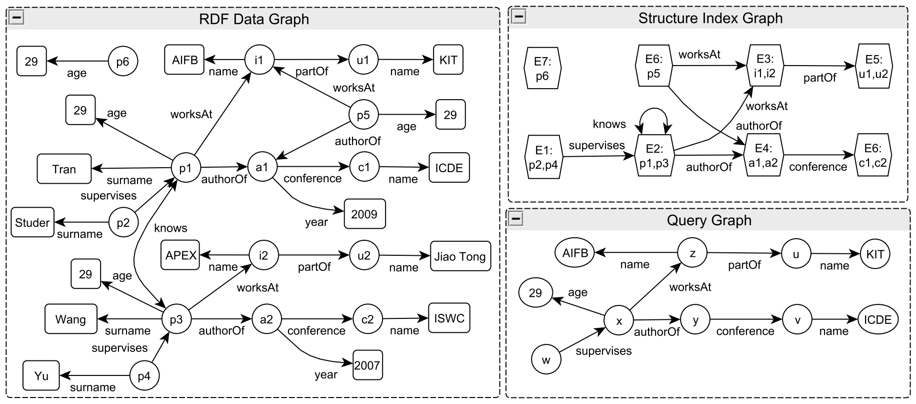


Fig. 1. a) A data graph, b) its structure index graph and c) a query graph.

– We propose a pipeline of operations for *graph pattern matching*, where results can be obtained in an *incremental* and *approximate* manner. We thereby allow a trade-off between precision and response time.
– Via four phases, results are reported early and can be incrementally refined as needed: First, entities matching the query are computed. Then, structural relationships between these entities are validated in the subsequent phases. To our knowledge, this is the first proposal towards a *pipelined processing* of complex queries for *incremental result computation*.
– For processing structural relationships, we introduce a novel *approximate structure matching technique* based on neighborhood overlap and show how it can be implemented efficiently using Bloom filters [3]. Another approximation is introduced for result refinement, which instead of using the large data graph, operates at summary level.
– Via a benchmark, we show that our incremental approach compares well w.r.t. time needed for computing exact and complete results. Further, it is promising w.r.t. approximate result computation.

Outline. In Section 2, we define the problem, describe the state-of-the-art and compare it to our approach. We discuss entity search in Section 3. In Section 4, we present our approximate structure matching technique, followed by the refinement phase in Section 5. We present an evaluation in Section 6. Finally, we conclude with Section 7.

2 Overview

In this section, we define the problem, discuss the state-of-the-art, highlight our main contributions and compare them with related work.

Definition 1. *A* data graph G *is a tuple* (V, L, E) *where* V *is a set of nodes connected by labeled edges* $l(v_1, v_2) \in E \subseteq V \times V$ *with* $v_1, v_2 \in V$ *and* $l \in L$. *Further,* V *is the union* $V_E \uplus V_D$ *with* V_E *representing entity nodes and* V_D *representing data nodes.* E *is the union* $E = E_R \uplus E_A$*, where* $E_R \subseteq V_E \times V_E$ *represents relations between entity nodes and* $E_A \subseteq V_E \times V_D$ *stands for entity attributes.*

Note, our graph-structured data model is of interest in the Semantic Web and database community, as it captures RDF[4], XML and relational data. Further, we consider *conjunctive queries*, a fragment of many query languages (e.g., SQL and SPARQL).

Definition 2. *A* conjunctive query $q = (V_v \uplus V_c, P_r \uplus P_a)$ *is an expression* $p_1 \wedge \ldots \wedge p_n$*, where* $p_i \in P_r \uplus P_a$ *are* query atoms *of the form* $p(n_1, n_2)$ *with* $n_1 \in V_v$*,*$n_2 \in V_v \uplus V_c$ *being* variables V_v *or* constants V_c *otherwise, and* p_i *are called* predicates*. We distinguish between* relation query atoms $p_r \in P_r$ *and* attribute query atoms $p_a \in P_a$*, where* p_r *and* p_a *are drawn from labels of relation edges* E_R *and attribute edges* E_A *respectively. Relation query atoms paths* $(p_{r_1}, \ldots, p_{r_k})$ *contained in q have maximum length* k^{max}*.*

Note, conjunctive queries can be conceived as graph patterns (corresponding to basic graph patterns in SPARQL). Fig. 1a, 1c depict a data and a query graph $q(V_q = V_v \uplus V_c, P_q = P_r \uplus P_q)$, with atoms as edges and variables (constants) as nodes. A match of a conjunctive query q on a graph G is a mapping μ from

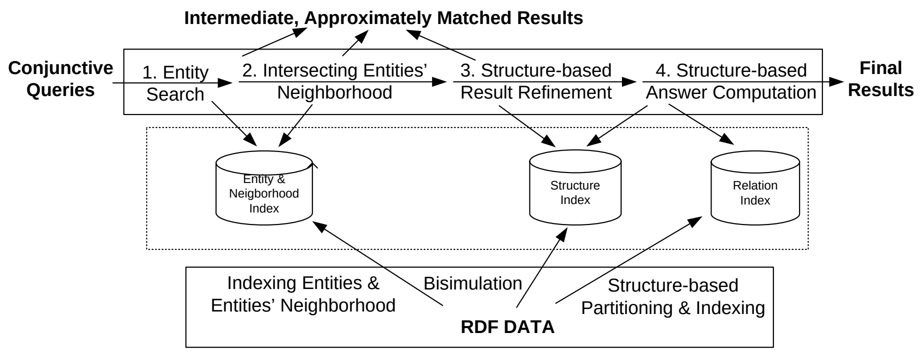

Intermediate, Approximately Matched Results
Conjunctive Queries
1. Entity Search
2. Intersecting Entities' Neighborhood
3. Structure-based Result Refinement
4. Structure-based Answer Computation
Final Results
Entity & Neigborhood Index
Structure Index
Relation Index
Indexing Entities & Entities' Neighborhood
Bisimulation
Structure-based Partitioning & Indexing
RDF DATA

Fig. 2. Offline data preprocessing and online query answering.

variables and constants in q, to vertices in G such that the according substitution of variables in the graph representation of q would yield a subgraph of G. Query processing is a form of *graph pattern matching*, where the resulting subgraph of G exactly matches q. All such matching subgraphs are returned. As opposed to such an *exact* and *complete* query processing, an *approximate* procedure might output results, which only partially match the query graph (i.e., a result matches only some parts of q). A query processing procedure is *incremental*, when results computed in the previous step are used for subsequent steps.

Related Work. Much work in RDF data management targets orthogonal problems, namely data partitioning [1] and indexing [10, 17]. We now discuss related approaches that focus on the problem of query processing.

– *Query Processing.* Matching a query against a data graph is typically performed by retrieving triples and joining them along the query atoms. Join processing can be greatly accelerated, when the retrieved triples are already sorted. Sorting is the main advantage of vertical partitioning [1] and sextuple indexing [17] approaches, which feature data partitioning and indexing strategies that allow fast (nearly linear) merge joins. Further efficiency gains can be achieved by finding an optimal query plan [14].

[4] `http://www.w3.org/TR/rdf-primer/`

- *Approximate Query Processing.* Above approaches deal with exact and complete query processing. In the Semantic Web community, notions for structural [11] and semantic approximation [7] have been proposed. So far, the focus is on finding and *ranking* answers that only approximately match a query. In database research, approximate techniques have been proposed for "taming the terabytes" [8, 5, 2]. Here, focus lies on *efficiency.* Instead of using the actual data, a query is processed over an appropriate synopsis (e.g., histograms, wavelets, or sample-based). Further, a suitable synopsis for XML data as been suggested [15], in order to compute approximate answers to twig-pattern queries. Unlike approaches for flat relational data [8], the synopsis used here takes both structural and value-based properties of the underlying data into account. Essentially, the synopsis is a structural summary of the data, which is augmented with statistical information (e.g., count or value distribution) at nodes and edges.
- *Incremental Query Processing.* Related to our incremental approach is work on top-k query processing. Different algorithms for top-k query processing have been proposed [12]. Here, the goals is to not compute all results, but to allow early termination by processing only the k best results.

Overview of our Approach. Fig. 2 illustrates the main concepts and techniques of our approach. The data graph is broken down into two parts. While attribute edges $a \in E_A$ are stored in the entity index, relations $r \in E_R$ are stored in the relation index. Also, a summary of the data (structure index [16]) is computed during data preprocessing. These indexes are employed in various operators in the pipeline, which we propose for query processing. We rely on sorted merge join and reuse related techniques [1, 17]. However, as opposed to such exact and complete techniques, operations in our pipeline match the query against the data in an approximate way to obtain possibly incorrect answers (which are refined during the process). Instead of operating on all intermediate answers, it is possible to apply a *cutoff* or let the user choose the candidates at every step.

Firstly, we decompose the query into entity queries and perform an *entity search* (ES), storing the results in sorted entity lists with a maximum length of *cutoff*. These results match attribute query atoms only. The next step is *approximate structure matching* (ASM): we verify if the current results also match the relation query atoms. By computing the overlap of the neighborhood of the entities obtained from the previous step, we verify if they are "somehow" connected, thereby matching the relation query atoms only in an approximate way. During structure-based result refinement (SRR), we further refine the matches by searching the structure index (a summary of the data) for paths, which "might" connect entities via relation query atoms. Only in the final step (*structure-based result computation* (SRC)), we actually use edges in the data graph to verify if these connections indeed exist, and output the resulting answers (exactly matching the query).

Example 1. During ES, we obtain 4 entity queries $\{q_x, q_z, q_u, q_v\}$ from the initial query (Fig. 1c), and the corresponding results $\{(p1, p3, p5, p6), i1, u1, c1\}$, for a *cutoff* ≤ 4. This and the results of the subsequent refinement steps are summarised in Table 1. During ASM, we find that all $p1, p3$, $p5$ are somehow connected with the other entities, leading to 3 tuples. During SRR, we find out that $p5$ is in the extension $E6$, and that this structure index node has no incoming *supervise* edge. Thus, $p5$ cannot be part of an answer to q_x. During SRC, we observe that the previous approximate techniques lead to one incorrect result: $p3$ could not be pruned through ASM, because $p3$ *knows* $p1$, and is thus "indirectly" connected with the other entities $i1, u1, c1$. $p3$ could also not be pruned through SRR, because when looking only at the summary (i.e., structure index), $p3$ exhibits the same structure as $p1$ (i.e., it is also in $E2$) and thus, must be considered as a potential result. Clearly, using SRC we can not find out that $p3$ is actually not connected with $i1$ via *worksAt* (thus, could be ruled out).

Design Rationales and Novelties. Our design is based on the observation that state-of-the-art techniques perform well w.r.t queries containing highly selective atoms (e.g., attribute atoms with a constant). Query atoms containing variables (e.g., relation query atoms), on the other hand, are more expensive. Considering Web-scale, these query atoms become prohibitive. Processing $type(x, y)$ or $friendOf(x, y)$ for instance, requires millions of RDF triples to be retrieved. When dealing with complex graph patterns having many relation query atoms (that might involve a large number of triples), we propose a pipeline of operations, which starts with "cheap" query atoms to obtain an initial set of approximate answers, and incrementally continues with refining operations via more expensive query atoms.

Work on data partitioning and indexing [1, 10, 17] are orthogonal, and complement our solution. Also, existing techniques for exact and complete query processing based on sorted merge join are adopted [1, 17]. Building upon these previous works, we present the first solution towards a *pipelined processing of complex queries* on Web data, enabling results to be computed approximately, incrementally, and reported early.

In particular, our approach is the first *approximate technique* for querying RDF data, which is capable of trading precision for time: approximately matched results can be reported early, and when needed, result precision can be improved through several subsequent refinement steps. Compared to existing techniques, the structure refinement step (SRR) resembles a technique for approximate twig pattern matching [15]. The difference is that our structure index is a synopsis for general graph-structured data, while the synopsis employed in [15], is for hierarchical XML data only. Different from any previous techniques, we introduce an additional level of approximation. This is realized by ASM, a novel approximate join operator that exploits the notion of neighborhood overlap for structure matching.

ES				**ASM**			
q_x	q_z	q_u	q_v	q_x	q_z	q_u	q_v
p1	i1	u1	c1	p1	i1	u1	c1
p3	i1	u1	c1	p3	i1	u1	c1
p5	i1	u1	c1	p5	i1	u1	c1
p6	i1	u1	c1				

SRR				**SRC**			
q_x	q_z	q_u	q_v	q_x	q_z	q_u	q_v
p1	i1	u1	c1	p1	i1	u1	c1
p3	i1	u1	c1				

Table 1. The different results for ES, ASM, SRR and SRC.

As opposed top-k approaches, our *incremental approach* does not compute the best, but all approximate results, which are then iteratively refined in several steps. In particular, we do not focus on ranking aspects in this work and simply apply a predefined cutoff to prune large results.

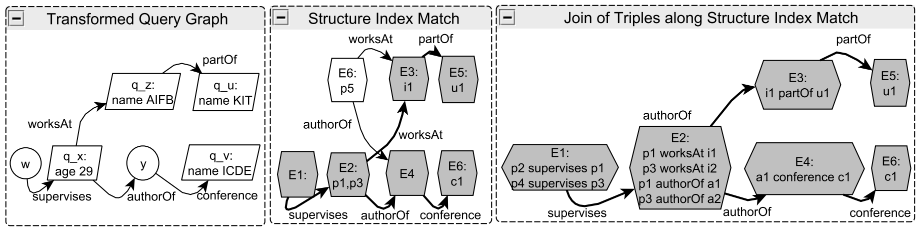


Fig. 3. a) The transformed query graph obtained in ES, b) the structure index match computed in SRR and c) SRC through joins along the structure index match.

3 Entity Search

Let us first describe offline entity indexing and then online entity search.

Entity Indexing. Attributes $a \in E_A$ that refer to a particular entity are grouped together and represented as a document (ENT-doc) in the inverted index. We use structured document indexing – a feature supported in standard IR engines such as Lucene – to store entities' attributes values in different fields: we have (1) *extension id*: the entity's extension id (used for SRR, c.f. Sec. 5); (2) *denotations*: the entity's URI and names; (3) *attributes*: concatenation of attribute/value; (4) *k-neighborhood*: neighbor entities reachable via paths with max. length k.

Query Decomposition. The goal is to parse the query graph q for computing intermediate results as follows:

– *Entity queries* Q_E. Each entity query $q_{var} \in Q_E$ is composed of a set of attribute query atoms $p_a(var,\ con) \in P_a$. Every q_{var} requires entities var to have attribute values, matching the specified constants $con \in V_c$.
– *Maximum distance.* $d^{max}_{q_x}$ is the max. distance between an entity query $q_x \in Q_E$ and the other $q_y \in Q_E$, where distance $d_{q_x}(q_y)$ between q_x and q_y is the length of the shortest path of relation atoms connecting the variable nodes x and y.
– *Transformed query.* $q'(Q_E \subseteq V_{q'}, P_r)$ contains entity queries $q_e \in Q_E$ as nodes and relation query atoms $p_r \in P_r$ connecting these nodes. This is a compact representation of q, where attribute query atoms p_a are collapsed into the entity queries q_e, i.e., each entity query node in q' represents a set of attribute query atoms from q.

For result computation, we select an attribute query edge $p_a(var, con)$ randomly and create an entity query q_{var} for the node var. Other attribute query edges referring to the same node var are added to q_{var}. Starting with this entity query node, we construct the transformed query graph q' by breadth-first-searching (BFS) q. We add visited relation query atoms as edges to the transformed query, and when attribute query atoms are encountered, we use them to create entity queries in same way as we did for $p_a(var, con)$. During the traversal, the length of visited relation chains are recorded. This allows us to compute the distance for every entity query pair. That is, for every entity query q_x, we compute its distance $d_{q_x}(q_y)$ to other entity queries q_y. Finally, the maximum distance is computed for every entity query q_x from this information, i.e., $d^{max}_{q_x} = \arg\max\{d_{q_x}(q_y) : q_x, q_y \in Q_E\}$.

Processing Entity Queries. Every entity query is evaluated by submitting its attribute query atoms as a query against the entity index, i.e., $q_e = \{p_{a_1}(e, con_1), \ldots, p_{a_n}(e, con_n)\}$ is issued as a conjunction of terms "$p_{a_1}//con_1, \ldots, p_{a_n}//con_n$". We use Lucene as the IR engine for indexing and for answering entity queries specified as keywords. Given q_e, this engine returns a sorted list of matching entities, where the maximum length of the list is less than a predefined *cutoff* value.

Example 2. The query q shown in Fig. 1c is decomposed into the entity queries q_x, q_z, q_u, q_v, resulting in the transformed query q' (Fig. 3a). For this, we start with $age(x, 29)$ to create $q_x = \{age(x, 29)\}$. Then, we traverse the relation edges to obtain $q' = \{q_x, worksAt(q_x, z),\ authorOf(q_x, y)\}$. Further, encountering $name(z, AIFB)$ results in $z = q_z = \{name(z, AIFB)\}$. The process continues for the remaining edges of q. For entity search, entity queries like q_x for instance, is submitted as "$age//29$" to obtain the list of entities $(p1, p3, p5, p6)$.

4 Approximate Structure Matching

So far, the entity query parts of q have been matched, while the remaining p_r still have to be processed. Typically, this structure matching is performed by retrieving triples for the entities computed previously (i.e., edges $e \in E_R$ matching p_r), and joining them along p_r. Instead of an equi-join that produces exactly matched results, we propose to perform a *neighborhood join* based on the intersection of entities' neighborhoods. We now define this novel concept for approximate structure matching and discuss suitable encoding and indexing techniques.

Definition 3. *The* k-neighborhood *of an entity $e \in V_E$ is the set $E^e_{nb} \subset V_E$ comprising entities that can be reached from e via a path of relation edges $e_r \in E_R$ of maximum length k. A* neighborhood overlap $e_1 \bowtie^{nb} e_2$ *between two entities e_1, e_2 is an evaluation of the intersection $E^{e_1}_{nb} \cap E^{e_2}_{nb}$, and returns true iff $e_1 \bowtie^{nb} e_2 \neq \emptyset$ s.t. e_1 is connected with e_2 over some paths of relations $e \in E_R$, otherwise it returns false. A* neighborhood join *of two sets $E_1 \bowtie^{nb} E_2$ is an equi-join between all pairs $e_1 \in E_1, e_2 \in E_2$, where e_1 and e_2 are equivalent iff $e_1 \bowtie^{nb} e_2$ returns true.*

Managing neighborhood via Bloom filters. For every entity node $e \in V_E$, we compute its k-neighborhood via BFS. Then, all elements in this neighborhood (including e) are stored in the entity index using the *neighborhood* field. We store the neighborhoods of entities as Bloom filters [3], a space-efficient, probabilistic data structure that allows for testing whether an element is a member of a set (i.e., the neighborhood). While false negatives are not possible, false positives are. The error probability is $(1 - e^{-f \times n/m})^f$, where m is the size of the Bloom filter in bits, n is the number of elements in the set and f is the number of hash functions used [3]. During the neighborhood computation, we count the number of neighbors n for each entity, and set the parameter m and f according to a probability of false positive that can be configured as needed.

Approximate matching via Bloom filters. Checking for connection between two entity queries $q_{e_1}, q_{e_2} \in Q_E$ can be achieved by loading candidate triples matching query edges p_r and then performing equijoins between the candidates and the entities E_1, E_2 obtained for q_{e_1}, q_{e_2}. However, because this may become expensive when a large number of triples match edges p_r, we propose to check for connections between these entities in an approximate fashion via a neighborhood join $E_1 \bowtie^{nb}_{E_{filter}} E_2$. This operates on the Bloom filters associated with the entities only, i.e., does not require retrieval and join of triples. In particular, the join is evaluated by processing $e_1 \bowtie^{nb} e_2$ for all $e_1 \in E_1$ and $e_2 \in E_2$ in a nested loop manner, using the filters of elements in E_1 or E_2 denoted by E_{filter}.

For processing $e_1 \bowtie^{nb}_{e_2} e_2$, we evaluate if $e_1 \in E^{e_2}_{nb}$ using the filter of e_2. Performing neighborhood overlap this way requires that the neighborhood index built for e_2 covers e_1, i.e., $k \geq d_{e_2}(e_1)$. This means that for supporting queries with relation paths of a maximum length k^{max}_q, we have to provide the appropriate neighborhood index with $k = k^{max}_q$. Note that for checking connections between entities in the lists E_1 and E_2 along a chain of k query atoms p_r, only one set of Bloom filters has to be retrieved to perform exactly one neighborhood join, while with the standard approach, $k + 1$ equi-joins have to be performed on the triples retrieved for all p_r.

The approximate matching procedure based on this neighborhood join concept is shown in Alg. 1. It starts with the center query node q_{center}, i.e., the one with lowest *eccentricity* such that maximum distance to any other vertex is minimized (where *eccentricity*$(q_x) = d^{max}_{q_x}$, the distance information computed previously). From q_{center}, we process the neighbor query nodes by traversing them in depth-first search (DFS) fashion. For every $q_{neighbor}$ in the current DFS path, we neighborhood join the entities associated with this node with entities in the result table A (line 9). Note, at the beginning, we marked the center node as q_{filter}. This is to indicate that filters of $E_{q_{center}}$ should be used for neighborhood join as long as possible, i.e., until finding out that $E_{q_{neighbor}}$ is at a distance greater than k. In this case, we proceed with the filters of $E_{q_{lastSeen}}$, the elements lastly processed along the path we currently traversed (line 7). By starting from q_{center}, we aim to maximize the "reusability" of filters.

Example 3. The 2-neighborhoods for $p1, p3$, $p5$ and $p6$ are shown in Fig. 3a. For instance, for $p1$ the neighborhood is obtained by BFS to reach the 1-hop neighbors $p3, i1$ and $a1$ and finally, the 2-hops neighbors $p5, u1$ and $c1$. In Fig. 4a, we illustrate the bloom filter encoding of the neighborhood of $p3$, using three hash functions. We start with entities for q_x (Fig. 3a), as it has the lowest eccentricity of 2, i.e., $q_x = q_{center}$. Via BFS of the query starting from q_x, we arrive at the 1-hop neighboring query nodes q_z and y. First, we use the filters of E_{q_x} $(k = 2)$ to check for overlap between entities E_{q_x} and E_{q_z}, i.e., lookup if $i1$ is in any of the filters retrieved for $p1, p3, p5$ and $p6$ (Fig. 4b) – to find out that $e_1 \bowtie^{nb} p_n \neq \emptyset$, except for $p_n = p_6$. Since y is not an entity query, no processing is required here. When encountering 2-hops neighboring nodes q_u and q_v, we find that the current filters are still usable, because distance to these nodes $d_{q_x}(q_u), d_{q_x}(q_v) = k = 2$. If $k = 1$ instead, we would need to retrieve the filter of $i1$ to check for set membership of $u1$, i.e., set $q_{filter} = q_z$ for processing q_u.

Algorithm 1: Approximate Matching based on Neighborhood Join

```
Input: Transformed query $q'(Q_E \subseteq V_{q'}, p_r(x,y) \in P_r)$. Every entity query
       $q_e \in Q_E$ is associated with a set of entities $E_{q_e}$.
Result: Table $A$, where each row represents a set of connected entities.
1  $q_{center} \leftarrow ARGMIN\{eccentricity(q_i) : q_i \in Q_E\}$
2  $q_{filter} \leftarrow q_{center}$
3  $A \leftarrow E_{q_{center}}$
4  while $\exists q_e \in Q_E : \neg visited(q_e)$ do
5      $q_{neighbor} \leftarrow q_e \in Q_E$ obtained via DFS along $p_r$ from $q_{center}$
6      if $d_{q_{filter}}(q_{neighbor}) > k$ then
7          $q_{filter} \leftarrow q_{lastSeen}$, where $q_{lastSeen}$ is the one lastly seen along the path
           currently traversed via DFS
8      end
9      $A \leftarrow A \bowtie^{nb}_{E_{q_{filter}}} E_{q_{neighbor}}$
10 end
11 return $A$
```

5 Structure-based Result Refinement and Computation

Result of the previous step is a set of tuples. Every tuple is a set of entities that are somehow connected, i.e., connected over some unknown paths. During refinement, we want to find out whether they are really connected via paths captured by query atoms. For this, we propose the *structure-based result refinement*, which helps to refine the previous results by operating against a summary called the structure index. Using the summary, we check if tuples computed in the previous step match query relation paths. If so, the final step called *structure-based result computation* is performed on the refined tuples.

Structure Index for Graph Structured Data. Structure indexes have been widely used for semi-structured and XML data [4, 13, 6]. A well-known concept is the dataguide [9], which is a structural description for rooted data graphs. Dataguide nodes are created for groups of data nodes that share the same incoming edge-labeled paths starting from the root. Similar to this concept, a structure index has been proposed for general data graphs [16]. Nodes in a structure index stand for groups of data elements that have equal structural "neighborhood", where equal structural neighborhood is defined by the well-known notion of *bisimulation*. Accordingly, two graph nodes v_1, v_2 are *bisimilar* ($v_1 \sim v_2$), if they cannot be distinguished by looking only at their outgoing or incoming "edge-labeled trees". Pairwise bisimilar nodes form an extension. Applying the bisimulation $\sim$ to the graph $G(V, L, E)$ of our data graph that contains relation edges only, results in a set of such *extensions* $\{[v]^\sim \mid v \in V\}$ with $[v]^\sim := \{w \in V \mid v \sim w\}$. These extensions form a complete partition of the entity nodes V of the data graph, i.e., form a family $\mathcal{P}^\sim$ of pairwise disjoint sets whose union is V. Based on this notion of bisimulation, the *structure index graph* $G^\sim$ of $G(V, L, E)$ can be defined in terms of extensions and relations between them. In particular, extensions from the partition $\mathcal{P}^\sim$ form the vertices of $G^{\sim}$. An edge with label l links $E_1, E_2 \in \mathcal{P}^{\sim}$ of $G^{\sim}$ iff G contains an l-edge linking an element in the extension E_1 to some element in extension E_2.

Algorithm 2: Structure-based Result Refinement using Structure Index

```
   Input: Transformed query $q'(V_{q'}, p_r(q_s, q_t) \in P_r)$. Entity query nodes
          $Q_E \subseteq V_{q'}$. Table $A_{m \times n}(q_{e_1}, ..., q_{e_n})$, where each row represents a set of
          somehow connected entities. Structure index graph $G^{\sim}(V^{\sim}, E^{\sim})$.
   Data: $EXT_{q_e}(q_e, ext(q_e))$ is a two column table containing the results $e \in E_{q_e}$
         of $q_e$ and their extensions $ext(e)$. $E^{\sim}(source(r), target(r))$ is a two
         column table containing source and target nodes of the edge $r$.
   Result: Refined table of entities $A$. Intermediate result table $M(c_1, ..., c_n)$
           containing entities and entity extensions, where $c_n$ denotes a query $q_e$
           or an extension $ext(q_e)$.
1  for $p_r(q_1, q_2) \in P_r$ do
2      $E^{\sim}(e_{q_1}, e_{q_2}) \leftarrow \{r^{\sim}(x, y) \in E^{\sim} | p_r = r^{\sim}\}$
3      for $q_n \in \{q_1, q_2\}$ do
4          if $q_n \in Q_E$ then $E^{\sim}(e_{q_1}, e_{q_2}) \leftarrow E^{\sim}(e_{q_1}, e_{q_2}) \bowtie_{q_n} EXT_{q_n}$
5      end
6      if $M = \emptyset$ then $M = E^{\sim}(e_{q_1}, e_{q_2})$
7      else $M \leftarrow E^{\sim}(e_{q_1}, e_{q_2}) \bowtie_{q_n} M$
8      $A \leftarrow \pi_{q \in Q_E}(M)$
9  end
10 return $A$ and $M$
```

Example 4. The data graph shown in Fig. 1a can be partitioned into 8 extensions, shown as nodes of the index graph in Fig. 1b. For instance, $p1$ and $p3$ are grouped into the extension E2 because they are bisimilar, i.e., both have incoming *supervise* and *knows* links and both have the same outgoing trees (paths) of edges *knows*, $(worksAt, partOf)$ and $(authorOf, conference)$.

It has been shown that the structure index is appropriate for investigating structures that can be found in the data [16]. In particular, it exhibits a property that is particularly useful for our approach:

Property 1. If there is a match of a query graph on a data graph G, the query also matches on the index graph $G^{\sim}$. Moreover, nodes of the index graph matches will contain all data graph matches, i.e., the bindings to query variables.

Structure-based Result Refinement. Property 1 ensures that nodes of the index graph matches will contain all data graph matches, i.e., the bindings to query variables. Therefore, entities computed in the previous step can only be answers to the query, when they are contained by some matches of the query on the structure index graph. Drawing upon this observation, Alg. 2: (1) matches the transformed query graph q' against the structure index and (2) checks if the resulting index graph matches contain the previously computed entities in table A. For index graph matching, edges $E^{\sim}$ of the index graph are retrieved (line 2) and joined along the query atoms $p_r(q_1, q_2) \in P_r$. When entity query nodes are encountered, i.e., q_n is an element of Q_E, we check if entities previously computed for q_n (stored in A) are contained in the matching extensions retrieved for q_n. For this, we use the extensions associated with these entities (as stored in ENT-doc) to construct an extension table EXT_{q_n} and join this table with $E^{\sim}$. Thereby, extensions that do not contain entities in A are discarded during the computation. After processing all edges, M contains only index matches, which connect entities in A. Finally, by projecting on the attributes q_e, we obtain the refined entities A from M (line 8).

Example 5. This example demonstrates refining result table $A = \{(p1, i1, u1, c1), (p3, i1, u1, c1), (p5, i1, u1, c1)\}$. The result of the refinement step is one index match (Fig. 3b). To obtain the index match, we can, e.g., start with the query atom $supervise(w, q_x)$. For this, one matching edge $supervise^{\sim} = \{(E1, E2)\}$ is retrieved from $G^{\sim}$. $supervise^{\sim}$ is joined with the extension table for q_x, i.e., $\{(E1, E2)\} \bowtie_{q_x} \{(E2, p1), (E2, p3)\}$. This results in $supervise^{\sim} = \{(E1, E2, p1), (E1, E2, p3)\}$, i.e., extension $E2$ obtained for q_x contains entities $p1, p3$ (previously computed for q_x). Thus, no match is discarded in this case. We continue with $authorOf(q_x, y)$ to obtain $authorOf^{\sim} = \{(E6, E4), (E2, E4)\}$. By joining on q_x, i.e., $\{(E6, E4), (E2, E4)\} \bowtie_{q_x} \{(E2, p1), (E2, p3)\}$, we obtain $\{(E2, p1, E4), (E2, p3, E4)\}$, i.e., we discard the extension $E6$, as it does not contain $p1, p3$. Since y is not an entity query, we do no need to check if the extension $E4$ contains entities in A. Now, $M = authorOf^{\sim} \bowtie supervise^{\sim}$, i.e., $M = \{(E1, E2, p1), (E1, E2, p3)\} \bowtie_{q_x} \{(E2, p1, E4), (E2, p3, E4)\} = \{(E2, E2, p1, E4), (E1, E2, p3, E4)\}$. This process continues for the remaining query atoms to obtain $M = \{(E1, E2, p1, E4, E3, i1, E5, u1, E6, c1), (E1, E2, p3, E4, E3, i1, E5, u1, E6, c1)\}$. Projecting M on the attributes $q \in Q_E$ results in $A = \{(p1, i1, u1, c1), (p3, i1, u1, c1)\}$.

Complete Structure-based Result Computation. Finally, results which exactly match the query are computed by the last refinement. Only for this step, we actually perform joins on the data. To improve efficiency, we do not retrieve and join data along the query atoms in a standard way [1]. Instead, we incrementally refine the results, i.e., reuse the index matches and the entities associated with them as stored in the intermediate result set M. Given the index graph match $G^{\sim}_q$, the algorithm for result computation iterates through the edges $l^{\sim}_q([e_1]^{\sim}, [e_2]^{\sim}) \in L^{\sim}$ of $G^{\sim}_q$, retrieves matching triples, and joins them. However, if results exist, i.e., there are entities contained in $[e_1]^{\sim}$ or $[e_2]^{\sim}$ such that $[e_1]^{\sim}.E \vee [e_2]^{\sim}.E \neq \emptyset$, they are taken into account. In particular, only triples $l^m_q(e_1, e_2)$, where $e_1 \in [e_1]^{\sim}.E$ and $e_2 \in [e_2]^{\sim}.E$ are retrieved from the data graph. In Fig. 3c, we see the triples that are retrieved and joined to obtain the final result of the query. Only by inspecting the actual triples along this structure index match, we note that $p3$ is not connected with the other entities.

6 Evaluation

We conducted a complexity analysis for our approach. Given a query graph with bounded size, we can prove that the complexity of query processing is polynomial, which is more promising than the worst-case exponential complexity of

	Data(#Edges)	Data(MB)	EntityIdx(MB)	RelIdx(MB)	StrucIdx(KB)	Schema(KB)
DBLP	12,920,826	2,084	2210	2,311	132	28
LUBM5	722,987	122	142	112	100	24
LUBM10	1,272,609	215	253	198	80	24
LUBM50	6,654,596	1,132	1391	1,037	82	24

Table 2. Statistics for the data graphs and indexes.

exact and complete graph-pattern matching. Due to space reasons, the details were omitted here but can be found in our technical report.[5] In this section, we present empirical performance results and also analyze the efficiency-precision trade-off to shed light on the incremental and approximate features of our approach.

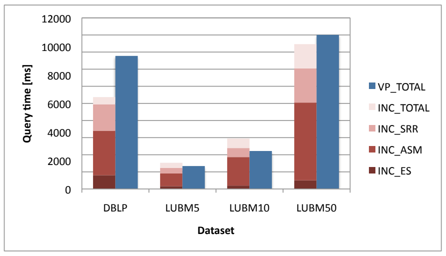
Systems. We have implemented the incremental process (INC) based on vertical partitioning and sextuple indexing [1, 17]. To compare our solution with the exact and complete approach [1], we implement sorted-merged equi-join using the same data partitions and indexes (VP). Since query optimization as proposed for the RDF-3X [14] is orthogonal, all experiments here were performed without optimization, i.e., based on fixed query plans (same for both approaches). There is no appropriate baseline for the approximate and incremental features of our solution. ASM is based on Bloom filter, which has not been applied to this problem of structure matching before. Also, there is no alternative for SRR. We have already pointed out (related work) that, while SRR is based on a summary, which is conceptually similar to the synopsis previously proposed for approximate query processing, it is not clear how to extend these concepts to graph-structured data and in particular, to use them in a pipeline. Our implementation is freely available.[6]

Datasets. We used *DBLP*, which captures bibliographic information. Further, we used the LUBM data generator to create 3 datasets for 5, 10 and 50 universities (Table 2). Note that the structure indexes were consistently bigger than the schemas, but were of magnitudes smaller than the data graphs.

Queries. For studying the proposed algorithms in a principled way, test queries were generated via random data sampling. We generated queries ranging from simple path-shaped to graph-shaped queries. For this, we use as parameters the maximum number of constants con_{max}, the maximum number of paths p_{max}, the maximum path length l_{max} and the maximum number of cycles cyc_{max} in the query graph. We sampled constants from data values V_D of the data graph. Paths and cycles were sampled from data graph edges E. The parameters used in the experiments are $con_{max} = 20$, $p_{max} = 6$, $l_{max} = 3$, $cyc_{max} = 2$.

Setting. We used a machine with two Intel Xeon Dual Core 2.33 GHz processors and 48GB of main memory running Linux (2GB were allocated to JVM). All data and indexes were stored on a Samsung SpinPoint S250 200GB, SATA II. All components have been implemented in Java 5. The bit-vector length and the number of hash functions used for Bloom filter encoding were computed

[5] `http://people.aifb.kit.edu/awa/ApproxIncrementalQueryProcessing.pdf`
[6] `http://code.google.com/p/rdfstores/`



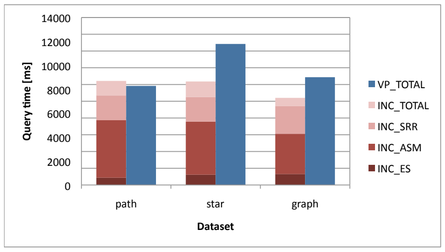


Fig. 4. Query processing times for a) different datasets and b) different query shapes.

to reach the configured probability of false positive of 0.1%. Neighborhood indexes were created for $k = 3$. All times represent the average of 10 runs of 80 queries generated for DBLP, and 80 queries for LUBM. For different steps of INC, we computed the precision using the formula: precision = (|correct results| ∩ |results retrieved|)/|results retrieved|. A result of an entity query in ES is correct, if it is contained in the respective column of the final result table. The precision for ES is computed as the average precision obtained for all entity query nodes of q. A tuple computed during ASM and SRR is correct, if it is contained as a row in the final result table.

Average Processing Time. For INC, we decomposed total processing time into times for ES, ASM, SRR and SRC. Averaging the processing time over 80 queries, we obtained the results shown in Fig. 4a. The time needed for ES is only a small fraction of the total time. Times for SRR and SRC make up a greater portion, and ASM constitutes the largest share. Thus, these results suggest that users can obtain an initial set of results in a small fraction of time via ES. In particular, instead of waiting for all exact results, users might spend only 6, 71 or 84 percent of that times when they choose to finish after ES, ASM or SRR respectively. The comparison of total times shows that INC was slower than VP for LUBM5 and LUBM10, but faster for the larger datasets LUBM50 and DBLP. While these results might change with query optimization, this promising performance indicates that our incremental approach was able to effectively reuse intermediate results.

The Effect of Data Size. We have measured total time for LUBM of different data sizes (shown in Table 2). As illustrated in Fig. 4a, query processing time increased linearly with the size of the data, for both VP and INC. Further, INC became relatively more efficient as the data size increased. It can be observed that the share of total time from ASM decreased with the data size, i.e., the gain from ASM unfolded as the dataset grew larger. This is particularly important in the Data Web context; ASM can help to quickly obtain initial results from a large amount of data.

The Effect of Query Complexity. Considering query complexity, we classified the 80 queries into three classes according to query shape. As shown in Fig. 4b, INC did not perform well on path queries. For this type of queries, ASM was particularly expensive. This is because in many cases, the reusability of Bloom filters was low (i.e., when path length was higher than k). Filter loading and nested loop joins became the bottleneck, resulting in slightly higher processing times compared to VP.

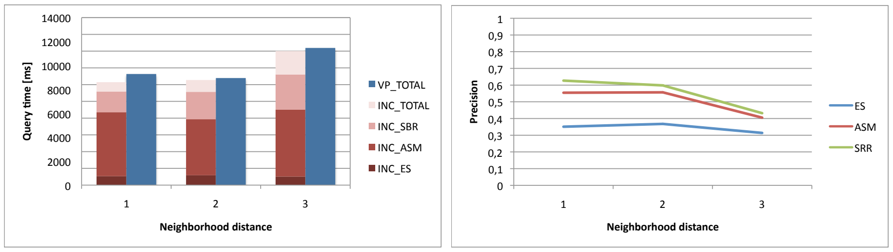


Fig. 5. Effect of neighborhood distance on a) processing times and b) precision.

The Effect of Relation Path Length k**.** In another experiment we classified queries into three classes according to the length of the longest relation path (i.e., the neighborhood distance between entities, respectively). As shown in Fig. 5a, queries with longer relation paths required more time, for both VP and INC. For INC, the share contributed by ASM remained relatively constant, suggesting that this step can be performed efficiently even for long relation paths. Thus, ASM can also help to deal with complex queries with long relation paths.

Precision. The average precision for the different steps at various k is shown in Fig. 5b. The precision for ES was relatively constant (0.3 - 0.4). This was expected, because k should have no effect on the quality of entity search. For ASM and SRR, precision decreased with larger k. The highest precision obtained for ASM was 0.56 and this increased to 0.62 after SRR.

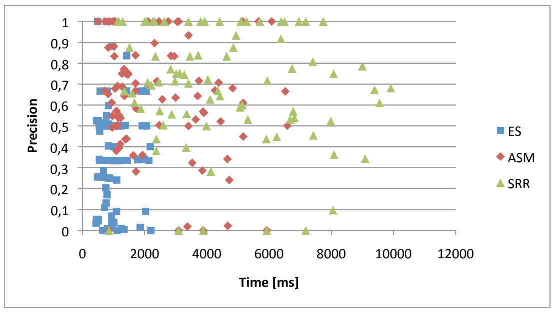


Fig. 6. Precision vs. time.

Time-Precision Trade-off. We illustrate average time and precision for different steps in Fig. 6. Clearly, through the incremental refinement steps, both precision and processing times increased. There are some outliers – however, overall, a trend may be noticed: ES produces fast results at low precision, i.e., below 50 % for most cases. Precision can be largely improved through ASM, i.e., in 30 % of the cases, ASM drove precision from 50 % up to 80 %. For most of these cases (60 %), the amount of additional processing was less than 10 % of total time.

7 Conclusion and Future Work

We proposed a novel process for approximate and incremental processing of complex graph pattern queries. Experiments suggest that our approach is relatively fast w.r.t exact and complete results, indicating that the proposed mechanism for incremental processing is able to reuse intermediate results. Moreover, promising results may be observed for the approximate feature of our solution. Initial results could be computed in a small fraction of total time and can be refined via approximate matching at low cost, i.e., a small amount of additional time. We believe that our approach represents the appropriate paradigm, and embodies essential concepts for dealing with query processing on the Web of data, where responsiveness is crucial. At any time, users should be able to decide if and for which results exactness and completeness is desirable. As future work, we will elaborate on ranking schemes, based on which we plan to integrate top-k techniques into the pipeline.

Acknowledgements. Research reported in this paper was supported by the German Federal Ministry of Education and Research (BMBF) in the CollabCloud (grant 01IS0937A-E) and iGreen (grant 01IA08005K) projects.

References

1. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.
2. B. Babcock, S. Chaudhuri, and G. Das. Dynamic Sample Selection for Approximate Query Processing. In *SIGMOD Conference*, pages 539–550, 2003.
3. B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.
4. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, pages 336–350. Springer, 1997.
5. K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB*, pages 111–122, 2000.
6. Q. Chen, A. Lim, and K. W. Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144. ACM, 2003.
7. O. Corby, R. Dieng-Kuntz, C. Faron-Zucker, and F. L. Gandon. Searching the Semantic Web: Approximate Query Processing Based on Ontologies. *IEEE Intelligent Systems*, 21(1):20–27, 2006.
8. M. N. Garofalakis and P. B. Gibbons. Approximate Query Processing: Taming the TeraBytes. In *VLDB*, 2001.
9. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.
10. A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB*, 2005.
11. C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking Approximate Answers to Semantic Web Queries. In *ESWC*, pages 263–277, 2009.
12. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40:11:1–11:58, 2008.
13. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.
14. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
15. N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate xml query answers. In *SIGMOD*, SIGMOD '04, pages 263–274, New York, NY, USA, 2004. ACM.
16. D. T. Tran and G. Ladwig. Structure index for RDF data. In *Workshop on Semantic Data Management at VLDB*, September 2010.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
18. L. Zhang, Q. Liu, J. Zhang, H. Wang, Y. Pan, and Y. Yu. Semplore: An IR Approach to Scalable Hybrid Query of Semantic Web Data. In *ISWC/ASWC*, 2007.