Structure Index for RDF Data

Thanh Tran
Institute AIFB
Karlsruhe Institute of Technology (KIT)
76128 Karlsruhe, Germany
ducthanh.tran@kit.edu

Günter Ladwig
Institute AIFB
Karlsruhe Institute of Technology (KIT)
76128 Karlsruhe, Germany
guenter.ladwig@kit.edu

ABSTRACT

In recent years, the amount of structured RDF data available on the Web has been increasing rapidly. Efficient query processing that can scale to large amounts of RDF data has become an important topic. Significant efforts have been dedicated to the development of solutions for RDF data management. Along this line of research, we elaborate on a novel data partitioning strategy, which leverages the structure of the underlying data. This structure is represented in form of a parameterized structure index we propose for (RDF) data graphs called PIG. It is not only used for data partitioning but also has been designed to accelerate the matching of graph-structured queries against RDF data. In our benchmark against state-of-the-art techniques, our structure-based approach for partitioning and query processing exhibits 7-8 times faster performance.

INTRODUCTION

In recent years, the amount of structured data available on the Web has been increasing rapidly, especially *RDF data* consisting of triples of the form $\langle s, p, o \rangle$, where s is the subject, p is a property, and o is the object. Such triples form a *data graph* $G(V, L, E)$ where the vertices V denote resources and their values, which are connected by directed edges E, each endowed with a label from a label set L. One example is shown in Fig. 1.

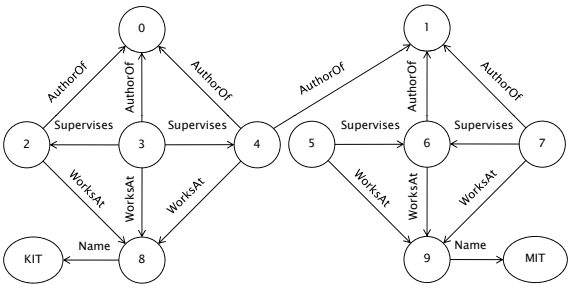

Figure 1: A data graph

Workshop on Semantic Data Management (SemData@VLDB) 2010, September 17, 2010, Singapore.

This development of a *data web* opens new ways for addressing complex information needs. Search is no longer limited to matching keywords against documents, but instead, structured queries can be processed against web resources. In this regard, *conjunctive queries* represent an important fragment of widely used languages (SQL, SPARQL), which has been a focus of recent work on RDF data management [1, 10, 6]. Essentially, a query of this type consists of a set of triple patterns of the form $p(s, o)$, where p is a predicate and s and o are variables (Var_q) or constants (Con_q). Intuitively speaking, variables appearing in the SELECT clause are called distinguished variables (Var_q^d), otherwise undistinguished variables (Var_q^u). Triple patterns constitute a query graph, as illustrated in Fig. 2b.

A *match* of a conjunctive query q on a graph G is a mapping h from the variables of q to vertices of G such that the according substitution of variables in the graph-representation of q would yield a subgraph of G. Therefore a query match h can be interpreted as a certain type of homomorphism (i.e. a structure preserving mapping) from the "query graph" to the data graph. Because the amount of data is enormous and largely increasing, *scalability* of this graph pattern matching on the data web has become a key issue.

State-of-the-art in RDF Management Triple stores developed for storing and querying RDF data such as Hexastore, RDF-3X and SW-Store, can be distinguished along three dimensions: data organization in terms of (1) partitioning and (2) indexing, and (3) query processing. *Vertical partitioning* has been proposed to decompose the data graph into n two-columns tables, where n is number of properties. This has shown to be superior than the "one single giant table" organization, and the use of property tables [1]. The strategy used for indexing is based on the coverage of multiple lookup patterns [6]. In [10], *sextuple indexing* has been suggested, which generalizes the work in [6, 1] such that for different access patterns, retrieved data comes in a sorted fashion. This greatly accelerates query processing, as fast (near linear) merge joins originally proposed in [1], can now be performed for many more combinations of triple patterns. Further efficiency gains can be achieved by finding an optimal query plan [8].

Overview and Main Contributions We elaborate on concepts that improve the state-of-the-art in data partitioning and query processing:

- *Parameterized Structure Index for RDF Data:* Generalizing work on XML data such as dataguide [5], we propose an index called PIG that summarizes the structure of general graph structured data like RDF. The size of this index can be controlled by means of parameters (e.g. derived from workload).

- *Structure-based Partitioning:* Based on PIG, we propose a

structure-based partitioning scheme, where triples about elements with the same structure are physically grouped. This is to obtain a contiguous storage of data that likely co-occurs in query answers.

- *Structure-aware Query Processing:* We propose to match the query against the structure index first, which is typically much smaller than the data graph (c.f. examples in Fig. 2). This helps to focus on data that satisfy the overall structure of the query and on this basis, to proceed with standard processing at the level of the data for only certain parts of the query.

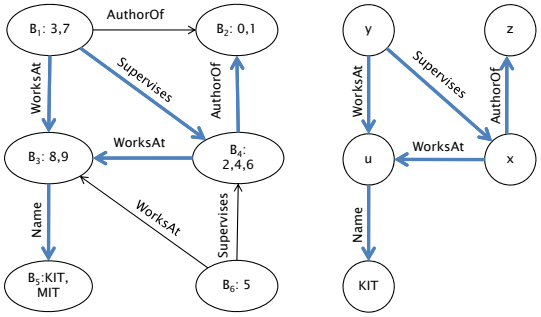

Figure 2: (a) An index graph (b) a query graph

Our solution is complementary to the concepts for indexing and query optimization [10, 8], and offers the following additional benefits:

- *Reduction of I/O Costs:* We do not simply retrieve all data that matches some given triple patterns but focus on the one that satisfies the entire query structure.
- *Reduction of Union and Joins:* These operations are only needed only for some parts of the query. In the extreme cases where no structure index matches can be found, we can skip data access and joins at the data level completely.

In a benchmark against the state-of-the-art techniques for data partitioning and query processing used in SW-Store [1], our approach is 7-8 times faster for a PIG that is parameterized according to the query workload.

Outline We introduce PIG in Section 2. Partitioning, query processing and parameterization are discussed in Section 3, 4 and 5. Experiments along with results are discussed in Section 6 before we review related work in Section 7 and conclude in Section 8. For more details, we refer the interest readers to our technical report [2].

STRUCTURE INDEX FOR RDF

We introduce the notion of a parameterizable structure index for general graph-structured data (like RDF) called PIG.

PIG - Parameterizable Index Graph PIG is a special graph forming a compact representation of the data graph, whose vertices stand for groups of data graph elements that have a similar or equal structural "neighborhood". We capture the concept of equal structural neighborhood by the well-known notion of *bisimulation* originating from the theoretical analysis of state-based dynamic systems. We adopt this notion to capture both directions of edges. We consider graph nodes v_1, v_2 as *bisimilar* (written: $v_1 \sim v_2$), if they cannot be distinguished by looking only at their outgoing (forward bisimilarity) and incoming (backward bisimilarity) "edge trees". For better control, we parameterize our notion of bisimulation by two sets of edge labels L_1 and L_2 that specify the labels of edges taken into account for determining the forward and backward bisimulation respectively:

DEFINITION 1. *Given a data graph $G = (V, L, E)$ and two edge label sets $L_1, L_2 \subseteq L$, a L_1-forward-L_2-backward* bisimulation *on G is a binary relation $R \subseteq V \times V$ on the vertices of G such that for $v, w \in V$, $l_1 \in L_1$ and $l_2 \in L_2$:*

- *vRw and $l_1(v, v') \in E$ implies that there is a $w' \in V$ with $l_1(w, w') \in E$ and $v'Rw'$,*
- *vRw and $l_1(w, w') \in E$ implies that there is a $v' \in V$ with $l_1(v, v') \in E$ and $v'Rw'$,*
- *vRw and $l_2(v', v) \in E$ implies that there is a $w' \in V$ with $l_2(w', w) \in E$ and $v'Rw'$,*
- *vRw and $l_2(w', w) \in E$ implies that there is a $v' \in V$ with $l_2(v', v) \in E$ and $v'Rw'$.*

Two vertices v, w will be called bisimilar (written $v \sim w$), if there exists a bisimulation R with vRw.

The relation $\sim$ is itself a bisimulation and can be represented by the set of its equivalence classes called *extensions*, where all vertices contained in one extension are pairwise bisimilar. These extensions form a partition of the vertices V of the data graph, i.e. a family $\mathcal{P}^{\sim}$ of pairwise disjoint sets whose union is V. The *index graph* $G^{\sim}$ of G is defined in terms of extensions and relations between them:

DEFINITION 2. *Let $G = (V, L, E)$ be a data graph and $\sim$ its L_1-L_2-bisimulation. Vertices of the associated* index graph *$G^{\sim} = (V^{\sim}, L_1 \cup L_2, E^{\sim})$ are exactly G's $\sim$-equivalence classes $V^{\sim} = \{[v]^{\sim} \mid v \in V\}$, with $[v]^{\sim} = \{w \in V \mid v \sim w\}$. Labels of $G^{\sim}$ are exactly the labels of G. An edge with a label e is established between two equivalence classes $[v]^{\sim}$ and $[w]^{\sim}$ exactly if there are two vertices $v^* \in [v]^{\sim}$ and $w^* \in [w]^{\sim}$ such that there is an edge $e(v^*, w^*)$ in the data graph, i.e. $E^{\sim} = \{e([v^*]^{\sim}, [w^*]^{\sim}) \mid e(v^*, w^*) \in E\}$.*

EXAMPLE 1. *The index graph associated with the data graph in Fig. 1 is depicted in Fig. 2a, for $L_1 = \{WorksAt, Supervises, AuthorOf\}$ and $L_2 = \{WorksAt, AuthorOf\}$. Elements that are indistinguishable in terms of their edge-labelled, incoming and outgoing paths are for instance 2, 4, and 6. Thus, they are elements of the extension B_4.*

Construction of PIG First, a bismulation for L_1 and L_2 is calculated, using an adapted version of the algorithm for determining the coarsest stable refinement of a partitioning [9]. The algorithm starts with a partition consisting of a single block that contains all data, and splits into smaller blocks until the partition is a forward bisimulation. In order to perform both backward and forward bisimulation for only the parameters L_1 and L_2, we essentially exploit the observation that L_1-forward-L_2-backward bisimulation on a data graph $G = (V, L, E)$ coincide with forward bisimulation on an altered data graph $G_{L_1L_2} = (V, L_1 \cup \{l^- \mid l \in L_2\}, E_{L_1L_2}\})$ where $E_{L_1L_2} = \{l(x, y) \mid l(x, y) \in E, l \in L_1\} \cup \{l^-(y, x) \mid l(x, y) \in E, l \in L_2\}$. After having determined the bisimulation, the resulting blocks from the partition $\mathcal{P}^{\sim}$ are used to form vertices in the index graph according to Definition 1.

Structured-based Partitioning Clearly, whether a graph vertex instantiates a variable of a query obviously depends on its structural properties, i.e. the incoming and outgoing edges resp. paths. Therefore, if nodes are physically grouped together based on structural similarity, a group would contain more candidates for variable instantiations. Thus, we apply structure-based partitioning to the data graph by creating a physical group (e.g. a table) for every vertex of the index graph, i.e. one group for every extension. Every group contains the triples, which "describe" elements contained in the corresponding extension. That is, triples are in the same group when they contain the same properties and their subjects belong to the same extension.

RDF QUERY PROCESSING

Query processing in our scenario is essentially finding a homomorphism from the query graph $q = (V_{var} \uplus V_{con}, L, P)$ to elements of the data graph $G = (V, L, E)$. According to the following proposition, the structure index can be exploited to perform this task:

PROPOSITION 1. *Let G be a data graph with associated index graph $G^\sim$ and let q be another graph such that there is a homomorphism h from q into G. Then $h^\sim$ with $h^\sim(v) = [h(v)]^\sim$ is a homomorphism from q into $G^\sim$.*

Proof Given a q-edge $l(v_1, v_2)$, first observe that $l(h(v_1), h(v_2))$ is a G-edge since h is a homomorphism. Then we find that $l(h^\sim(v_1), h^\sim(v_2)) = l([h(v_1)]^\sim, [h(v_2)]^\sim)$ is a $G^\sim$-edge due to the definition of the index graph. □

Intuitively speaking, a mapping into G exists only if it does also exist into the associated index graph $G^\sim$. Further, the resulting extensions $B_i = [h(v)]^\sim$ from $V^\sim$ in $G^\sim$ that match the nodes in q will obviously contain the data graph matches $h(v)$. Thus, the procedure for query processing can be decomposed into two steps: (1) finding matching extensions B_i on the index graph $G^\sim$ first, (2) then combining data elements retrieved for B_i to obtain the final data graph matches. In the following, we denote $G^\sim Idx$ as the index used for the retrieval of elements from the index graph and $GIdx$ is used for accessing elements of the data graph.

Index Graph Matches

Just like an answer, an index graph match is the result of a homomorphic mapping $h^\sim$ from the query graph $q(V_q, L_q, P)$ to the index graph $G^\sim(V^\sim, L, E^\sim)$. Elements of an index graph match are vertices of $G^\sim$ that are assigned to variables and constants of q. For this computation, we propose a join procedure that returns a result table R containing all matches $h^\sim : V_q \to V^\sim$. First, a set of index graph candidate edges $E_l^\sim$ is retrieved from $G^\sim$ for every query edge label l occurring in the query (using $G^\sim Idx$). Then, these candidate sets are joined along the vertices of q to obtain R. Figure 3 illustrates two matches.

Data Matches for an Index Match

The previous computation results in a set R of index graph matches $h^\sim : V_q \to V^\sim$. Every element of these matches is an extension which essentially is a set of vertices of the queried data graph G. According to Proposition 1, every match of the query against the data graph is "contained" in one of the index graph matches calculated so far (e.g. $h(v_1)$ is in $h^\sim(v_1) = [h(v_1)]^\sim$). It suffices to focus on the index graph matches for the computation of the data graph matches because only data contained by them satisfy the overall query structure. We will now show that tree-shaped parts containing only undistinguished variables can even be pruned

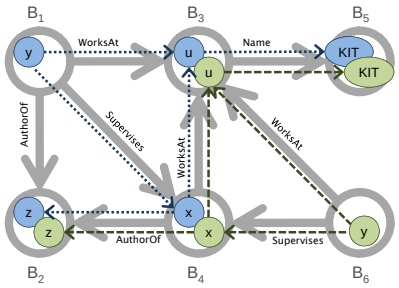

Figure 3: Two index matches $h_1^\sim$ and $h_2^\sim$

away entirely. We now inductively define this notion of tree-shaped query part:

DEFINITION 3. *Every edgeless single-vertex rooted graph (G, r) with $G = (\{r\}, L, \emptyset)$ is L_1-forward-L_2-backward tree-shaped. Moreover let (G_1, r_1) and (G_2, r_2) with $G_1 = (V_1, L, E_1)$ and $G_2 = (V_2, L, E_2)$ be two L_1-forward-L_2-backward tree-shaped graphs with disjoint vertex sets. Let $v \in V_1$ and let $e \in \{l(v, r_2) \mid l \in L_1\} \cup \{l(r_2, v) \mid l \in L_2\}$. Then the rooted graph (G_3, r_1) with $G_3 = (V_1 \cup V_2, L, E_1 \cup E_2 \cup \{e\})$ is L_1-forward-L_2-backward tree shaped.*

Given such a tree-shaped query part, a stronger property can be asserted for the index graph matches, i.e. they contain all and only data graph matches such that no further processing at the level of the data graph is needed:

PROPOSITION 2. *Let G be a data graph with associated index graph $G^\sim$ (where $\sim$ is a L_1-forward-L_2-backward bisimulation) and let q be a L_1-forward-L_2-backward tree-shaped graph with root r. Let h' be a homomorphism from q to $G^\sim$. Then for every data graph node $v \in h'(r)$, there is a homomorphism h from q to G with $h(r) = v$*

Proof. We do an induction on the maximal tree-depth of G'. As base case, note that for tree depth 0, the claim is trivially satisfied. For any greater tree depth, subsequently consider every child node v' of r in G'. Assume $l \in L_1$ and $l(r, v') \in E'$ (the backward case follows by symmetry). Then, by definition of bisimularity, there must exist a $w \in h(v')$ with $l(v, w) \in E$. We chose $h(v') = w$. Now we invoke the induction hypothesis for the subtree of G' with root v' which yields us the values $h(u')$ for all successors of v'. So we have constructed h with the desired properties. □

In words: if the query is of the aforementioned tree shape, then every data node from any extension associated to the query root r by an index graph match is a data graph match for r. Hence, before computing data graph matches, the respective query parts can be removed.

After pruning the query, we use another join procedure to compute a result table where rows capture bindings to distinguished query variables. These bindings are data elements contained in the index graph matches $h^\sim$, which satisfy the structure as well as the concrete elements (i.e. constants and distinguished variables) mentioned in the query. Query edges are processed successively. At every iteration, triples are retrieved from $GIdx$ and are joined with the (intermediate) results set. More precisely, given the query edge $p(x, y)$, the triples $\langle x \mapsto s, p, y \mapsto o \rangle$ matching $p(x, y)$ are considered. They are fetched from the corresponding block $[s]^\sim$ of the structure-based partitioned data graph index $GIdx$, where $s \in [s]^\sim$ and $h^\sim : x \to [s]^\sim$. Intuitively speaking, only triples with subjects that are contained in the index match $[s]^\sim$ are retrieved from disk.

Thus, only subjects that are known to satisfy the query structure are considered. This is different from the standard approaches [1, 8], where all triples matching the query edge are taken into account, which might contain subjects not in $[s]^{\sim}$.

EXAMPLE 2. *Fig. 2b depicts a query, which asks for authors x working at the same place as their supervisors y, namely a place called KIT. One match on the index graph is $h_1^{\sim} = \{u \mapsto B3, x \mapsto B4, y \mapsto B1, z \mapsto B2, KIT \mapsto B5\}$. Based on this, we know that data elements belonging to extensions obtained from the index graph match satisfy the query structure, e.g. elements in $B4$ are authors of z, supervised by y, work at some place u that has a name. A tree-like part that can be pruned is $AuthorOf(x, z)$. It produces the index graph match $\langle B4\ AuthorOf\ B2\rangle$. Since $2, 4$, and 6 in $B4$ are already known to be authors of some z, no further data processing is needed for this query part. However, we have to look at the data to verify that elements in $B4$ work at KIT, and are supervised by some y also working at KIT. For this, we need to retrieve and join the triple matches for $\langle x\ WorksAt\ u\rangle$, $\langle y\ WorksAt\ u\rangle$, $\langle u\ Name\ KIT\rangle$, $\langle y\ Supervises\ x\rangle$. Note that the query example here contains cycles. In practice, there are many queries exhibiting simpler structure, which offer greater potential for query pruning. In the extreme cases where no index graph matches can be found, we can skip the entire second step to avoid data access and joins completely.*

Data Matches for Index Match Classes

The procedure presented in the previous section computes answers data matches for an index graph match $h^{\sim}$. In order to compute all data matches, this has to be repeated for all index graph matches $h^{\sim}$ in R. However, the diverse matches might partially overlap. To formalize and computationally exploit this, we introduce the following notion:

DEFINITION 4. *Let G be a data graph and $G^{\sim}$ be an index graph. For a query $q(V_q, L_q, P)$, let $P' \subseteq P$ be a subset of the query atoms. We say that two matches $h_1^{\sim}$ and $h_2^{\sim}$ from q into $G^{\sim}$ are in the same P'-*match class *if for all $l(v, w) \in P'$ $h_1^{\sim}(v) = h_2^{\sim}(v)$ and $h_1^{\sim}(w) = h_2^{\sim}(w)$. The set of* partial data matches *of a P'-match class M consists of all the functions $\pi : vertices(P') \to V$ satisfying*

1. *$l(\pi(v), \pi(w)) \in E$ for all $l(v, w) \in P'$ and*
2. *there is a $h^{\sim} \in M$ such that for all $v \in vertices(P')$ we have $[\pi(v)]^{\sim} = h^{\sim}(v)$.*

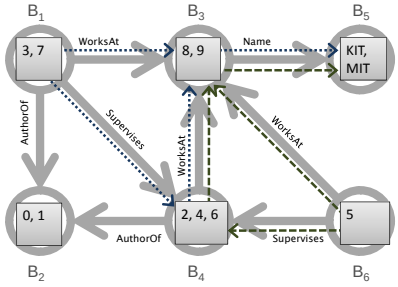

Figure 4: Data graph matching for $q_{pruned}, h_1^{\sim}, h_2^{\sim}$.

Fig. 4 displays the case for our example, where $P' = P \setminus \{AuthorOf(x, z)\}$, i.e. the two matches $h_1^{\sim}, h_2^{\sim}$ overlap on all query edges (resp. atoms) except $AuthorOf(x, z)$. Given such an equivalence between matches, the aforementioned naïve strategy for the independent evaluation of matches might result in many redundant disk accesses and join operations. The following proposition enables a more efficient computation of data graph matches:

PROPOSITION 3. *Let G be a data graph, let $G^{\sim}$ be the associated index graph, and let q be a conjunctive query.*

1. *Every match of $q(V_q, L_q, P)$ into G is contained in the set of partial data matches associated to one of the P-match classes.*
2. *Let $P' \subseteq P$ and $l(v, w) \in P \setminus P'$. For every $P' \cup \{l(v, w)\}$-match class M with a corresponding partial data match π, there is a P'-match class M' with corresponding data match π' such that there is a $h'^{\sim} \in M'$ with*
   - *$M = \{h^{\sim} \mid h^{\sim}(v) = h'^{\sim}(v)$ and $h^{\sim}(w) = h'^{\sim}(w)\}$,*
   - *$\pi \in \{\pi'\} \bowtie \{\{v \mapsto v_1, w \mapsto v_2\} \mid v_1 \in h'^{\sim}(v), v_2 \in h'^{\sim}(w), l(v_1, v_2) \in E\}$.*

Proof. The first part of the proposition is an immediate consequence of Proposition 1 and Definition 4. For the proposition's second part, the claim follows from choosing $M' = \{h^{\sim} \mid h^{\sim}(v) = h'^{\sim}(v)$ for all $v \in vertices(P')$ and some $h'^{\sim} \in M\}$ and $\pi' : v \mapsto \pi(v)$. □

In words, the preceding proposition ensures that all data graph matches of a query can be obtained by a successive refinement of match classes and their associated data matches. Consequently, the optimized procedure for computing query data matches consists of two main parts: (1) update of match classes and (2) evaluation of match classes.

Match classes are defined w.r.t. query vertices. For the first part, match classes are thus created (updated) according to the query vertices that are added during the process of join processing. At first, there is only one initial match class R consisting of all index graph matches (line 1). During the processing of query atoms $p(x, y)$, the set of classes MC becomes more and more "fine-grained", as any matches not coinciding on how x and y are mapped to $V^{\sim}$ will be distributed to different match classes (line 11). A hash map H, which associates pairs of index matches (x-y-instantiations) with match classes, is employed to check for overlaps. Note that during the processing of the atoms in P, the number of classes grows as high as the number of matches, i.e. every match constitutes its own class.

For answer computation, results are retrieved and stored in A_i for every match class M_i encountered during processing of atoms in P. Since every class groups together matches that overlap w.r.t. the already considered part of the query, all matches are evaluated only once during query processing. After all atoms have been evaluated for all match classes, intermediate data matches for each match class are combined to obtain the final data matches (line 20).

OPTIMALLY PARAMETERIZED PIG

We briefly discuss the complexity of query processing and upon the results, elaborate on the parametrization of PIG.

Complexity For RDF query processing, we have $O(edgemax^{|P|})$, where $|P|$ denotes the number of query atoms and $edgemax$ is $|l(v_1, v_2)|$ with l being the property instantiated by the largest number of edges, e.g. the size of the largest table used in [1]. This is due to multiplicative cost of join processing in the general case – but the use of special indexes [10] gives us near-linear behavior in practice. Compared to this, the complexity of our approach is as follows:

PROPOSITION 4. *For a query with $|P|$ atoms, the time and space for calculating the answers through the computation of index matches and the subsequent processing of data graph matches is bounded by $O(edgemaxidx^{|P|} + |R| \cdot edgemaxdata^{|P_{pruned}|})$.*

Algorithm 1: Evaluating Match Classes

Input: $G(V, L, E)$, $q(V_q, L_q, P)$, Index matches $R = \{h_1^\sim, h_2^\sim, \ldots, h_n^\sim : V_q \to V^\sim\}$.
Data: Two-columns table E_p containing matches $h'^\sim : \{x, y\} \to V$ retrieved for a predicate p, a set of match classes $MC = \{M_1, M_2, \ldots\}$ with $M_i \subseteq R$, intermediate partial matches A_i for every match class M_i, partial hash map $H : V^\sim \times V^\sim \to MC$ mapping pairs of index matches to match classes.
Result: Table A, where each row represents an answer.

```
1  set MC = {R}.
2  foreach p(x, y) ∈ P do
3      foreach M_i ∈ MC do
4          H ← ∅
5          foreach h^~ ∈ M_i do
6              if H contains (h^~(x), h^~(y)) as key then
7                  move h^~ from M_i to the match class
                   H(h^~(x), h^~(y)).
8              end
9              else
10                 create, add new match class M_j to MC.
11                 move h^~ from M_i to M_j.
12                 add (h^~(x), h^~(y)) ↦ M_j to H.
13                 E_p ←GIdx.retrieve(h^~(x), p, x, y).
14                 A_j ← A_i ⋈ E_p.
15             end
16         end
17     end
18     remove M_i from MC.
19 end
20 A ← Union of all A_i associated to M_i ∈ MC.
21 restrict A to the distinguished variables of q.
22 return A.
```

This is because our approach involves an additional component, i.e. the cost $O(edgemaxidx^{|P|})$ for computing index matches, where $edgemaxidx = \max_{l \in L} |E_l^\sim|$ is bounded by the size of the index graph. However, we need only to join along a pruned version $P_{pruned} \subseteq P$ of the query. So we obtain $O(|R| \cdot edgemaxdata^{|P_{pruned}|})$ where $edgemaxdata = \max_{B_1, B_2 \in V^\sim, l \in L} |\{l(v, v') \mid v \in B_1, v' \in B_2\}|$. Thus, $edgemaxdata$ is bounded by the size of the largest extension.

Parametrization of PIG In comparison, less data have to be retrieve from G, i.e. $edgemaxdata \leq edgemax$, and also, fewer joins are required, i.e. $|P_{pruned}|$ instead of P. The overhead introduced to achieve this $O(edgemaxidx^{|P|})$. The parametrization of L_1 and L_2 has the following effect on the overhead and gain introduced by our structure-based query processing:

When more labels are included in L_1 and L_2, the index graph becomes larger. Thus, $edgemaxidx$ becomes larger. On the other hand, more labels can be considered for query pruning, thus potentially increasing the fraction of prunable query parts ($|P \setminus P_{pruned}|$). Also, the physical groups obtained from structure-based partitioning become more fine-grained, thereby reducing the size of $edgemaxdata$.

Thus $L1$ and $L2$ shall be parameterized w.r.t the query workload to keep the index as small as possible. As a general strategy, the parametrization shall be computed based on frequent prunable query parts, i.e. search for prunable query parts in the workload and then, use labels occurring in these parts as parameters.

EVALUATION

More optimized systems have been built that implement the concepts of indexing and query optimization [8, 10]. Since these aspects are orthogonal, we use the work in [1] as baseline, which is the state-of-the-art in, and is purely focused on partitioning and query processing. We compare our work called structure-based query processing

We now summarize the experiment reported in details in [2]. It is based on DBLP and several synthetic datasets containing several millions of triples created using the Lehigh University Benchmark. A set of 30 queries categorized into five classes ranging from single-atom query to complex structured graph-shaped queries has been used. We use two parameterizations for the experiments: (1) SPB is based on G'_B calculated using backward bisimulation only and (2) SPFB uses G'_{FB}, a restricted back- and forward bisimulation adapted to the workload by setting L_1, L_2 to include only labels occurring in prunable query parts. G'_{FB} is much smaller (4%-30%) and the indexes for G'_B makes up only a small percentage (0.08%-2%) of the data graph.

Performance In Fig. 5a+b, total query processing time is plotted for both approaches. Clearly, SQP is faster than VPQP. For DBLP in particular, SQP performs a factor of 7-8 faster. Further, we note that SQP exhibits much better performance than VPQP w.r.t. queries that have more complex structures, i.e. the queries $Q4$-$Q15$. SQP is slightly worse w.r.t. simple queries, i.e. the single triple patterns $Q1$-$Q3$. This suggests that with more complex queries, the overhead incurred by the additional structure-level processing can be outweighed by the accumulated gain.

Breaking down the results, we found that the gain in loading and join is small for single-atom queries, but substantial for more complex structured queries. In these cases, it outweighs the additional cost of index matching.

Scalability We measured the average query performance for LUBM with varying size (i.e. generated for 1, 5, 10, 20 and 50 universities). We found that the performance of our improves with the size of the data. In particular, the gain for load and join increases in larger proportion than the overhead incurred for index match. This is because match performance is determined by the size of the index graph. This depends on the structure but not on the size of the data graph. Thus, the match time does not necessarily increase when the data graph becomes larger. The positive effect of data filtering (IO reduction) and query pruning (load and join) however, correlates with the data size.

RELATED WORK

We already compare our approach against triple stores along different dimensions and now, discuss related work on XML and semi-structured data processing.

Structure Index Structural information has been exploited for indexing semi-structured and XML data. Unlike *Dataguide* [5] and extensions of this concept for dealing with XML and semi-structured data [4, 7], (1) PIG can be constructed from general graph-structured data, (2) its concept of structural similarity is more fine grained, as it rests on characterizing the neighborhood via trees instead of paths, and (3) it can be parameterized such that only certain edge labels are considered.

Query Processing Our structure-aware processing is similar to the work on using structure indexes for (a) *evaluating path queries*, and for (b) *finding occurrences of twig patterns* (tree-structure) in a XML database based on multi-predicate merge join [11] such as *TwigJoin* [3]. These techniques rely on tree structures, and in particular, assume that relationships among query elements are of the

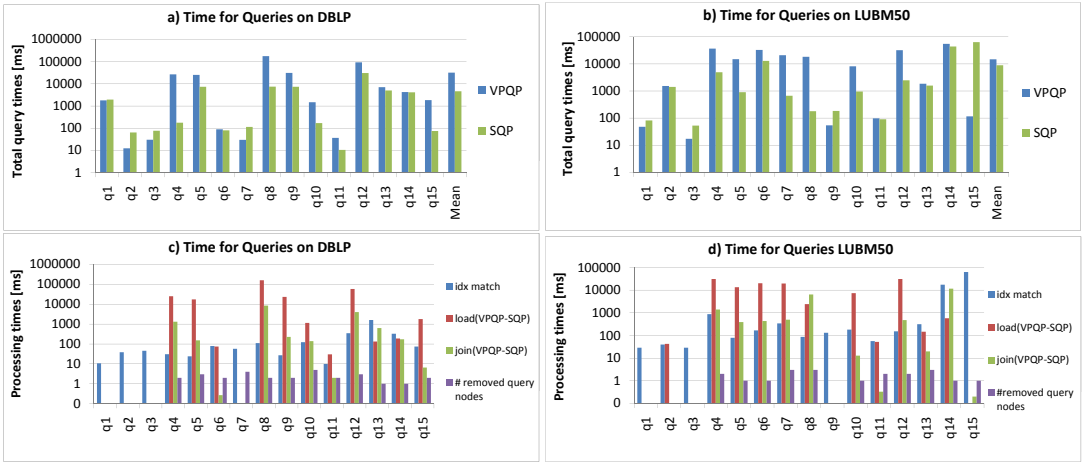

Figure 5: Total times for (a) DBLP and (b) LUBM50; Times for index matching, loading and joining for (c) DBLP and (d) LUBM50.

types parent-child or ancestor-descendant whereas in our setting, both query and data are generally graph structured. We employ rather a combination of (a) and (b): similar to (a), we match the query against the index graph. Similar to (b), intermediate results are computed and then joined to obtain the final answers. While intermediate results in (b) are simple root-to-leaf paths of the twig pattern, we retrieve and operate only on data elements that satisfy the structural constraints of the entire graph-structured query.

CONCLUSIONS AND FUTURE WORK

We have proposed techniques for RDF data partitioning and query processing that can exploit the underlying structure to improve the management of RDF data, based on a novel structure index call PIG. In an principled manner, we showed that this approach is faster than the state-of-the-art, especially for complex structured queries.

As future work, we will elaborate on how existing work on RDF query optimization can be used for the proposed structure-based query processing technique. Further, strategies proposed for optimizing updates of XML structure indexes will be studied and adopted.

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] Anonymous. Efficient RDF Data Management Using Structure Indexes for General Graph Structured Data. Technical report, 2010. `http://rs385.rapidshare.com/files/337240978/paper.pdf`.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[4] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *ICDT*, pages 336–350. Springer Verlag, 1997.

[5] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.

[6] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB*, pages 71–80, 2005.

[7] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *ICDE*, pages 129–140, 2002.

[8] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[9] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[10] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[11] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Conference*, pages 425–436, 2001.