Managing Structured and Semi-structured RDF Data Using Structure Indexes

Thanh Tran, Günter Ladwig, Sebastian Rudolph
Institute AIFB, Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
firstname.lastname@kit.edu

Abstract—We propose the use of a structure index for RDF. It can be used for querying RDF data for which the schema is incomplete or not available. More importantly, we leverage it for a structure-oriented approach to RDF data partitioning and query processing. Based on information captured by the structure index, similarly structured data elements are physically grouped and stored contiguously on disk. At querying time, the index is used for "structure-level" processing to identify the groups of data that match the query structure. Structure-level processing is then combined with standard "data-level" operations that involve retrieval and join procedures executed against the data. In the experiment, our solution provides several times faster performance than a state-of-the-art technique for data partitioning and query processing, and compares favorably with full-fledged RDF stores.

Index Terms—RDF, structure index, RDF data management, data partitioning, query processing.

1 Introduction

In recent years, the amount of structured data available on the Web has been increasing rapidly, especially *RDF data*. The Linking Open Data project[1] alone maintains tens of billions of RDF triples in more than 100 interlinked data sources. Besides strong (Semantic Web) community support, this proliferation of RDF data can also be attributed to the generality of the underlying graph-structured model, i.e., many types of data can be expressed in this format including relational and XML data. This data representation, although flexible, has the potential for serious *scalability* issues [1]. Another problem is that schema information is often unavailable or incomplete, and evolves rapidly for the kind of RDF data published on the Web. Thus, Web applications built to exploit RDF data cannot rely on a fixed and complete schema but in general, must assume the data to be semi-structured.

1.1 Problem

Towards the scalable management of semi-structured RDF data at Web-scale, we address the issues of (1) pseudo-schema construction, (2) RDF data partitioning and (3) conjunctive query processing.

Omitting special features such as blank nodes, a (semi-structured) RDF data source can be conceived as a graph:

Definition 1. *A* data graph *G is a tuple (V, L, E) where*

- *V is a finite set of* vertices. *Thereby, V is conceived as the disjoint union $V_E \uplus V_V$ of* entities *V_E and data* values *V_V.*
- *L is a finite set of* edge labels *denoting* properties, *subdivided via $L = L_R \uplus L_A$ into L_R representing* relations *between entities, and L_A representing entity* attributes.[2]
- *E is a finite set of* edges *of the form $e(v_1, v_2)$ with $v_1, v_2 \in V$ and $e \in L$. We require $e \in L_R$ if and only if $v_1, v_2 \in V_E$ and $e \in L_A$ if and only if $v_1 \in V_E$ and $v_2 \in V_V$. An edge $e(v_1, v_2)$ is also called a* triple *where e is referred to as the* property, *v_1 is the* subject *and v_2 the* object *of the triple.*

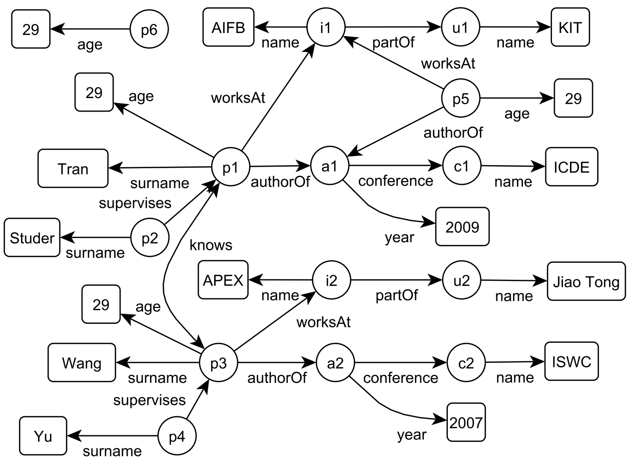


Fig. 1: A data graph.

Information needs commonly addressed by RDF-based applications can be expressed as a particular type of conjunctive queries:

Definition 2. *Let X be a countably infinite set of variables. A* conjunctive query *is an expression of the form $(x_1, \ldots, x_k).\exists x_{k+1}, \ldots x_m.P_1 \wedge \ldots \wedge P_r$, where*

1. http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

2. Note that the separation between L_R and L_A is not inherent to the RDF model but is used here to explicitly distinguish properties that associate entities with their attribute values and those that connect entities.

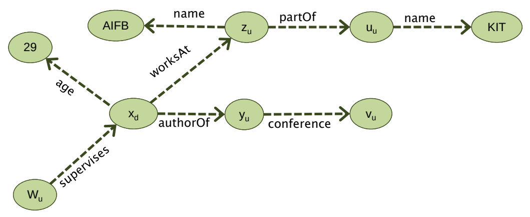


Fig. 2: A query graph for our example, where u stands for undistinguished and d stands for distinguished variable.

$x_1, \ldots, x_k \in X$ *are called* distinguished variables, $x_{k+1}, \ldots, x_m \in X$ *are* undistinguished variables *and* $P_1, \ldots, P_r$ *are* query atoms. *These atoms are of the form* $p(v_1, v_2)$*, where* $p \in L$ *are edge labels and* $v_1, v_2 \in X \cup V_E \cup V_V$ *are variables or, otherwise, are called* constants.

Fig. 1+2 show examples for the data and query we are concerned with. Fig. 2 shows that since variables can be connected in arbitrary ways, a conjunctive query q is graph-structured. In fact, this type of queries corresponds to the notion of Basic Graph Pattern (BGP) in SPARQL (more precisely, BGP queries without unbound predicates because e is always explicitly given): every query represents a *graph pattern* $q = (V^q, L^q, E^q)$ consisting of a set of *triple patterns* $e(v_1, v_2) \in E^q$, $e \in L^q$, and $v_1, v_2 \in V^q$, where V^q can be subdivided by $V^q = V^q_{var_d} \uplus V^q_{var_u} \uplus V^q_{con}$ representing the sets of distinguished variables, undistinguished variables, and constants.

A solution to a graph pattern q on a graph G is a mapping μ from the variables in q to vertices in G such that the substitution of variables would yield a subgraph of G. The *substitutions of distinguished variables* constitute the answers. In fact, μ' can be interpreted as a homomorphism (i.e., a structure preserving mapping) from the query graph to the data graph.[3]

1.2 State of the Art

This task of matching a query graph pattern against the data graph is supported by various RDF stores, which retrieve data for every triple pattern and join it along the query edges. While the efficiency of retrieval depends on the physical data organization and indexing, the efficiency of join is largely determined by the join implementation and join order optimization strategies. We discuss these performance drivers that distinguish existing RDF stores:

Data Partitioning. Different schemes have been proposed to govern the ways data is physically organized and stored. A basic scheme is the *triple-based organization*, where one big three-column table is used to store all triples. To avoid the many self-joins on the giant table, *property-based partitioning* is suggested [2], where data is stored in several "property tables", each containing triples of one particular type of entities. *Vertical partitioning* (VP) has been proposed to decompose the data graph into n two-column tables, where n is number of properties [1]. As this scheme allows entries to be sorted, fast merge joins can be performed.

Indexing Scheme. With *multiple indexing*, several indexes are created for supporting different lookup patterns. The scheme with the widest coverage of access patterns is used in YARS [3], where six indexes are proposed to cover 16 possible access patterns of quads (triple patterns plus one additional context element). In [4], *sextuple indexing* has been suggested, which generalizes the strategy in [3] such that for different access patterns, retrieved data comes in a sorted fashion. In fact, this work extends VP with the idea of multiple indexing to support fast merge joins on different and more complex query patterns. Thus, this indexing technique goes beyond triple lookup operations to support fast joins. Along this line, entire join paths have been materialized and indexed using suffix arrays [5]. A different path index based on judiciously chosen "center nodes" coined GRIN has been proposed in [6].

Query Processing & Optimization. Executing joins during query processing can be greatly accelerated when the retrieved triples are already sorted. Through VP, retrieved data comes in sorted fashion, enabling fast *merge joins* [1]. This join implementation has near linear complexity, resulting in best performance. Sextuple indexing takes this further to allow this join processing to be applied on many more query patterns, e.g. when the query contains unbound predicates such that p is a variable [4]. As shown for RDF-3X, further efficiency gains can be achieved by finding an *optimal query plan*, and additional optimization techniques such as dictionary encoding and index compression [7]. For query plan optimization, different approaches for selectivity estimation have been proposed recently [8], [9]. Besides, much of recent work also focuses on efficient data compression and storage [10], [11].

There are no single systems but rather, the state-of-the-art in RDF data management is constituted by a combination of different concepts. In particular, VP [1] is a viable candidate for physical data organization (especially for those queries with bound predicates studied here), multiple indexes [3] enable fast lookup, and optimized query plans [7] result in fast performance for complex join processing.

1.3 Contributions

In this work, we focus on the aspects of *data partitioning* and *query processing*. For these tasks, we propose a *structure-oriented approach* that exploits the structure patterns exhibited by the underlying data captured using a *structure index*. The contributions of our approach can be summarized as follows:

Height- and Label-Parameterized Structure Index for RDF. For capturing the structure of the underlying data, we propose to use the structure index, a concept that has been successfully applied in the area of XML- and semi-structured data management. It is basically a graph, where vertices represent groups of data elements that are similar in structure. For constructing this index, we consider structure patterns that exhibit certain edge labels L containing paths

3. As usual, a *homomorphism* from $G = (V, L, E)$ to $G' = (V', L', E')$ is a mapping $h : V \mapsto V'$ such that for every G-edge $e(v_1, v_2) \in E$ we have an according G'-edge $e(h(v_1), h(v_2)) \in E'$.

of some maximum length n. By varying these parameters, indexes with different properties and sizes can be obtained.

Structure-oriented Data Partitioning. A structure index can be used as a pseudo-schema for querying and browsing semi-structured RDF data on the Web. Further, we propose to leverage it for RDF data partitioning. To obtain a contiguous storage of data elements that are structurally similar, vertices of the structure index are mapped to tables. This is different to VP, where properties of a given schema are mapped to tables. Compared to VP, this scheme called structure-oriented data partitioning (SP) results in more fine-granular groupings of elements. Instead of triples with the same property label, triples with subjects that share the same structure are physically grouped. Such fine-granular groups that match a given query contain more candidate answers because while elements in a VP table have only the property label in common with the query, elements in a SP group match the entire query structure.

Integrated Data- and Structure-Level Processing. Standard query processing [1] relies on what we call data-level processing. It consists of operations that are executed against the data only. We suggest to use the structure index for structure-level query processing. A *basic strategy* is to match the query against the structure index first to identify groups of data that satisfy the query structure. Then, via standard data-level processing, data in these relevant groups are retrieved and joined. However, this needs to be performed only for some parts of the query, which additional to the structure constraints, also contain constants and distinguished variables representing more specific constraints that can only be validated using the actual data. Instead of performing structure- and data-level operations successively and independent from each other like in this basic strategy, we further propose an *integrated strategy* that aims at an optimal combination of these two types of operations.

We compared our solution with the state-of-the-art technique for data partitioning and join processing [1]. Through an evaluation on commonly used datasets, we show w.r.t. a given query workload that the basic structure-oriented strategy is several (up to 7-8) times faster. The integrated strategy plus further optimizations result in additional improvements. We also extend our solution with some optimizations that are employed by full-fledged RDF engines. Results suggest that it also compares favorably with state-of-the-art engines (RDF-3X and Sesame).

1.4 Outline

Section 2 introduces the basic structure-oriented approach, describing the notion of structure index, and the idea behind structure-oriented partitioning and query processing. The optimization of this approach comprising of index parameterization and the integrated evaluation strategy is elaborated in Section 3. Algorithmic complexity and optimal strategies for parameterization derived from it them are presented in Section 4. Experiments along with performance results are discussed in Section 5 before we review related work in Section 6 and conclude in Section 7.

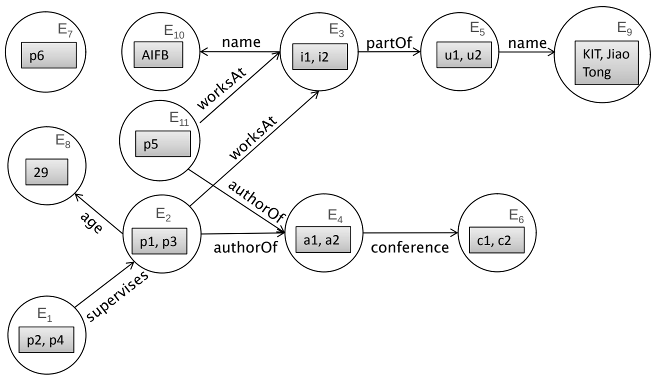


Fig. 3: A snippet of the structure index graph.

2 Structure-oriented Approach

We propose a structure-oriented approach to RDF data management where data partitioning and query processing make use of structure patterns captured by a structure index.

2.1 Structure Index for RDF

A structure index for graph structured data such as RDF is a special graph derived from the data. Vertices of such an *index graph* essentially stand for groups of data graph elements, which are similar in structure. Thereby, the structure of an element is determined by its incoming and outgoing paths. We draw from the well-known notion of *bisimulation* originating from state-based dynamic systems, and consider two graph vertices v_1 and v_2 as *bisimilar*, if they cannot be distinguished by looking at their paths. Later, we discuss the use of a fixed neighborhood to consider only certain paths with a maximum length. We first present the basic approach which is based on *complete bisimilarity*, where the neighborhood to be considered is simply the entire graph.

Definition 3. *Given a data graph $G = (V, L, E)$, a* backward-and-forward bisimulation *on G is a binary relation $R \subseteq V \times V$ s.t. for $v, w \in V$:*
(there is a backward bisimulation *on incoming edges s.t.)*

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

(and there is a forward bisimulation *on outgoing edges s.t.)*

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

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

Based on this notion of bisimilarity, we compute *extensions* (discussed later in Section 3), which contain only of bisimilar elements. These extensions form a partition of the vertices V, 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 4. *Let $G = (V, L, E)$ be a data graph and $\sim$ a bisimulation (or an adapted notion of bisimulation introduced later on) on V. Vertices of the associated* index graph *$G^{\sim} = (V^{\sim}, L, E^{\sim})$ 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 certain label l is established between two equivalence classes $[v]^{\sim}$ and $[w]^{\sim}$ if there are two vertices $v \in [v]^{\sim}$ and $w \in [w]^{\sim}$ such that there is an edge $l(v, w)$ in the data graph, i.e., $E^{\sim} = \{l([v]^{\sim}, [w]^{\sim}) \mid v \in [v]^{\sim}, w \in [w]^{\sim}, l(v, w) \in E\}$.*

Example 1. *An index graph for our example data is shown in Fig. 3. For the sake of presentation, this example considers all structures formed by relation edges (all edges with $l \in L_R$) but omits some data value nodes such as $ISWC$ and the attribute edge $surname$. Note that for instance, $p1$ and $p3$ are grouped into extension E2 because they share the incoming paths $supervise$ and $knows$ and the outgoing paths $knows$, $\langle worksAt, partOf\rangle$ and $\langle authorOf, conference\rangle$.*

2.2 Structure-oriented Partitioning & Indexing

The idea of structure-oriented partitioning (SP) is to apply the grouping of elements captured by the structure index to the physical organization of data. We create a physical group for every vertex of the index graph, i.e., for every extension. A group $g_{[v_i]^{\sim}}$ contains all triples that refer to elements in $[v_i]^{\sim}$, i.e., all $l(v_1, v_2)$ where $v_1 \in [v_i]^{\sim}$. Recall that extensions represent partitions of the data graph. Thus, grouping triples based on extensions guarantees an exhaustive and redundancy-free decomposition of the data.

Example 2. *The physical group of triples corresponding to E_2 is shown in Tab. 1.*

property	subject	object
authorOf	p1	a1
authorOf	p3	a2
worksAt	p1	i1
worksAt	p3	i2
knows	p1	p3
knows	p3	p1

TABLE 1: Triples for extension E2.

Compared to VP where triples with the same property are grouped together, SP applies to triples that are similar in structure. Using VP tables, triples retrieved from disk match the property of a single triple pattern. However, whether such a triple is also relevant for the entire query (i.e., contributes to the final results) depends on its structure. Since SP tables contain only triples that are similar in structure, they, when identified to be relevant for a query, are likely to contain relatively more relevant triples. In fact, triples of a SP table retrieved for a given query satisfy not only the property of a triple pattern of that query but also the entire query structure. Thus with SP, we can focus on data contributing to the final results. It can reduce the amount of irrelevant data that might have to be retrieved from disk when using VP, and thus, can reduce I/O costs.

For fast lookup on SP tables, we employ multiple indexes to support different access patterns. We use the standard PSO and POS indexes (e.g. as employed for indexing VP tables), which given the property and subject or given the property and object, return the object or subject, respectively. These are the indexes that are employed for standard data-level query processing, i.e. the kind of query processing based on the PSO and POS indexes that has been proposed for VP. Besides, we use two additional indexes called PESO and PEOS to support structure-level query processing. These indexes are similar to the other two, with the only difference that they also require extension information, e.g. PESO returns the object, given the property, extension and subject.

Because the structure index is also a kind of data graph, VP (in combination with the PSO and POS indexes) can be used for managing those that are large in size. In our approach, we propose to parameterize the index construction process to consider only certain structure information so that its size can be controlled to fit in main memory.

2.3 Structure-aware Query Processing

We propose not to process the query directly against the data graph but to use the index graph first. Through this *structure-level processing*, we aim to locate candidate groups of data that satisfy the query structure. Only then, "relevant" data is retrieved from these candidate groups and processed using standard query processing techniques. Since these operations performed subsequently apply on the data, they are referred to as *data-level processing*.

In the basic query evaluation strategy, data-level processing applies after structure-level processing has been completely finished. The procedure for that can be decomposed into three main steps: (1) *index graph matching*, (2) *query pruning* and (3) *final answer computation*.

(1) At first, the query graph q is matched against the index graph $G^{\sim}$ (kept in memory) to find matching extensions $[v_i]^{\sim}$. This graph matching is performed using the same operations that are also used for standard query processing, i.e. retrieve triples (edges of $G^{\sim}$) for every triple pattern of q and then join them along edges of q. This step results in extensions $[v_i]^{\sim}$, which contain candidate results to q.

(2) In particular, we know after the first step that data in $[v_i]^{\sim}$ satisfies the entire query structure. Query parts containing undistinguished variables only capture structure constraints. Since we already know that data in $[v_i]^{\sim}$ satisfies these constraints, these parts are removed in the second pruning step to obtained q'.

(3) Final results are computed from $[v_i]^{\sim}$ in the third step. Using standard query processing techniques, triples are retrieved from tables corresponding to $[v_i]^{\sim}$, and joined along the edges of the pruned query q'.

The soundness and completeness for this procedure is based on the following property:

Proposition 1. *Let G be a data graph, $G^{\sim}$ its index graph and q be a query graph s.t. there is a homomorphism h from*

q to G. Then $h^\sim$ with $h^\sim(v) = [h(v)]^\sim$ is a homomorphism from q to $G^\sim$.

Intuitively speaking, there is a match of q on G only when there is also a match of q on $G^\sim$. Further, the resulting index graph matches $[h(v)]^\sim$ will contain *all* answers $h(v)$.

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. □

For computing the answers $h(v)$, another property of the structure index can be exploited. It is based on the observation that for certain tree-shaped parts of the query, no further processing at the data-level is needed because they capture structure constraints that have already been verified during the previous index matching step. We firstly inductively define this notion of tree-shaped query part:

Definition 5. *Every edgeless single-vertex rooted query graph (q, r) with $q = (\{r\}, L, \emptyset)$ is* tree-shaped. *Let (q_1, r_1) and (q_2, r_2) with $q_1 = (V_1, L, E_1)$ and $q_2 = (V_2, L, E_2)$ be two tree-shaped query graphs with disjoint vertex sets V_1 and V_2, $v \in V_1$, and $e \in \{l(r_2, v) \mid l \in L\} \cup \{l(v, r_2) \mid l \in L\}$, then the rooted graph (q_3, r_1) with $q_3 = (V_1 \cup V_2, L, E_1 \cup E_2 \cup \{e\})$ is tree-shaped. Further, such a query graph (q, r) is* undistinguished tree-shaped *when except for the root node r, all other nodes of q represent undistinguished variables.*

Given an undistinguished tree-shaped query (q, r) of this kind, the following proposition specifies that data contained in the index graph extension $[h(r)]^\sim$ matching the root node r of q contains *all* and *only* the final answers such that no further processing is required for q.

Proposition 2. *Let G be a data graph, $G^\sim$ its index graph, (q, r) be an undistinguished tree-shaped query graph s.t. h is a homomorphism from q to G and $h^\sim$ is a homomorphism from q to $G^\sim$ with $h^\sim(r) = [h(r)]^\sim$, then $h(r) = v$ for every data graph node $v \in h^\sim(r)$.*

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

Example 3. *As illustrated in Fig. 4, the query in Fig. 2 produces one match on the index graph in Fig. 3, i.e., $h_1^\sim = \{h^\sim(z) = E3, h^\sim(u) = E5, h^\sim(x) = E2, h^\sim(y) = E4, h^\sim(v) = E6, h^\sim(w) = E1\}$. We know that data elements in extensions of this match satisfy the query structure, e.g. elements in $E2$ are authors of y, supervised by w, work at some place z etc. Tree-*

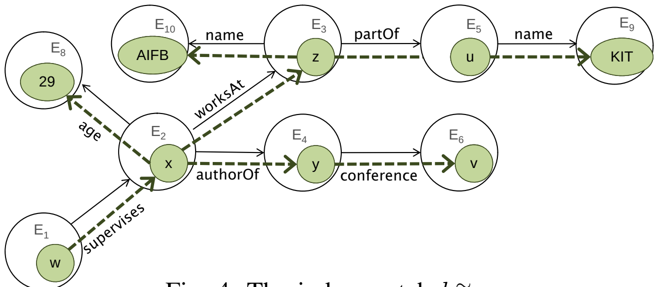


Fig. 4: The index match $h_1^\sim$.

shaped parts that can be pruned are $supervises(w, x)$ and $\langle authorOf(x, y), conference(y, v)\rangle$. For instance, since elements in $E2$ are already known to be supervised by some w, there is no need to process $supervises(w, x)$.

Data-level processing is needed for the remaining query parts. Fig. 5 shows the pruned query and the data involved in this step. Now, we have to find out which elements in E2 are of age 29, which elements in E3 have the name $AIFB$, which elements in E5 have the name KIT, and whether the elements matching these constants are connected over the relations specified in the query. For this, we need to retrieve and join the triples for $age(x, 29)$, $worksAt(x, z)$, $name(z, AIFB)$, $partOf(z, u)$, $name(u, KIT)$. Note that in the extreme cases where no index graph matches can be found, or when the entire query can be pruned, we can skip the entire data-level processing step.

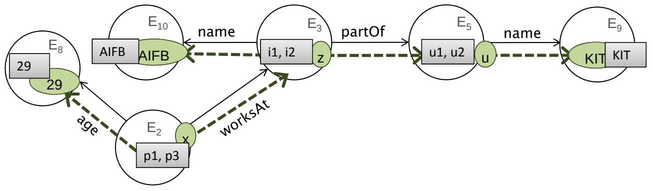


Fig. 5: Data graph match for $q_{pruned}, h_1^\sim$.

3 Optimized Query Processing

The basic approach captures the main idea of using the structure index. However, it has certain drawbacks, which are addressed by the optimization discussed here.

3.1 Parameterizing the Structure Index

The structure index introduced previously is based on complete bisimilarity. For constructing such an index, all paths in the data graph have to be considered. It results in very fine-grained extensions, where constituent elements are similar w.r.t. possibly very complex structures. Thus, the number of extensions might be high,[4] resulting in an index graph that is too large in size, which in turn, has a negative effect on the complexity of structure-level processing. To address this problem, we propose a parameterizable structure index which applies a more relaxed notion of neighborhood-equivalence than full bisimulation, hence results in a coarser partition.

4. Note however, that there are at most as many extensions as data graph vertices, i.e., the index is never larger than the data.

Height- and Label-Parameterized Structure Index. Instead of all property labels, we propose to consider only certain labels (L_1 for backward and L_2 for forward bisimulation). In addition, we can restrict the size of the neighborhood taken into account by an upper limit n for the number of consecutive hops from the considered node. For this, we introduce the L_1-L_2 *label-parameterized bisimulation of bounded heigth* n.

In particular, two elements w and v are *n-bisimilar* ($w \overset{n}{\sim} v$), if G contains for every tree structure (t, w) with root w of height n an equivalent structure (t, v) with root v and vice versa; and furthermore, they are L_1*-backward-*L_2*-forward bisimilar* if w and v are coincide w.r.t. being roots of trees with the property that all edges with leaf-to-root orientation are from L_1 and all edges with root-to-leaf orientation are from L_2.

Clearly, Proposition 1 and 2 also hold in the case of the height- and label-parameterized structure index, given some restrictions on the query. Namely, the notion of undistinguished tree-shaped query must be extended:

Definition 6. *The notion of a* L_1-backward-L_2-forward tree-shaped *(short:* L_1L_2ts*) query is inductively defined as follows: Every edgeless single-vertex rooted query graph* (q, r) *with* $q = (\{r\}, L, \emptyset)$ *is* L_1L_2ts*. Let* (q_1, r_1) *and* (q_2, r_2) *with* $q_1 = (V_1, L_1 \cup L_2, E_1)$ *and* $q_2 = (V_2, L_1 \cup L_2, E_2)$ *be two* L_1L_2ts *query graphs,* $v \in V_1$*, and* $e \in \{l(r_2, v) \mid l \in L_1\} \cup \{l(v, r_2) \mid l \in L_2\}$*, then the rooted graph* (q_3, r_1) *with* $q_3 = (V_1 \cup V_2, L_1 \cup L_2, E_1 \cup E_2 \cup \{e\})$ *is* L_1L_2ts*. Further, such a query graph* (q, r) *is* $L1$-backward-$L2$-forward n-tree-shaped *when the height of* q *is* n*.*

Only certain query parts that are $L1$-backward-$L2$-forward n-tree-shaped can be pruned. This parameterization reduces the size of the index: for a given data graph, the number of vertices and edges of the index graph increases with n and the number of labels contained in L_1 and L_2.

Constructing the Parameterized Index. The concept of structure index is not new. Various strategies for index construction, management, and updates exist [12], [13], [14], [15].

In general, the construction of a structure index is based on the algorithm for forward bisimulation presented in [16] which essentially, is an extension of Paige & Tarjan's algorithm [17] for determining the coarsest stable refinement of a partitioning. This algorithm starts with a partition consisting of one single extension that contains all nodes from the data graph. This extension is successively split into smaller extension until the partition is stable, i.e., the graph formed by the partition is a complete bisimulation.

In order to support the parameterization proposed previously, we make the following modifications to this algorithm:

(1) For creating a bisimulation of depth n only, we also compute partitions of the data graph by splitting blocks so that each partition is stable with respect to its predecessor. However, this refinement is performed n times only.

(2) In order to perform both backward and forward bisimulation according to 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 $G_{L_1L_2}$ with the same set of nodes as G, containing all of G's L_1-edges and flipped around copies of G's L_2-edges, formally: $G_{L_1L_2} = (V, L_1 \cup \{l^- \mid l \in L_2\}, E_{L_1L_2}\})$ where $E_{L_1L_2} = \{l(x, y) \mid l(x, y) \in E, l \in L_1\} \cup \{l^-(y, x) \mid l(x, y) \in E, l \in L_2\}$. Therefore, applying the algorithm from [16] to the specified label sets L_1 and L_2 yields the desired result, and time complexity $(O(|L_1 \cup L_2| \cdot |E| \cdot \log |V|))$.

After having determined the bisimulation, the resulting extensions from the stable partition $\mathcal{P}^{\sim}$ are used to form index graph vertices, and edges between them are constructed according to Definition 4.

It is important to note that updates pose an interesting problem in the presence of a structure index. There is existing work [18] which shows that it is feasible for XML- and semi-structured data. However, adopting this work to support our structure index concept goes beyond the scope of this paper as the focus is set on how to exploit such an index for data partitioning and query processing.

3.2 Integrated Query Processing

Recall that the main idea behind our approach is to apply structure-level processing to obtain a smaller query and smaller sets of candidates. This strategy makes sense when the effect of this "pruning" outweighs the cost of structured-level processing. Here, we propose an optimization based on the observation that the naive strategy discussed before might fail to succeed in the following cases:

(1) *Highly selective (non-prunable) triple patterns:* When queries contain only few triple patterns and these patterns are highly selective, e.g. have many constants, standard query processing (i.e., data-level processing only) can be performed efficiently. The amount of data retrieved in these cases is relatively small and thus, can be joined fast. Here, there is simply no much potential gain that can be achieved through pruning. In particular, patterns containing constants are non-prunable.

(2) *Queries with only a few prunable parts:* This is the case when most query parts contain constants and distinguished variables. Structure-level processing only helps to locate candidate groups and data-level processing is still needed for most parts of the query.

General Idea. Instead of carrying out structure-level and data-level operations subsequently, we perform these operations in an integrated fashion. It is based on the following intuitions:

(1) Data-level processing shall be executed when processing selective query patterns, i.e., patterns with constants. In the extreme case when all triple patterns contain constants, the query is processed using the standard approach based on *data-level operations only*.

(2) Structure-level processing shall be executed for prunable parts. In the case when the entire query can be pruned, it is processed using *structure-level operations only*.

(3) Structure-level processing shall leverage results of data-level processing to reduce the number of candidates,

and vice versa, i.e., the query is processed via a *mixture of structure- and data-level operations*.

Detailed Algorithm. The procedure for integrated query processing is shown in Alg. 1 and illustrated in Fig. 6. The query is decomposed into two parts, one prunable part that is processed via structure-level processing and one non-prunable part that is processed via data-level processing. Results of both processes are combined and propagated to reduce the number of candidates at both levels.

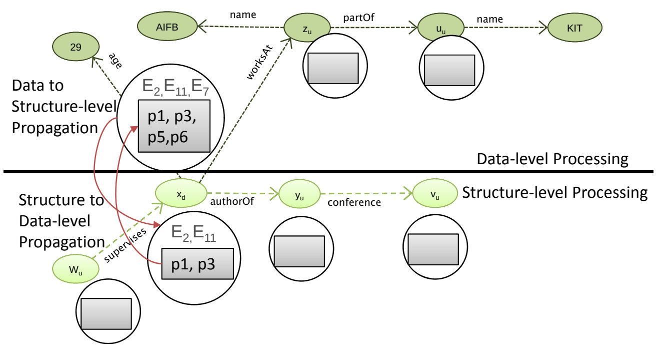


Fig. 6: Integrated processing.

Algorithm 1: Integrated Query Processing

```
Input: Query $q(V^q_{var_d} \uplus V^q_{var_u} \uplus V^q_{con}, L^q, E^q)$; data
       $G(V, L, E)$; the parameters $n, L_1, L_2$.
Data: Queue $E^q_{sorted}$ of triple patterns sorted
      according to selectivity; prunable query parts
      $Q'$.
Result: $R$ containing data matches of $q$ on $G$.
1  $Q' \leftarrow$ prunableQueryParts($q, n, L_1, L_2$).
2  $e(v^q_1, v^q_2) \leftarrow$ selectNextBest($E^q_{sorted}$).
3  $R \leftarrow \{e(h(v^q_1), h(v^q_2)) \in E\}$.
4  while $|E^q_{sorted}| \neq 0$ do
5      $e(v^q_1, v^q_2) \leftarrow$ selectNextBest($E^q_{sorted}$).
6      if $e(v^q_1, v^q_2) \in q' \in Q'$ then
7          $R_r \leftarrow$ structureJoin($q', R$).
8          $R \leftarrow R \bowtie R_r$.
9      end
10     else
11         $E_{e(v^q_1, v^q_2)} \leftarrow \{e(h(v^q_1), h(v^q_2)) \in E\}$.
12         $R \leftarrow R \bowtie E_{e(v^q_1, v^q_2)}$.
13     end
14 end
15 Return $R$.
```

The integrated strategy starts with (Part 1) identifying prunable parts $q' \in Q'$ (line 1). Then, the method *selectNextBest* is used to select and remove the first triple pattern from E^q_{sorted} and R is then initialized to contain the data matching that pattern, such that further joins can be performed. Then, triple patterns $e(v^q_1, v^q_2) \in E^q_{sorted}$ are worked off. This is performed in the order of their selectivity as proposed in [19]. Here, *selectNextBest* selects and removes a triple pattern from E^q_{sorted} that can be joined with at least one already processed pattern (that is, join processing follows a left-deep query plan). If $e(v^q_1, v^q_2)$ belongs to one of the prunable query parts, (Part 2) structure-level processing is performed (lines 5-6), (Part 3) data-level processing otherwise (line 9-10).

(*Part 1*) Prunable query parts are computed for the parameters n, L_1, L_2. For this, we firstly identify all tree-shaped parts of q containing undistinguished variables only. This is performed by traversing q. Starting from leave nodes representing undistinguished query variables, neighbors of these nodes are visited (in a breadth-first-search manner) and labeled. A search from a particular leave node stops when (1) the neighbor to be visited is not an undistinguished variable node or (2) is a root node of a tree structure of height n or (3) the neighbor edge is not labeled with elements in the sets L_1, L_2. In the end, all triple patterns containing only labeled nodes are grouped into connected components which constitute exactly the prunable tree-shaped query parts $q' \in Q'$.

(*Part 2*) During structure-level processing, prunable query parts q' are processed using structure-level join (line 5). The procedure for that is shown in Alg. 2. It matches q' against the index graph $G^{\sim}$ to produce a result set R_r containing data graph elements that match the root node $v^{q'}_r$ of q'. However, only extensions found to be relevant through previous data-level processing are taken into account, i.e., $R^{\sim}$ is instantiated with extensions in $R^{\sim}_r$ (line 2); and $R^{\sim}_r$ comprises only those extensions which contain data elements v matching the root node $v^{q'}_r$ (line 1). Note that these data matches have already been computed and stored in the intermediate results R. We leverage these data-level results to reduce the number of structure-level candidates. Now, starting from $v^{q'}_r$, all triple patterns $e(v^q_1, v^q_2)$ in q' are visited via breadth-first search (line 4). Edges of the index graph, which match $e(v^q_1, v^q_2)$ are retrieved (line 5), and combined with the intermediate results $R^{\sim}$ (line 6). Finally, data contained in extensions in $R^{\sim}$, which match the root node $v^{q'}_r$, are returned (line 8-9).

These results of the structure-level processing feed back into the main query processing procedure (Alg. 1). Data elements in the result table R_r are joined with element in R (line 6). As a result, only data elements that match the query part q' are maintained in R, i.e., structure-level results are propagated back to the data-level process.

(*Part 3*) During data-level processing (Alg. 1), triples matching the current triple pattern are retrieved from the data graph G (line 9) and combined with the intermediate results in R (line 10).

Example 4. *This example illustrates how the query in Fig. 2 is processed against the data graph in Fig. 1.*

During the search for prunable parts of q we obtain the patterns $Q' = \{supervises(w_u, x_d),\ authorOf(x_d, y_u),\ conference(y_u, v_u)\}$. As they are connected, they form one prunable part q'.

For processing the query, we start with $age(x_d, 29)$ (age is most selective). From the data graph we retrieve all matches for x_d, which are $p1$, $p3$, $p5$, and $p6$. When processing the next triple pattern $authorOf(x_d, y_u)$, we observe that this pattern is part of the prunable part q' which

Algorithm 2: Structure-level Join Processing

```
Input: Index graph $G^{\sim}(V^{\sim}, L, E^{\sim})$; prunable query
    part $q'$; $R$ containing data matches for $q$.
Data: $R_r^{\sim}$ containing candidate index matches for the
    root node $v_r^{q'}$ of $q'$; $R^{\sim}$ containing index graph
    matches for $q'$.
Output: $R_r$ containing data matches for $v_r^{q'}$.
1 $R_r^{\sim} \leftarrow \{[v]^{\sim} \mid h(v_r^{q'}) = v,\ h \in R\}$.
2 $R^{\sim} \leftarrow R_r^{\sim}$.
3 while nextBFSNeighbor($q'$) $\neq \emptyset$ do
4     $e(v_1^q, v_2^q) \leftarrow$ nextBFSNeighbor($q'$).
5     $E^{\sim}_{e(v_1^q, v_2^q)} \leftarrow \{e(h^{\sim}(v_1^q), h^{\sim}(v_2^q)) \in E^{\sim}\}$.
6     $R^{\sim} \leftarrow R^{\sim} \bowtie E^{\sim}_{e(v_1^q, v_2^q)}$.
7 end
8 $R_r \leftarrow \{v \mid h^{\sim}(v_r^{q'}) = [v]^{\sim},\ h^{\sim} \in R^{\sim}\}$.
9 Return $R_r$.
```

is to be processed on the index level. Hence, we determine $R_r^{\sim}$ by looking up the extensions the matches of x_d belong to: $E2$, $E11$, $E7$. We now retrieve the index level edges for the pattern $authorOf(x_d, y_u)$: $(E2, E4)$ and $(E11, E4)$. The subsequent join with $R_r^{\sim}$ does not further reduce this edge set in this case (because $E2, E11 \in R_r^{\sim}$). Next, we process the triple pattern $supervises(w_u, x_d)$. In the index graph, there is only one $supervises$ edge: $(E1, E2)$. The join with $R_r^{\sim}$ leaves us with only one index graph match: $h^{\sim}(w_u) = E1$, $h^{\sim}(x_d) = E2$, $h^{\sim}(y_u) = E4$. Processing $conference(y_u, v_u)$, the last triple pattern of q', yields $h^{\sim}(v_u) = E6$.

Hence, on the structural level, we found $E2$ as the only match for x_d. As a result, the data elements matching x_d must be contained in $E2$, i.e., $R_r = \{p1, p3\}$. Returning to the data level, we can thus rule out $p5$ and $p6$ as matches for x_d. Joining the two remaining matches with the $worksAt$ edge results in the (x_d, z_u) assignments $(p1, i1)$ and $(p3, i2)$. The second one of these is ruled out by the next triple pattern $name(z_u, AIFB)$. The two remaining joins on the data level do not change this result. As answer, $p1$ is returned as the only binding to x_d.

4 Algorithm Analysis

4.1 Soundness and Completeness

Based on the properties elaborated for the structure index, the following result can be established:

Proposition 3. *Given a data graph, its associated index graph and a query, the result obtained by Algorithm 1 is sound and complete, i.e., it returns exactly all data graph matches.*

Proof sketch: Soundness can be inferred from the soundness of data- and the structure-level operations: the procedure (including the join operations) we employ for data as well as index graph matching is the same as the standard and sound technique used for processing graph patterns (e.g. [1]). Completeness of the retrieved results is guaranteed by completeness of the join operations on the data-level for the non-prunable query part and by Proposition 1. Recall that this proposition ensures that any match of a query on the data graph (hence also any match of prunable query parts) is "witnessed" by an according match on the index graph. Therefore, constraining the search for prunable query part matches by according index matches will not result in the loss of any match. Subsequent join operations performed on the remaining non-prunable query part find all the possible data matches per index match. □

4.2 Complexity

Complexity of standard query processing [1] is $O(edgemax^{|E^q|})$, where $|E^q|$ denotes the number of triple patterns and $edgemax$ is $|\{(v_1, v_2) \mid l(v_1, v_2) \in E\}|$ with l being the property instantiated by the largest number of edges. This is because joins have to be performed $|E^q|$ times and every operation $A \bowtie E$ can be calculated in at most $|A| \cdot |E|$ time and space. This cost cannot be avoided in the general case but in practice, techniques for partitioning and indexing ([1], [4]) result in near-linear behavior. Compared to this, the complexity of our approach is as follows:

Proposition 4. *For a query with $|E^q|$ triple patterns, the time and space for query processing is bounded by $O(edgemaxidx^{|E^q|} + edgemaxdata^{|E^q_{pruned}|})$ where $edgemaxidx = \max_{l \in L} |\{l([v_1]^{\sim}, [v_2]^{\sim}) \in E^{\sim}\}|$ and $edgemaxdata = \max_{[v_1]^{\sim}, [v_2]^{\sim} \in V^{\sim}, l \in L} |\{l(v_1, v_2) \mid v_1 \in [v_1]^{\sim}, v_2 \in [v_2]^{\sim}\}|$.*

Proof sketch: The complexity of our algorithm is composed of the complexity for data-level as well as structure-level processing. The cost for computing index matches is $O(edgemaxidx^{|E^q|})$, where $edgemaxidx$ is bounded by the size of the index graph (presisely: the label l that is associated with the largest number of edges). Data-level joins have to be performed only along the pruned version $E^q_{pruned} \subseteq E^q$ of the query. So we obtain a complexity of $O(edgemaxdata^{|E^q_{pruned}|})$, where $edgemaxdata$ is bounded by the size of the largest extension. □

Compared to existing approaches ([1], [4], [7]), less data have to be retrieved from G (i.e., $edgemaxdata \leq edgemax$ because structure-oriented data groups are more fine-granular), and also, fewer joins are required (i.e., $|E^q_{pruned}| \leq |E^q|$). The overhead introduced to achieve this $O(edgemaxidx^{|E^q|})$.

4.3 Structure Index Parametrization

Note that the parametrization has an effect on both the overhead and the gain introduced by structure-oriented query processing: when more labels and (or) larger height are used, the index graph and thus $edgemaxidx$ becomes larger. On the other hand more labels and (or) larger height can be considered for query pruning, potentially resulting in larger prunable query parts (i.e., smaller $|E^q_{pruned}|$).

Also, the physical groups obtained from structure-oriented partitioning become more fine-grained, thereby reducing the size of $edgemaxdata$.

Thus, it is not straightforward to derive the optimal parameterization for the general case. However, the parameters can be used for fine tuning the approach w.r.t. the given workload. One simple strategy is to support only the labels and the maximum query height derived from the workload, i.e., L_1, L_2 contains all query predicates and the length of the longest query path is $\leq n$. More fine-granularly, the parametrization can be done for classes of queries, which are used frequently. In this case, structure-oriented technique can be seen as an optimization technique, which is applied only to some queries. Instead of assuming a given workload, another strategy is to focus on potentially frequent and problematic queries derived from the data. For this, existing techniques for frequent graph patterns can be adopted [20]. In particular, the strategy here would be to select those patterns as queries, which are frequent and produce a large number of results (and thus are potentially more problematic because they require more data to be processed.

Note that while the integrated evaluation strategy does not change the worst-case complexity, it aims at minimizing both $edgemaxidx$ and $edgemaxdata$ by prioritizing cheap operations to quickly filter out irrelevant candidates.

5 EVALUATION

We performed two experiments, one to focus the comparison on differences between our approach and an implementation of VP, the most related work, and the other also involves full-fledged RDF stores (Sesame, RDF-3X). All experiments were carried out on a machine with two Intel Xeon Dual Core 2.33 GHz processors and 2GB main memory were allocated to the Java virtual machines. All data and indexes are stored on a Samsung SpinPoint S250 200GB, SATA II. Components of all systems under consideration have been implemented in Java 5. All times presented represent the average of 10 runs. Between queries, we explicitly clear the operating system cache and internal caches of subprograms (e.g. Lucene).

5.1 Structure-oriented Approach vs. Baseline

Systems. To focus on the aspect of data partitioning and query processing, we compare our approach with *VP*. In particular, we implemented three systems. The first system called VPQP is the baseline, which is based on the execution of merge joins on VP tables [1]. This is compared against a simple version of our structure-oriented approach called SQP, which implements the basic strategy and uses the label-parameterized index $G^{\sim}_{L_1,L_2}$ (described below). Then, we compare SQP against another version called OSQP, which uses the same index and additionally, implements the integrated evaluation strategy.

To achieve comparability, all these approaches were implemented on top of the same infrastructure: For data indexing and storage, we have implemented all systems using the inverted index that comes with Lucene (following the design in [21]). For query processing, we use the same implementation for merge join, and query predicates are worked off in a random but same order for all systems (i.e., no query optimization).

Datasets and Indexes. We use *DBLP*, which captures bibliographic information. Further, we used the data generator proposed for the *LUBM* benchmark to create 4 datasets for 1, 5, 10, and 50 imaginary universities.

For these datasets, the number of edges are shown in Table 2. For comparison, Table 2 also contains the number of edges for three different versions of the structure index. The first one called $G^{\sim}_{\text{Full}}$ is calculated using complete bisimulation. The second one, called $G^{\sim}_{L_1,L_2}$, represents a label-parameterization that was derived from the the query workload, i.e., by searching for prunable query parts in the queries and parameterizing L_1, L_2 accordingly such that labels of all edges contained in these parts are included. The third one denoted $G^{\sim}_{\text{2-d}}$ is based on height-parameterization with $n = 2$. The results show that while the size of the full indexes $G^{\sim}_{\text{Full}}$ is almost as big as the size of the dataset (62%-87%), the size of $G^{\sim}_{L_1,L_2}$ is much smaller (4%-30%). The size of the heigth-parameterized indexes $G^{\sim}_{\text{2-d}}$ makes up only a small percentage (0.08%-2%).

	$G^{\sim}_{\text{2-d}}$	$G^{\sim}_{L_1,L_2}$	$G^{\sim}_{\text{Full}}$	Dataset
DBLP	1278	557,423	11,600,000	12,920,826
LUBM1	665	30,641	87,590	100,577
LUBM5	736	151,396	553,362	722,987
LUBM10	552	253,088	1,127,231	1,272,609
LUBM50	559	343,682	5,754,191	6,654,596

TABLE 2: Statistics for the data graphs and indexes.

5.2 Queries

We use queries of five different types, each representing a particular query shape. We create 3 queries for each of the types, resulting in 15 queries for each dataset and a total of 30 queries. The complete list of queries can be found in our technical report [22]. We will now discuss examples.

(1) *Single-triple-pattern queries* consist of exactly one single triple pattern. Query 1 on DBLP ($Q_{DBLP}1$) for instance, simply asks for all persons. (2) *Path queries* consist of several connected triple patterns that form a path. As an example, $Q_{LUBM}6$ retrieves all students and courses that are lectured by full professors. (3) *Star queries* are composed of more than two path queries. These parts share exactly one common node, i.e., the center node. For instance, $Q_{DBLP}12$ retrieves names of people, who are both editors and authors, and whose papers have been cited. (4) *Entity queries* are similar to star queries. They are formed by triple patterns that share exactly one common node, i.e., the entity node. $Q_{LUBM}9$ asks for entities who have a given email address, research interest and telephone. (5) *Graph-shaped queries* consist of nodes and edges that form a graph. As opposed to the star query, queries of this type might contain cycles and loops. $Q_{LUBM}15$ for instance, retrieve authors x and advisors a that match some specified constraints. Namely, both shall be member of the same organization, a telephone number is specified for a,

and x shall be taking the course run by $FullProfessor5$ and also, has authored $Publication7$.

Results. Total query processing for both approaches for the two datasets are presented in Fig. 7a+7b. For most cases, SQP is faster than VPQP. For DBLP, SQP is a factor of 7-8 faster on average. We noted that SQP exhibited much better performance than VPQP w.r.t. queries that have more complex structures, i.e., the queries $Q4$-$Q15$ representing path, entity, star and graph-shaped queries. 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 additional structure-level processing can be outweighed by the accumulated gain.

To better understand the reasons for this, we decomposed total processing time into the fragment required for structure-level processing, i.e., index matching (idx match), to retrieve data from disk (load) and to combine them (join). We compared the time SQP needed for matching with the difference for load and join between SQP and VPQP. This is shown in Fig. 7c+7d to illustrate when the additional cost of index matching is outweighed by the gain in loading and join performance. Fig. 7c+7d also illustrates the impact of query pruning measured in terms of the number of query nodes, which were discarded after structure-level processing. Along these factors, we will now discuss results for each specific query type.

Single-triple-patterns Q1-Q3: Here, SQP was slower than VPQP. This is because only a single index lookup or table scan is needed to answer these queries. In the cases of index lookup only, i.e., $Q_{\mathrm{DBLP}}1$-3, $Q_{\mathrm{LUBM}}1$ and $Q_{\mathrm{LUBM}}3$, our approach did not provide any advantages but required additional effort for structure-level processing. For $Q2$ on LUBM, which required a scan to retrieve all triples matching a predicate, one can see that SQP improved loading time. Performance of SQP equals VPQP in this case, as the gain fully compensated the cost for structure-level processing.

Path Queries Q4-Q6: For processing these queries, triples retrieved for the patterns have to be joined along the path. Here, we observed that SQP largely reduced time for load and join w.r.t. all queries of this type. This is especially true for queries containing patterns with low selectivity. For instance for $Q_{\mathrm{LUBM}}6$ the number of triples retrieved for the SP table $\langle y \in B_i \ \mathit{teacherOf} \ z\rangle$ was many times smaller than the VP table $\langle y \ \mathit{teacherOf} \ z\rangle$, resulting in large relative advantages for load and join performance. Another advantage is that path queries lend themselves to be pruned: on average 1.6 query nodes were removed.

Entity Queries Q7-Q9: For these queries, triples need to be retrieved and joined on the node representing the entity. Also here, SQP improved I/O and join performance. SQP outperformed VPQP w.r.t. all queries except for $Q_{\mathrm{LUBM}}9$. We observe that best performance was achieved with queries containing few constants ($Q_{\mathrm{LUBM}}8$, $Q_{\mathrm{DBLP}}8$, $Q_{\mathrm{DBLP}}9$). Query patterns without constants could be removed in most cases, thus helped to avoid unnecessary load and join operations. For instance $Q_{\mathrm{LUBM}}8$ contains only one constant, while $Q_{\mathrm{LUBM}}9$ has 3 constants. Both contain the same number of triple patterns but $Q_{\mathrm{LUBM}}8$ benefited much more from reduction in I/O and join. In fact, the performance gain for $Q_{\mathrm{LUBM}}9$ did not outweigh the cost for structure-level processing. SQP was slower than VPQP in this case, as many constants could be effectively used for triple lookups. There is thus not much data that could be “filtered” through structure-level processing. One could say that the number of constants and the selectivity of triple patterns have an adverse effect on the gain that can be achieved with SQP.

Star Queries Q10-Q12: These queries behaved similar to entity queries. However, pruning is more delicate as distinguished variables might be not at the center but anywhere. However, since star queries were larger than entity queries on average, larger portion of the queries lent themselves to be pruned. Also for this type of queries, results show that performance of SQP was superior to VPQP.

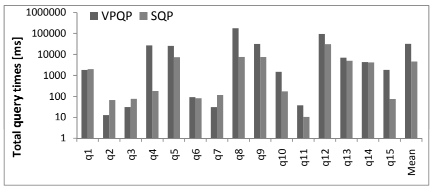
Graph-shaped Queries Q13-Q15: We expected that the relative performance of SQP is best here, since these queries have most complex structures. However, while the performance of SQP was still better than VPQP for all graph-shaped queries, except for $Q_{\mathrm{LUBM}}15$, the relative improvement of SQP over VPQP is not as high as the improvement achieved for entity and star queries, i.e., 3-4 times compared to 9-10 times improvement. Results show that the main reason for this is because structure-level processing cost largely increased with complexity.

Scalability. We measured the average performance for LUBM with varying size, i.e., different number of universities (1, 5, 10, 20, 50). Fig. 7e shows the differences (VPQP-SQP) in total processing time, load and join and also the times SQP needed for index matching. Clearly, the performance of SQP improved with the size of the data. The improvement of SQP over VPQP (indicated by VPQP-SQP) increased more than linearly with the size, e.g., LUBM5 (LUBM60) is 7 (66) times bigger in size than LUBM1 while the improvement for LUBM5 (LUBM50) is more than 8 (92) times the improvement achieved for LUBM1.

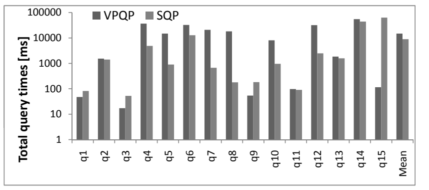
In particular, the differences in performance for load and join increased in larger proportion (the gain) than the overhead incurred for index match. This is because match performance is determined by the size of the index graph. This size depends on the structures exhibited by the data but not the actual size of the data. For instance, the fact that the size of $G^{\sim}_{\text{2-d}}$ for LUBM10 is smaller than for LUBM5 (see Table 2) simply tells that LUBM5 exhibits higher structural complexity (when only paths of maximum length 2 are considered). Thus, the index match time does not necessarily increase when the data graph becomes larger. The positive effect of data filtering and query pruning (load and join) however, correlates with the data size.

5.3 The Results of Optimization

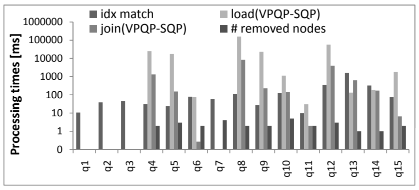
The structure index may be large for datasets that have diverse structures. Also, SQP does not perform well on queries that are simple and (or) do not lend themselves to



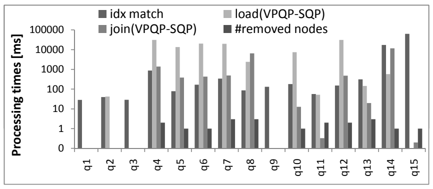
(a) Total times for DBLP.



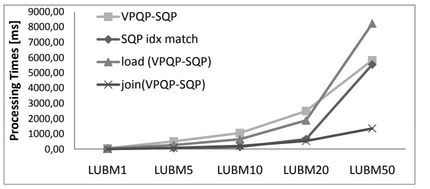
(b) Total times for LUBM50.



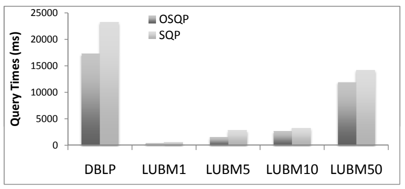
(c) Overhead vs. gain for DBLP.



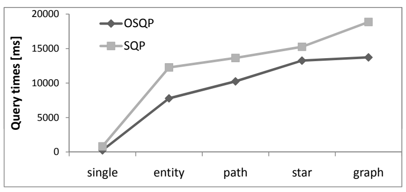
(d) Overhead vs. gain for LUBM50.



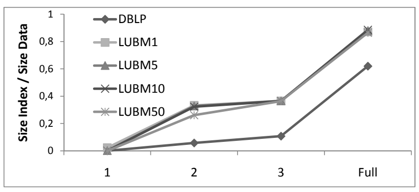
(e) Performance for LUBM datasets.



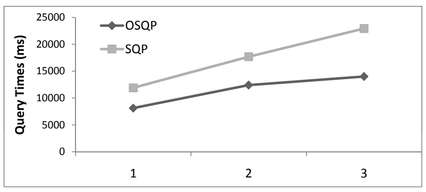
(f) Average SQP and OSQP times.



(g) Average SQP and OSQP times.



(h) Ratio of structure index size to data size.



(i) Times for different bisimulation heights.

Fig. 7: Evaluation results for structure-oriented approach (SQP), its optimization (OSQP) and VP.

pruning. We will discuss the effects of the further optimizations, which aim to address these problems. Namely, we show how parameterization can help to obtain indexes with a manageable size (especially when height is set to $n = 1$), and that the integrated strategy can leverage standard data-level processing in cases where structure-level processing cannot be successfully applied.

Queries. For this part, we use a query generation procedure to obtain a larger number of test queries, which vary in the maximum number of constants, paths, cycles, and the maximum path length. It is run for both DBLP and LUBM to obtain 160 queries of the 5 types already discussed.

The Effect of the Integrated Strategy. We compare SQP with OSQP. Fig. 7f shows the average time for the 160 queries. Clearly, OSQP consistently outperformed SQP. The average time obtained for OSQP was lower than SQP for all datasets, up to 50 percent in the case of LUBM5.

We provide a breakdown of the average time to understand the effect of query structure (and complexity). In Fig. 7g, we show the average time for the different query types. For both approaches, query processing times increased with the complexity of the query. For single triple patterns, OSQP slightly improved SQP. In fact, it equals VPQP because in this case, only data-level operations were performed. The largest difference could be observed for entity queries. Improvement could also be observed for more complex structures, i.e., graph-shaped queries.

The Effect of Bisimulation Height. Fig. 7h shows the ratio of structure index size to data size at various bismulation heights $n = 1, 2, 3, Full$. Clearly, for all datasets, the relative index size largely increased with n.

To understand the effect of this parameter on query processing, we run both OSQP and SQP on structure indexes with different heights. Fig. 7i shows the average times for $n = 1, 2, 3$. They increased with the bisimulation height. For greater height, the structure index size increased, thus resulting in higher overhead for evaluating structure-level joins. While the prunable parts become larger, it seems that the effect of pruning could not compensate the overhead when the index is too large. For the datasets in the experiments, using structure index with $n = 1$ achieved best performance.

5.4 Comparison to Triple Stores

We now describe the second part of the evaluation, where we compare our solution to full-fledged triple stores, namely RDF3-X and Sesame.

Systems. In order to support queries with more advanced access patterns (e.g. unbound predicates), full-fledged RDF stores such as *Sesame*[5] and *RDF-3X*[6] typically employ additional indexes. Furthermore, they implement advanced optimization techniques such as index compression, materialized join indexes, and query plan optimization that are largely orthogonal to our solution [23]. For instance, RDF-3X employs multiple B+-tree indexes to support lookups on all possible triple patterns. By using separate indexes for each triple pattern, RDF-3x is able to provide sorted access, thereby allowing for efficient merge joins without sorting at runtime. Sesame also uses B+-tree indexes to store data using dictionary encoding. We configured these systems to use all indexes needed to support the queries in the experiments (SPO, POS, OSP).

To make the comparison more fair, we extended OSQP with some optimization techniques also used by these

5. Sesame 2.6.1, http://www.openrdf.org/
6. RDF3-X 0.3.7, https://code.google.com/p/rdf3x/

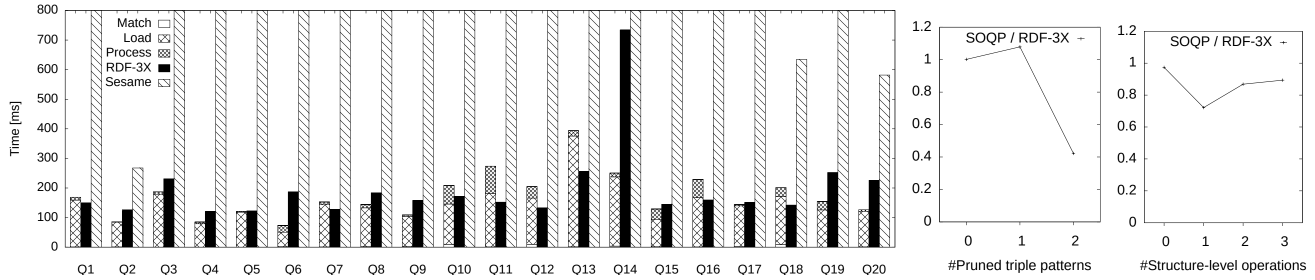


Fig. 8: a) Query times for OSQP, RDF-3X and Sesame. The query times for OSQP are split into index matching, data loading and processing. Query times for Sesame are cut off at 800 ms (Sesame's average is 13 s); b) Ratio of OSQP vs. RDF-3X query times in relation to number of pruned triple patterns; c) Ratio in relation to number of structure-level operations.

stores. Similar to RDF-3X and Sesame, we also used dictionary encoding to manage the data and indexes. In particular, this version of OSQP employs a sparse index and page-level compression using the Snappy algorithm[7]. It uses hash and merge join operators, and employs join optimizations similar to RDF-3X (sideways information passing).

Datasets and Indexes. In order to see how our approach performs w.r.t. more heterogeneous datasets, the Crossref Domain (DBPedia, LinkedMDB, Geonames, New York Times, Semantic Web Dog Food, Jamendo) and Life Science Domain (DBPedia, KEGG, Drugbank, ChEBI) dataset collections from the FedBench [24] benchmark were used in this experiment. These datasets cover a wide range of domains and are representative of RDF data as they are used in real-world applications. More information about them can be found on the benchmark website[8].

The dataset used in this experiment consists of about 167M triples. The structure index $G^{\sim}_{L_1,L_2}$ created for this (with $n = 1$) contains only 136,566 edges.

Queries. We used modified FedBench queries as some of them contain operators that are not supported (such as UNION and FILTER). From the Cross Domain, Life Science and Linked Data query sets, we obtained 20 queries that differ in sizes (number of triple patterns) and prunability.

Overall Results. Fig. 8a shows the query times for all queries for all three systems. The measurements for OSQP are again split into the times for structure index matching, data loading and (join) processing. Overall, we see that OSQP and RDF-3X both outperformed Sesame by a large margin. One reason for this seems to be that Sesame decoded dictionary encoded values directly after loading from disk while the other systems process encoded values, thereby requiring less disk accesses. OSQP on average performed slightly better than RDF-3X. The average query time for OSQP is 169.61 ms, compared to 191.89 ms for RDF-3X and 13059.24 ms for Sesame.

However, individual queries varied a lot in the comparison between OSQP and RDF-3X. For example, for queries Q6 and Q14, OSQP outperformed RDF-3X by factors of 2.55 and 2.96, respectively. For both queries, OSQP was able to prune query patterns (one for Q6 and two for Q14) and could apply structure-level operations. On the other hand, queries such as Q11 and Q13 could not be pruned, and consequently, were executed faster by RDF-3X than OSQP. We found that while the amount of data required were comparable between these two systems (OSQP relied mostly on data-level processing) in these cases, RDF-3X was still more efficient in processing the data. In fact, also the join implementations used by these systems are conceptually the same in these cases. However, RDF-3X seems to benefit from more optimized data structures (and faster execution of the C++ code), compared to our Java-based implementation.

In Fig. 8a we see that query times for OSQP were dominated by data loading (86% of total time on average). Structure index matching and join processing had a relatively small impact on query times for most queries. The structure index was small enough to be kept in memory, leading to an average of 2.47 ms (0.02%) for structure index matching. Join processing on average took 23.28 ms (13.8%). For a few queries, it took a larger fraction of total query time. For query Q11, join processing took 92 ms (51% of 180.5 ms total time). The triple patterns in this query were not very selective and only a single merge join could be used while the other joins were performed as hash joins. Here, RDF-3X performed better.

Effect of Structure-level Processing. Fig. 8b+c compare query times of OSQP and RDF-3X by showing the ratio of OSQP and RDF-3X query times for two query properties: a) the number of pruned triple patterns and b) the number of structure-level operations performed during query processing. In Fig 8b we see that when there was no or little pruning possible, query times of OSQP and RDF-3X were roughly equal. However, when OSQP was able to prune two triple patterns, query time decreased to 42%.

Fig. 8c shows the ratio between OSQP and RDF-3X for different numbers of structure-level operations applied by OSQP during query processing. These include pruned query patterns and query patterns that were not pruned but evaluated on structure-based partitioned data (i.e., using matches from the structure index). Overall, we can see that OSQP performed better when structure-level opera-

7. https://code.google.com/p/snappy-java/
8. http://code.google.com/p/fbench/

tions were performed. When no structure-level operations were applied, OSQP times were roughly equal to RDF-3X (97%), whereas OSQP performed better when one or more such operations could be used (72%-89%).

6 Related Work

We have already pointed out specific differences to VP, the related partitioning and query processing techniques used in existing triple stores [1]. Standard (data-level) processing [1], [7] is used for matching query patterns against graphs (same data access and join implementation). However, additional to processing query patterns against the data graph, our solution involves structure-level matching and an integrated strategy that interleaves data- and structure-level operations.

Now, we focus on related work in the field of graph-structured, XML and semi-structured data management.

Graph Indexing This is a popular topic in the graph database community. A well-studied technique is based on the idea of grouping graphs based on their contained subgraphs. In [25] for instance, frequent substructures are identified and used for indexing graph databases (such as chemical databases containing molecule data). With such an approach, the complexity of subgraph matching inevitably leads to scalability problems for larger data sets, which can be partially overcome by essentially restricting to tree-shaped substructures [26]. As a crucial difference to our approach, these methods aim at indexing sets of graphs while we are primarily concerned with query processing over one single large data graph.

Structure Index This concept has been extensively studied for indexing semi-structured and XML data ([12], [13], [14], [15]). *Dataguide* [27] is a well-known concept that has been proposed for rooted graphs, i.e., graphs having one distinguished root node from which all other vertices can be reached through directed paths. A strong dataguide is established by grouping together nodes sharing edge label sequences of incoming paths starting from the root. The technique is similar to the conversion of a nondeterministic finite automaton into an equivalent deterministic automaton. As opposed to our approach, the grouping in dataguides is not a partition, i.e., one data element may be assigned to several groups. The advantage is that for any labeled path starting from the root, there is exactly one associated node in the dataguide. However, the size of the dataguide can get exponentially larger than that of the original data graph. The index proposed in [12] can avoid this worst-case exponential blow-up of dataguides. The strategy of constructing this index is similar to minimizing a nondeterministic finite automaton. To further reduce the index size, the *A(k)-Index* [14] has been proposed, which relaxes the equivalence condition to consider only incoming paths whose lengths are no longer than k. Further, the *D(k)-Index* allows for adjusting k to the query load [15]. Instead of backward bisimulation only, both back- and forward bisimulation is employed for the construction of a covering index for XML branch queries [13].

The differences to the structure index employed in this scenario are as follows: its concept of structural similarity is more fine grained, as it rests on characterizing the structure of an element via trees instead of paths [14], [13]; it can be parameterized not only based on height [14] but also based on the edge labels L_1 and L_2.

Recently, a bisimulation-based structure index has also been proposed for RDF [28]. The idea there is to group nodes by their labeled neighborhoods. However, the bisimulation is actually defined on a neighborhood with labeled nodes and unlabeled edges. Our approach is rather focused on the structure defined by the edge labels. Further, we propose a principled way to parameterize the index construction so that the index size can be controlled.

Query Processing Our structure-aware processing technique is similar to work on *evaluating path queries* using structure indexes such as dataguides [27]. The procedure proposed in [29] for instance is similar to structure-level processing. Firstly, the set of index nodes is computed. Then, the union of all data elements associated with these index matches are returned as answers. Since the queries we consider are graph-structured, structure-level processing is not sufficient. Our technique involves the additional steps of query pruning and data-level processing.

There are techniques for finding occurrences of *twig patterns* (tree-structures) in XML data ([30], [31]). Typically, the twig pattern is decomposed into binary relationships, i.e., parent-child and ancestor-descendant. Matching is achieved in two main steps: (1) match binary relationships against XML data and combine them using structural join algorithms to obtain basic matches and (2) combine basic matches to obtain final answer. For this first step, a variation of the traditional merge join algorithm has been proposed to deal with "structure predicates", i.e., the *multi-predicate merge join* (MPMJ) [30]. The main problem with MPMJ is that it may generate unnecessary intermediate results such that join results of individual binary relationships might not appear in the final results. Bruno et al. proposed a method called *TwigJoin* for solving this problem [31]. It outputs a list of element paths, where each matches one root-to-leaf path of the twig pattern. When there are only ancestor-descendant edges, this algorithm is optimal such that each of these matches is part of the final answer.

These XML-based techniques rely on tree structures, and in particular, assume that relationships among elements are either parent-child or ancestor-descendant. They are not applicable to our setting, where both query and data are graph structured, and different edge labels have to be considered for matching.

7 Conclusion

We propose the use of a structure index for dealing with semi-structured RDF data on the Web. Based on this concept, we proposed structure-oriented data partitioning and query processing which effectively reduces I/O and the number of joins. Compared to state-of-the-art data partitioning and query processing, our solution achieves several times faster performance w.r.t. a given workload. For complex structured queries with bound predicates, it provides superior performance w.r.t. full-fledged RDF engines. More importantly, the proposed solution shows promising scalability results, i.e., performance improvement increases more than linearly with the size of the data. This is because its performance does not strictly correlate with data size but depends on the heterogeneity of structure patterns exhibited by the data.

References

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[2] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient rdf storage and retrieval in jena2," in *SWDB*, 2003, pp. 131–150.

[3] A. Harth and S. Decker, "Optimized index structures for querying rdf from the web," in *LA-WEB*, 2005, pp. 71–80.

[4] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

[5] B. Liu and B. Hu, "Path queries based rdf index," in *SKG'05*, 2005, pp. 91–91.

[6] O. Udrea, A. Pugliese, and V. Subrahmanian, "Grin: a graph based rdf index," in *AAAI*, vol. 22, no. 2, 2007.

[7] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[8] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *ICDE*, 2011, pp. 984–994.

[9] H. Huang and C. Liu, "Estimating selectivity for joined rdf triple patterns," in *CIKM*, 2011, pp. 1435–1444.

[10] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto, "Compressed k2-triples for full-in-memory rdf engines," in *AMCIS*, 2011.

[11] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "bit" loaded: a scalable lightweight join query processor for rdf data," in *WWW*, 2010, pp. 41–50.

[12] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu, "Adding structure to unstructured data," in *ICDT*. Springer Verlag, 1997, pp. 336–350.

[13] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries," in *SIGMOD Conference*, 2002, pp. 133–144.

[14] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *ICDE*, 2002, pp. 129–140.

[15] C. Qun, A. Lim, and K. W. Ong, "D(k)-index: An adaptive structural summary for graph-structured data," in *SIGMOD Conference*, 2003, pp. 134–144.

[16] J.-C. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Science of Computer Programming*, vol. 13, pp. 13–219, 1989.

[17] R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM J. Comput.*, vol. 16, no. 6, pp. 973–989, 1987.

[18] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy, "Updates for structure indexes," in *VLDB*, 2002, pp. 239–250.

[19] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in *WWW*, 2008, pp. 595–604.

[20] X. Yan and J. Han, "Closegraph: mining closed frequent graph patterns," in *KDD*, 2003, pp. 286–295.

[21] X. Dong and A. Y. Halevy, "Indexing dataspaces," in *SIGMOD Conference*, 2007, pp. 43–54.

[22] "Efficient rdf data management using structure indexes for general graph structured data," in *Technical Report available at https://sites.google.com/site/kimducthanh/publication/StrucStoreTR.pdf*.

[23] T. Neumann and G. Weikum, "x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases," *PVLDB*, vol. 3, no. 1, pp. 256–263, 2010.

[24] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran, "Fedbench: A benchmark suite for federated semantic data query processing," in *International Semantic Web Conference (1)*, 2011, pp. 585–600.

[25] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *SIGMOD Conference*, 2004, pp. 335–346.

[26] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: Tree + delta >= graph," in *VLDB*, 2007, pp. 938–949.

[27] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB*, 1997, pp. 436–445.

[28] S. Khatchadourian and M. P. Consens, "Explod: Summary-based exploration of interlinking and rdf usage in the linked open data cloud," in *ESWC (2)*, 2010, pp. 272–287.

[29] T. Milo and D. Suciu, "Index structures for path expressions," in *ICDT*, 1999, pp. 277–295.

[30] 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*, 2001, pp. 425–436.

[31] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal xml pattern matching," in *SIGMOD Conference*, 2002, pp. 310–321.

Thanh Tran worked as consultant and software engineer for IBM and Capgemini, and served as vising assistant professor at Stanford University. He has been mainly working on the topics of semantic data management, keyword search on semantic data, and semantic search. His interdisciplinary work is published in over 40 top-level journals and conference proceedings and earned the second prize at the Billion Triple Challenge 2008 as well as the best Linked Data paper at ESWC 2011. He is currently working as assistant professor, managing the projects iGreen and CollabKloud at the AIFB institute of the Karlsruhe Institute of Technology.

Günter Ladwig is a Research Associate at the Institute AIFB at the Karlsruhe Institute of Technology. He holds a Diploma in Computer Science from the same institution. His research interest include query processing, query optimization and keyword search in the context of semantic data management. His work is published in numerous conference proceedings (CIKM, ISWC, ESWC, DEXA) and earned the best Linked Data paper award at ESWC 2011.

Sebastian Rudolph is a Reader at the Institute AIFB at Karlsruhe Institute of Technology (KIT). He received his PhD in mathematics from Dresden University of Technology in 2006 and his habilitation in computer science from KIT in 2011. His active research interests, including algebra, complexity theory, logic, machine learning, optimization, database theory, and computational linguistics are witnessed by over 60 peer-reviewed scientific publications at high-level international conferences and journals. Sebastian coauthored two textbooks on the foundations of the Semantic Web and was a member of the W3C OWL Working Group. He serves on the editorial board of the Journal on Data Semantics and on the steering committees of the international conferences RR, ICFCA, ICCS, and DL.