A Structural Approach to Indexing Triples

François Picalausa[1], Yongming Luo[2], George H.L. Fletcher[2], Jan Hidders[3], and Stijn Vansummeren[1]

[1] Université Libre de Bruxelles, Belgium
{fpicalau,stijn.vansummeren}@ulb.ac.be
[2] Eindhoven University of Technology, The Netherlands
{y.luo,g.h.l.fletcher}@tue.nl
[3] Delft University of Technology, The Netherlands
{a.j.h.hidders}@tudelft.nl

Abstract. As an essential part of the W3C's semantic web stack and linked data initiative, RDF data management systems (also known as triplestores) have drawn a lot of research attention. The majority of these systems use *value-based indexes* (e.g., B^{+}-trees) for physical storage, and ignore many of the structural aspects present in RDF graphs. *Structural indexes*, on the other hand, have been successfully applied in XML and semi-structured data management to exploit structural graph information in query processing. In those settings, a structural index groups nodes in a graph based on some equivalence criterion, for example, indistinguishability with respect to some query workload (usually XPath). Motivated by this body of work, we have started the SAINT-DB project to study and develop a native RDF management system based on structural indexes. In this paper we present a principled framework for designing and using RDF structural indexes for practical fragments of SPARQL, based on recent formal structural characterizations of these fragments. We then explain how structural indexes can be incorporated in a typical query processing workflow; and discuss the design, implementation, and initial empirical evaluation of our approach.

1 Introduction

As an essential part of the W3C's semantic web stack, the RDF data model is finding increasing use in a wide range of web data management scenarios, including linked data[1]. Due to its increasing popularity and application, recent years have witnessed an explosion of proposals for the construction of native RDF data management systems (also known as triplestores) that store, index, and process massive RDF data sets.

While we refer to recent surveys such as [12] for a full overview of these proposals, we can largely discern two distinct classes of approaches. *Value-based approaches* focus on the use of robust relational database technologies such as B^{+}-trees and column-stores for the physical indexing and storage of massive

[1] `http://linkeddata.org/`

RDF graphs, and employ established relational database query processing techniques for the processing of SPARQL queries [1,7,15,18,22]. While value-based triplestores have proven successful in practice, they mostly ignore the native graph structure like paths and star patterns that naturally occur in RDF data sets and queries. (Although some value-based approaches consider extensions to capture and materialize such common patterns in the data graph [1,15].)

Graph-based approaches, in contrast, try to capture and exploit exactly this richer graph structure. Examples include GRIN [20] and DOGMA [6], that propose index structures based on graph partitioning and distances in the graphs, respectively. A hybrid approach is taken in dipLODocus[RDF], where value-based indexes are introduced for more or less homogeneous sets of subgraphs [23]. These somewhat ad-hoc approaches work well for an established query workload or class of graph patterns, but it is unclear how the indexed patterns can flexibly support general SPARQL queries outside of the supported set.

Structural indexes have been successfully applied in the semi-structured and XML data management context to exploit structural graph information in query processing. A structural index is essentially a reduced version of the original data graph where nodes have been merged according to some notion of structural similarity such as bisimulation [4,5,9,13]. These indexes effectively take into account the structure of both the graph and query, rather then just the values appearing in the query as is the case for value-based indexes. Furthermore, the success of structural indexes hinges on a precise coupling between the expressive power of a general query language and the organization of data by the indexes [9]. The precise class of queries that they can support is therefore immediately clear, thereby addressing the shortcomings of other graph-based approaches.

While structural indexes have been explored for RDF data, for example in the Parameterizable Index Graph [19] and gStore [24] proposals, these proposals simplify the RDF data model to that of *resource-centric* edge-labeled graphs over a *fixed* property label alphabet (disallowing joins on properties), which is not well-suited to general SPARQL query evaluation where pattern matching is *triple-centric*, i.e., properties have the same status as subjects and objects. The relevance of such queries is observed by studies of the usage of SPARQL in practice [3,16]. Furthermore, there is no tight coupling of structural organization of these indexes to the expressivity of a practical fragment of SPARQL.

Motivated by these observations, we have initiated the SAINT-DB (Structural Approach to INdexing Triples DataBase) project to study the foundations and engineering principles for native RDF management systems based on structural indexes that are faithful to both the RDF data model and the SPARQL query language. As a initial foundation, we have recently established a precise structural characterization of practical SPARQL fragments in terms of graph simulations [8]. Our goal in SAINT-DB is to leverage this characterization in the design of native structural indexing solutions for massive RDF data sets.

Contributions and Overview. In this article, we report on our first results in SAINT-DB. In particular, we make the following contributions: **(1)** A new notion of structural index for RDF data is introduced that, reflecting the SPARQL

t_1 :	(*sue*,	*type*,	*CEO*)	t_{10} :	(*reportsTo*,	*type*,	*socialRel*)
t_2 :	(*crispin*,	*type*,	*VP*)	t_{11} :	(*friendOf*,	*type*,	*socialRel*)
t_3 :	(*sue*,	*manages*,	*joe*)	t_{12} :	(*knows*,	*type*,	*socialRel*)
t_4 :	(*joe*,	*reportsTo*,	*jane*)	t_{13} :	(*bestFriendOf*,	*type*,	*socialRel*)
t_5 :	(*jane*,	*friendOf*,	*lucy*)	t_{14} :	(*dislikes*,	*type*,	*socialRel*)
t_6 :	(*crispin*,	*knows*,	*larry*)	t_{15} :	(*yonei*,	*knows*,	*yongsik*)
t_7 :	(*larry*,	*bestFriendOf*,	*sarah*)	t_{16} :	(*yongsik*,	*reportsTo*,	*tamae*)
t_8 :	(*sarah*,	*dislikes*,	*hiromi*)	t_{17} :	(*kristi*,	*manages*,	*filip*)
t_9 :	(*manages*,	*type*,	*socialRel*)	t_{18} :	(*filip*,	*bestFriendOf*,	*sriram*)

Fig. 1. A small RDF graph, with triples labeled for ease of reference

language, contains complete triple information and therefore allows for the retrieval of sets of triples rather than sets of resources. **(2)** A formalization of the structural index, coupled to the expressivity of practical fragments of SPARQL, is given, together with the algorithms for building and using it. **(3)** We demonstrate the effective integration of structural indexing into a state-of-the-art triple store with cost-based query optimization.

We proceed as follows. In Sec. 2 we introduce our basic terminology for querying RDF data. In Sec. 3 we present the principles behind triple-based structural indexes for RDF. In Sec. 4 we then discuss how these principles can be put into practice in a state of the art triple store. In Sec. 5, we present an empirical study where the effectiveness of the new indices within this extended triple store is demonstrated. Finally, in Sec. 6 we present our main conclusions and give indications for further research.

2 Preliminaries

RDF. All information in RDF is uniformly represented by triples of the form (s, p, o) over some fixed but unspecified universe $\mathcal{U}$, $(s, p, o) \in \mathcal{U}^3$. Here, s is called the *subject*, p is called the *predicate*, and o is called the *object.* An *RDF graph* D is a finite set of *RDF* triples, $D \subseteq \mathcal{U}^3$. To illustrate, a small RDF graph of social relationships in a corporate setting is given in Fig. 1.

BGP Queries. RDF comes equipped with the SPARQL [17] language for querying data in RDF format. Using so-called *basic graph patterns* (BGPs for short) as building blocks, SPARQL queries search for specified subgraphs of the input RDF graph. While SPARQL queries can be more complex in general, we will focus in this article on so-called *BGP queries*: SPARQL queries that consist of basic graph patterns only. The reason for this is threefold. First and foremost, the evaluation of basic graph patterns is a problem that occurs as a subproblem in all SPARQL query evaluation problems. Second, BGP queries correspond to the well-known class of conjunctive queries from relational databases. Third, recent analysis of real-world SPARQL query logs has illustrated that the majority of SPARQL queries posed in practice are BGP queries [3, 16].

The formal definition of BGP queries is as follows. Let $\mathcal{V} = \{?x, ?y, ?z, \ldots\}$ be a set of variables, disjoint from $\mathcal{U}$. A *triple pattern* is an element of $(\mathcal{U} \cup \mathcal{V})^3$. We write $vars(p)$ for the set of variables occurring in triple pattern p. A *basic graph pattern* (BGP for short) is a set of triple patterns. A *BGP query* (or simply *query* for short) is an expression Q of the form SELECT X WHERE P where P is a BGP and X is a subset of the variables mentioned in P.

Example 1. As an example, the following BGP query retrieves, from the RDF graph of Fig. 1, those pairs of people p_a and p_c such that p_a is a CEO and p_c has a social relationship with someone directly related to p_a.

$$\begin{array}{ll} \text{SELECT} & ?p_a, ?p_c \\ \text{WHERE} & \{\ (?p_a, \mathit{type}, \mathit{CEO}), (?p_a, ?\mathit{rel}_{ab}, ?p_b), (?p_b, ?\mathit{rel}_{bc}, ?p_c), \\ & \quad (?\mathit{rel}_{ab}, \mathit{type}, \mathit{socialRel}), (?\mathit{rel}_{bc}, \mathit{type}, \mathit{socialRel})\} \end{array} \qquad \square$$

To formally define the semantics of triple patterns, BGPs, and BGP queries, we need to introduce the following concepts. A *mapping* μ is a partial function $\mu\colon \mathcal{V} \to \mathcal{U}$ that assigns values in $\mathcal{U}$ to a finite set of variables. The domain of μ, denoted by $dom(\mu)$, is the subset of $\mathcal{V}$ where μ is defined. The restriction $\mu[X]$ of μ to a set of variables $X \subseteq \mathcal{V}$ is the mapping with domain $dom(\mu) \cap X$ such that $\mu[X](?x) = \mu(?x)$ for all $?x \in dom(\mu) \cap X$. Two mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, when for all common variables $?x \in dom(\mu_1) \cap dom(\mu_2)$ it is the case that $\mu_1(?x) = \mu_2(?x)$. Clearly, if μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ is again a mapping. We define the join of two sets of mappings Ω_1 and Ω_2 as $\Omega_1 \bowtie \Omega_2 := \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$, and the projection of a set of mappings Ω to $X \subseteq \mathcal{V}$ as $\pi_X(\Omega) := \{\mu[X] \mid \mu \in \Omega\}$. If p is a triple pattern then we denote by $\mu(p)$ the triple obtained by replacing the variables in p according to μ. Semantically, triple patterns, BGPs, and queries evaluate to a set of mappings when evaluated on an RDF graph D:

$$\begin{aligned} [\![p]\!]_D &:= \{\mu \mid dom(\mu) = vars(p) \text{ and } \mu(p) \in D\}, \\ [\![\{p_1, \ldots, p_n\}]\!]_D &:= [\![p_1]\!]_D \bowtie \cdots \bowtie [\![p_n]\!]_D, \\ [\![\text{SELECT } X \text{ WHERE } P]\!]_D &:= \pi_X([\![P]\!]_D). \end{aligned}$$

Example 2. Let Q be the query of Example 1 and D be the dataset of Fig. 1. Then $[\![Q]\!]_D = \{\langle ?p_a \mapsto \mathit{sue}, ?p_c \mapsto \mathit{jane}\rangle\}$. In other words, Q evaluated on D contains a single mapping μ, where $\mu(?p_a) = \mathit{sue}$ and $\mu(?p_c) = \mathit{jane}$. $\square$

3 Principles of Triple-Based Structural Indexing

To evaluate a BGP $P = \{p_1, \ldots, p_n\}$ on an RDF graph D we need to perform $n - 1$ joins $[\![p_1]\!]_D \bowtie \cdots \bowtie [\![p_n]\!]_D$ between subsets of D. Since D is large in practice, we are interested in pruning as much as possible the subsets $[\![p_i]\!]_D$ of D that need to be joined, where $1 \leq i \leq n$. In particular, we are interested in efficiently removing from $[\![p_i]\!]_D$ any "dangling" triples that do not participate in the full join. Towards this purpose, we next introduce the notions of equality type and structural index.

Definition 1. *An* equality type *is a set of pairs (i,j) with $1 \leq i,j \leq 3$. Intuitively, a pair (i,j) in an equality type indicates the position in which two triples share a common value. In particular, let $t = (t_1, t_2, t_3)$ and $u = (u_1, u_2, u_3)$ be two RDF triples or two triple patterns. Then the* equality type of t and u, *denoted $eqtp(t,u)$, is defined as $eqtp(t,u) := \{(i,j) \mid t_i = u_j \text{ and } 1 \leq i,j \leq 3\}$.*

Essentially, the equality type of t and u specifies the kinds of natural joins that t and u can participate in. For example, when evaluating the BGP $\{(?x,?y,1), (?z,?x,?y)\}$ we are looking for triples t and u such that $\{(1,2),(2,3)\} \subseteq eqtp(t,u)$. This is a necessary condition in general for a mapping to be in a BGP result:

$$\mu \in [\![P]\!]_D \quad \Rightarrow \quad eqtp(p,q) \subseteq eqtp(\mu(p),\mu(q)) \text{ for all } p,q \in P.$$

Intuitively speaking, a structural index (defined below) groups triples into index blocks, and summarizes the equality types that exist between triples in those blocks. The necessary condition above can then be used to prune triples that can never realize the desired equality type, by looking at the structural index only.

Definition 2 (Structural Index). *Let $\mathcal{T}$ denote the set of all equality types and let D be an RDF graph. A* structural index *for D is an edge-labeled graph $I = (N,E)$ where N is a finite set of nodes, called the* blocks *of the index, and $E \subseteq N \times \mathcal{T} \times N$ is a set of edges labeled by equality types. The nodes N of I must be sets of triples in D (i.e., $N \subseteq 2^D$), and must form a partition of D. We write $[t]_I$ to denote the unique block of I that contains $t \in D$. Furthermore, it is required that E reflects the equality types between the triples in its blocks, in the sense that for all $t,u \in D$ we must have $([t]_I, eqtp(t,u), [u]_I) \in E$.*

An embedding *of a BGP P into a structural index I is a function $\alpha\colon P \to N$ that assigns to each triple pattern $p \in P$ a node $\alpha(p) \in N$ such that for every $p,q \in P$ there exists $\tau \in \mathcal{T}$ with $eqtp(p,q) \subseteq \tau$ and $(\alpha(p),\tau,\alpha(q)) \in E$.*

Example 3. Consider the graph shown in Fig. 2, where nodes are labeled with triples from the dataset D of Fig. 1, and, for clarity of presentation, self-loops (all labeled by $\{(1,1),(2,2),(3,3)\}$), symmetric edges (e.g., there is also a $\{(1,3)\}$ edge from n_3 to n_2), and transitive edges (e.g., there are also $\{(2,2)\}$ edges between n_1 and n_6 and n_1 and n_7) have been suppressed. The reader is invited to verify that this graph is indeed a structural index for D, and that there is only one embedding α of the BGP of query Q of Example 1 into this structural index. In particular, α assigns the triple patterns $(?p_a, ?rel_{ab}, ?p_b)$ and $(?p_b, ?rel_{bc}, ?p_c)$ of Q to index nodes n_2 and n_3, respectively. Whereas in the absence of structural information, these triple patterns individually can match any triple of D, α restricts their possible bindings to a small fraction of D, a significant reduction in search space for evaluating Q on D. □

3.1 Query Processing with Structural Indexes

The following proposition (proof omitted) establishes in general this connection between query embeddings in a structural index and query evaluation on a dataset.

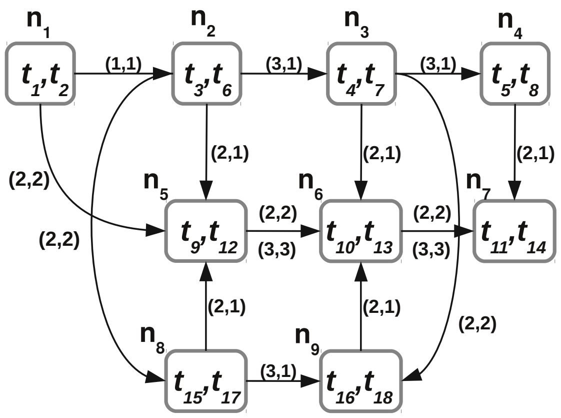

Fig. 2. A structural index for the RDF graph of Fig. 1. As described in Example 3, a few edges have been suppressed for clarity of presentation.

Proposition 1. *Let $P = \{p_1, \ldots, p_n\}$ be a BGP, let D be a dataset, let I be a structural index for D and let A be the set of all embeddings of P into I. Then*

$$\begin{aligned} [\![P]\!]_D &= \bigcup_{\alpha \in A} [\![p_1]\!]_{\alpha(p_1)} \bowtie \cdots \bowtie [\![p_n]\!]_{\alpha(p_n)} \\ &= [\![p_1]\!]_{\bigcup_{\alpha\in A} \alpha(p_1)} \bowtie \cdots \bowtie [\![p_n]\!]_{\bigcup_{\alpha\in A} \alpha(p_n)} \end{aligned}$$

This proposition indicates two natural ways we can use a structural index I to alternatively compute $[\![P]\!]_D$:

(M1). First compute the set A of all embeddings of P into I. Ideally, I is small enough so that finding these embeddings is computationally fast. For each $\alpha \in A$ we join $[\![p_1]\!]_{\alpha(p_1)} \bowtie \cdots \bowtie [\![p_n]\!]_{\alpha(p_n)}$, and add the result to the output. Note that, since $\alpha(p_i) \subseteq D$ for each $1 \leq n$, also $[\![p_i]\!]_{\alpha(p_i)} \subseteq [\![p_i]\!]_D$. Potentially, therefore, we compute joins on smaller relations than when computing $[\![p_1]\!]_D \bowtie \cdots \bowtie [\![p_n]\!]_D$. Nevertheless, we risk computing many such joins (as many as there are embeddings of P into I).

(M2). To circumvent this problem, we can alternatively compute, for each i, the subset $D_i = \bigcup_{\alpha\in A} \alpha(p_i)$ of D, and then join $[\![p_1]\!]_{D_1} \bowtie \cdots \bowtie [\![p_n]\!]_{D_n}$. This requires computing the join only once, but on larger subsets of D.

We will empirically validate the effectiveness of these methods in Section 5.2.

3.2 Index Construction

A crucial assumption in the query processing strategies (M1) and (M2) outlined above is that the structural index I is small enough to efficiently compute embeddings on, yet detailed enough to ensure that candidate triples that cannot participate in the required joins are pruned. Indeed, note that computing all embeddings of P in the trivial index I in which each block consists of a single triple will be as hard as computing the result of P on D itself. On the other hand, while it is trivial to compute all embeddings of P into the other trivial index J in which all triples are kept in a single block, we always have $\alpha(p_i) = D$ and hence no pruning is achieved.

We next outline a method for constructing structural indexes that are guaranteed to have *optimal* pruning power for the class of so-called *pure acyclic* BGPs, in the following sense.

Definition 3 (Pruning-optimal). *A structural index I for RDF graph D is* pruning-optimal *w.r.t. BGP P if, for every $p \in P$, we have $\pi_{vars(p)}[\![P]\!]_D = [\![p]\!]_{\bigcup_{\alpha \in A} \alpha(p)}$, where A is the set of all embeddings of P in I. Index I is pruning-optimal w.r.t. a class of BGPs $\mathcal{C}$ if I is pruning-optimal w.r.t. every $P \in \mathcal{C}$.*

Note that the inclusion $\pi_{vars(p)}[\![P]\!]_D \subseteq [\![p]\!]_{\bigcup_{\alpha \in A} \alpha(p)}$ always holds due to Prop. 1. The converse inclusion does not hold in general, however.

Stated differently, pruning-optimality says that *every* element in $[\![p]\!]_{\alpha(p)}$ can be extended to a matching in $[\![P]\!]_D$, for every triple pattern $p \in P$ and every embedding α of P into I. Hence, when using Prop. 1 to compute $[\![P]\!]_D$ we indeed optimally prune each relation $[\![p_i]\!]_D$ to be joined.

As already mentioned, we will give a method for constructing structural indexes that are pruning-optimal w.r.t. the class of so-called *pure acyclic* BGPs. Here, purity and acyclicity are defined as follows.

Definition 4. *A BGP P is* pure *if it contains only variables, i.e., if $P \subseteq \mathcal{V}^3$.*

The restriction to pure BGPs is motivated by the following proposition, stating that pruning-optimal indexes do not always exist for non-pure BGPs. Intuitively, this is due to the fact that structural indexes only contain information about the joins that can be done on an RDF graph D, but do not contain any information about the universe values present in D. Since non-pure BGP *do* query for these universe values, structural indexes do not have enough information to ensure that for every α, every $p, q \in P$ and every $t \in [\![p]\!]_{\alpha(p)}$ there always exists a matching tuple $u \in [\![q]\!]_t$ that not only has the correct join type (i.e., $eqtp(p, q) \subseteq eqtp(t, u)$ but also fulfills the universe value constraints required by q (i.e., $u \in [\![q]\!]_D$).

Proposition 2. *Let p, q be distinct triple patterns with $eqtp(p, q) \neq \emptyset$ and $\{p, q\}$ not pure. There exists an RDF graph D such that any structural index I for D is not pruning-optimal w.r.t. P.*

The other restriction, acyclicity is a very well-known concept for relational select-project-join queries [2]. Its adaption to BGP queries is as follows.

Definition 5 (Acyclicity). *A BGP P is* acyclic *if it has a join forest. A join forest for P is a forest F (in the graph-theoretical sense) whose set of nodes is exactly P such that, for each pair of triple patterns p and q in P that have variables in common the following two conditions hold:*

1. *p and q belong to the same connected component of F; and*
2. *all variables common to p and q occur in every triple pattern on the (unique) path in F from p to q.*

The depth *of F is the length of the longest path between any two nodes in F. The* depth *of an acyclic BGP P is the minimum depth of a join forest for P.*

Recent analysis has illustrated that 99% of the BGP queries found in real-world SPARQL query logs are acyclic [16]. The class of acyclic BGPs is hence of practical relevance.

Similar to the way in which the concept of *graph bisimulation* (as used e.g., in modal logic and process calculi) is used to build structural indexes for semi-structured and XML databases and XPath-based query languages (e.g., [4,9]), our pruning-optimal index is obtained by grouping triples that are equivalent under the following notion of *guarded simulation.*

Definition 6 (Guarded simulation). *Let D be an RDF graph and let k be a natural number. We say that $u \in D$* simulates *$t \in D$* guardedly up to depth *k, denoted $t \preceq_k u$, if either (1) $k = 0$; or (2) if $k > 0$ there exists for every $t' \in D$ some $u' \in D$ such that $eqtp(t, u) \subseteq eqtp(t', u')$ and $t' \preceq_{k-1} u'$. We write $t \simeq_k u$ if $t \preceq_k u$ and $u \preceq_k t$. Finally, we write $t \simeq u$ if $t \simeq_k u$ for every k.*

Although space constraints prohibit us from discussing the origin of the above definition in detail (cf. [8]), readers familiar with the notion of graph simulation may note that the above notion of guarded simulation is equivalent to the graph simulation (up to depth k) of the edge-labeled graph $G = (D, \{(t, \tau, u) \in D \times \mathcal{T} \times D \mid \tau \subseteq eqtp(t, u)\})$ to itself. It follows immediately that efficient main-memory algorithms for computing the relations $\simeq_k$ and $\simeq$ hence exist [10,21].

Definition 7 (Simulation Index). *The* depth-k simulation index of RDF graph *D, denoted $\textsc{sim}_k(D)$, is the structural index $I = (N, E)$ for D such that*

– *N consists of the equivalence classes of $\simeq_k$, i.e., if we denote by $[t]_{\simeq_k}$ the set $\{u \in D \mid t \simeq_k u\}$ then $N = \{[t]_{\simeq_k} \mid t \in D\}$.*
– *$E = \{([t]_{\simeq_k}, \tau, [u]_{\simeq_k}) \mid t, u \in D, \tau = eqtp(t, u)\}$.*

The simulation index of *D, denoted $\textsc{sim}(D)$ is defined similarly, but then using $\simeq$ instead of $\simeq_k$.*

The following proposition (proof omitted) shows that simulation indexes are pruning-optimal with respect to the class of pure acyclic BGPs.

Proposition 3. *Let D be an RDF graph. $\textsc{sim}(D)$ is pruning-optimal w.r.t. the class of pure acyclic BGPs. Moreover, $\textsc{sim}_k(D)$ is pruning-optimal w.r.t the class of pure acyclic BGPs of depth at most k, for each k.*

Although pure BGPs are infrequent in practice, they are the only reasonable class of queries to couple queries with structural indexes from a theoretical point of view, as indicated by Prop. 2. This result hence shows that the $\textsc{sim}(D)$ and $\textsc{sim}_k(D)$ indexes allow one to take into account *precisely* the structural (join) information in the dataset.

4 Applying the Principles in Practice

In this section we discuss the design of SAINT-DB in which we have implemented the principles of Sec. 3. We start with a description of the triplestore upon which SAINT-DB is built.

RDF-3X. RDF-3X is a state-of-the-art, open source native RDF storage and retrieval system [15]. It is widely used by the research community and has, according to many previous studies, excellent query performance.

RDF-3X makes extensive use of B$^+$-trees as its core underlying data structure. In particular, it stores all (s, p, o) triples of the RDF graph in a (compressed) clustered B$^+$-tree in which the triples themselves act as search keys. This means that the triples are sorted lexicographically in the leaves of the B$^+$-tree, which allows the conversion of triple patterns into efficient range scans. For example, to compute $[\![(\mathit{jane}, \mathit{friendOf}, ?x)]\!]_D$ it suffices to search the B$^+$-tree using the prefix search key $(\mathit{jane}, \mathit{friendOf})$, and subsequently scan the relevant leaf pages to find all bindings for $?x$. RDF-3X actually employs this idea aggressively: to guarantee that not only triple patterns of the form $(\mathit{jane}, \mathit{friendOf}, ?x)$ can be answered by efficient range scans, but also triple patterns of the form $(?x, \mathit{friendOf}, \mathit{lucy})$, $(\mathit{jane}, ?x, ?y)$, and so on, it maintains all six possible permutations of subject (S), predicate (P) and object (O) in six separate indexes (corresponding to the sort orders SPO, SOP, PSO, ...). Compression of the B$^+$-tree leaf pages is used to minimize storage overhead. Since each possible way of lexicographically ordering the RDF graph (SPO, SOP, PSO, ...) is materialized in a separate index, joins can be answered using efficient merge-only joins during query processing (as opposed to the sort-merge joins that are normally required). We mention that in addition, RDF-3X also builds six so-called *aggregated* indexes and three so-called one-valued indexes, but refer to RDF-3X paper for full details [15].

The RDF-3X query optimizer uses detailed statistics (available, among others, in the aggregated and one-valued indexes) to efficiently generate bushy join orderings and physical query plans using an RDF-tailored cardinality and selectivity estimation algorithm [15].

SAINTDB. SAINT-DB represents structural indexes $I = (N, E)$ by assigning a unique integer $\mathit{id}(n) > 0$ to each index block $n \in N$. Both the partition N of D and D itself are represented by storing all triples $(s, p, o) \in D$ as *quads* of the form $(s, p, o, \mathit{id}([s, p, o]_I))$, where $\mathit{id}([s, p, o]_I)$ denotes the identifier of the index block containing (s, p, o). The set of labeled edges E over N is represented by storing each $(m, \tau, n) \in E$ also as a quad $(\mathit{id}(m), \tau, \mathit{id}(n), 0)$, where the 0 in the fourth column allows us to distinguish quads that represent E-edges from quads representing D-triples. All of these quads are conceptually stored in a single quaternary relation.

Since SAINT-DB hence stores quads instead of triples, we have updated the complete RDF-3X infrastructure (B$^+$-tree storage management and indexes, query optimization and compilation, query processing, data statistics, etc.) to reason about quads instead of triples. This effectively means that we save all permutations of subject (S), predicate (P), object (O), and block-id (B) (as well as their aggregate and one-value versions) into B$^+$-trees. As a consequence, it becomes possible to retrieve the set of all triples that (1) match a given triple pattern and (2) belong to a given index block by accessing the suitable B$^+$-tree. For example, to compute $[\![(\mathit{jane}, \mathit{friendOf}, ?x)]\!]_n$ with n an index block we would search the *SPBO* B$^+$-tree using the prefix key $(\mathit{jane}, \mathit{friendOf}, \mathit{id}(n))$ and find all bindings for $?x$ using a range scan over the corresponding leaves. This idea is easily extended to compute the set of all triples that (1) match a given triple pattern and (2) belong to a *set* of given index blocks. For example, to compute $[\![(\mathit{jane}, \mathit{friendOf}, ?x)]\!]_{n_1 \cup n_2}$ we would search and scan the *SPBO* B$^+$-tree using the prefix key $(\mathit{jane}, \mathit{friendOf}, \mathit{id}(n_1))$; search and scan again using the $(\mathit{jane}, \mathit{friendOf}, \mathit{id}(n_2))$ prefix; and merge the two results to produce a sorted list of bindings for $?x$.

Adding Predicates to the Index. During our experiments we have noticed that the set A of all embeddings of BGP P into I frequently contains embeddings α that cannot contribute to $[\![P]\!]_D$ due to the fact that, for some triple pattern $p \in P$, there is actually no triple in $\alpha(p)$ that mentions the constants required in p. To remedy this deficiency while keeping the index small, we store, for each index block $n \in N$ the set $\mathit{preds}(n)$ of predicates mentioned, $\mathit{preds}(n) := \{\mathit{pred} \mid (s, \mathit{pred}, o) \in n\}$. Since the set of all predicates used in an RDF graph is typically quite small, each $\mathit{preds}(n)$ is also small and efficient to represent. By storing $\mathit{preds}(n)$ in the index we can then remove from A all embeddings α for which there is some triple pattern $(s, p, o) \in P$ with p a constant and $p \notin \mathit{preds}(n)$. Let us denote this reduced set of embeddings by A'.

SAINTDB Query Processing. We have implemented the following three query processing strategies in SAINT-DB. In each of these strategies, we first compute the reduced set A' of embeddings of the P into I, as described above.

The first two strategies corresponds to the methods (M1) and (M2) described in Sec. 3 where embeddings are only taken from A' and where the sets $[\![p_i]\!]_{\alpha(p_i)}$ and $[\![p_i]\!]_{\bigcup_{\alpha \in A'} \alpha(p)}$ are computed using the suitable B$^+$-tree range scans, as outlined above. No join ordering is attempted; all joins are executed in the same order as when RDF-3X computes $[\![p_1]\!]_D \bowtie \cdots \bowtie [\![p_n]\!]_D$. Since the operands of the (M1) and (M2) may be smaller than the corresponding operands of the RDF-3X join, this order may not be optimal.

The third strategy, denoted (M3) in what follows, is a variant of (M2) that employs full quad-based query optimization to reach a suitable physical query plan. In particular, (M3) uses the statistics to estimate the cardinality of both $[\![p]\!]_D$ and $[\![p]\!]_{\bigcup_{\alpha \in A'}}$, for each $p \in P$. In the event that the set $\{\alpha(p) \mid \alpha \in A\}$ contains multiple index blocks (and we hence have to do multiple B$^+$-tree scans and merge the results) it uses these cardinalities to check that the costs for loading and merging $[\![p]\!]_{\bigcup_{\alpha \in A} \alpha(p)}$ is lower than the cost of simply loading $[\![p]\!]_D$. If not, the structural index information is thrown away, and $[\![p]\!]_D$ will be executed (but only for the triple pattern under consideration in isolation). Once it has determined, for each triple pattern, whether the available structural index embeddings should be used, it computes a bushy join ordering and physical plan, based on the quad cardinality statistics.

5 Experimental Validation

5.1 Experimental Setup

We have implemented SAINT-DB upon RDF-3X version 0.36[2]. All experiments described in this section have been run on an Intel Core i7 (quad core, 3.06 GHz, 8MB cache) workstation with 8GB main memory and a three-disk RAID 5 array (750GB, 7200rpm, 32MB cache) running 64-bit Ubuntu Linux.

Our performance indicator is the number of I/O read requests issued by SAINT-DB and RDF-3X, measured by counting the number of calls to the buffer manager's `readPage` function. Thereby, our measurements are independent of the page buffering strategies of the system. Since SAINT-DB currently does not yet feature compression of the B^+-tree leaf pages, we have also turned off leaf compression in RDF-3X for fairness of comparison. During all of our experiments the structural indexes were small enough to load and keep in main memory. The computation of the set of all embeddings into the index hence does not incur any I/O read requests, and is not included in the figures mentioned.

Datasets, Queries, and Indexes. We have tested SAINT-DB on two synthetic datasets and one real-world dataset. The first synthetic dataset, denoted CHAIN, is used to demonstrate the ideal that SAINT-DB can achieve on highly graph-structured and repetitive data. It contains chains of triples of the form $(x_1, y_1, x_2), (x_2, y_2, x_3), \ldots, (x_n, y_n, x_{n+1})$, with chain length n ranging from 3 to 50. Each chain is repeated 1000 times and CHAIN includes around 1 million triples in total. The full simulation index $\textsc{sim}$(CHAIN) has been generated accordingly, and consists of 1316 index blocks, each consisting of 1000 triples. On CHAIN we run queries that also have a similar chain-shaped style $(?x_1, ?y_1, ?x_2), (?x_2, ?y_2, ?x_3), \ldots, (?x_n, ?y_n, ?x_{n+1})$, with n varying from 4 to 7.

The second synthetic dataset, denoted LUBM, is generated by the Lehigh University Benchmark data generator [11] and contains approximately 2 million triples. For this dataset, we computed the depth-2 simulation index $\textsc{sim}_2$(LUBM), which consists of 222 index blocks. Index blocks have varying cardinalities, containing as little as 1 triple to as many 190,000 triples.

The real-world RDF dataset, denoted SOUTHAMPTON, is published by the University of Southampton[3]. It contains approximately 4 million triples. For this dataset, we also computed the depth-2 simulation index $\textsc{sim}_2$(SOUTHAMPTON), which consists of 380 index blocks. Index blocks have varying cardinalities, containing as little as 1 triple to as many 10^6 triples.

For all datasets the indexes in their current non-specialized form require only a few megabytes and therefore can be kept in main memory. A specialized in-memory representation could easily further reduce this footprint. The detailed description of the queries used can be found online[4]. In the rest of this section we denote queries related to the LUBM dataset as *L1*, ..., *L16*, and those related to SOUTHAMPTON as *S1*, ..., *S7*.

[2] `http://code.google.com/p/rdf3x/`

[3] `http://data.southampton.ac.uk/`

[4] `http://www.win.tue.nl/~yluo/saintdb/`

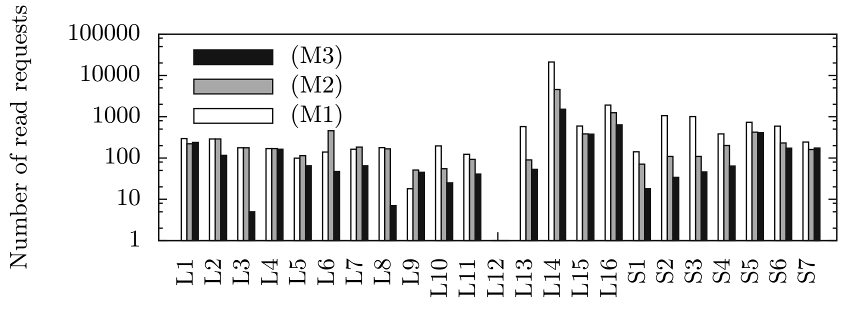

Fig. 3. Number of read requests for different query processing strategies

5.2 Experimental Analysis

We first examine the different query processing strategies implemented in SAINT-DB to exploit the structural index, and then compare the performance of SAINT-DB to that of RDF-3X. In the following, we assume that the database has loaded the partition blocks into main memory, and ensure that all queries are executed on a cold cache/buffer. The embeddings of the queries into the structural index can therefore be efficiently computed, and are available to the query optimizer.

Query Processing Strategies. Fig. 3 shows the number of I/O read requests issued to the buffer manager during query evaluation by each of the three different query processing strategies (M1), (M2), and (M3) introduced in Sec. 4.

As a general observation, (M1) requires more reads from the database than (M2), which in turn requires more reads than (M3). Conceptually, (M1) executes a different query for each embedding of the original BGP into the structural index, while (M2) executes the same queries in parallel, sharing evaluation costs. (M3) differs from both (M1) and (M2) by exploiting not only the structural index but also the selectivity of particular triple patterns. A lower number of reads can therefore be achieved, by accessing directly the relevant information when very specific triple patterns are issued. For example, *L4* asks for every undergraduate student with a single triple pattern. Hence, no structural information is available while the triple pattern itself selects the right data.

While this observation explains the general behavior, a more detailed analysis provides more insights. First, query *L12* does not require any read. Indeed, this query does not produce any result, and the absence of results is identified at the index level: no embedding of the BGP of this query exist into the structural index. Second, queries *L6* and *L9* show that strategy (M1) can perform better than (M2) and (M3). This is due to different join orderings, which, along with sideways information passing [14], causes scans to skip different data sets. The current plan generation cannot identify these differences, as sideways information passing depends on runtime data.

SAINT-DB vs RDF-3X. We now turn to a comparison of the query evaluation costs of SAINT-DB and RDF-3X. For this comparison we use the (M3) strategy in SAINT-DB, given our observations above on the performance of this strategy.

Table 1. Read requests for SAINT-DB and RDF-3X on the CHAIN dataset. The columns denote the length of the chain in the query. Speed-up is the ratio of read requests of RDF-3X over those of SAINT-DB.

	4	5	6	7
SAINT-DB	306	350	393	438
RDF-3X	3864	4799	5734	6669
Speed-up	12.63	13.71	14.59	15.23

Table 2. Read requests for SAINT-DB and RDF-3X on the LUBM and SOUTHAMPTON datasets. Speed-up is the ratio of read requests of RDF-3X over those of SAINT-DB.

	C1			**C2**				**C3**				
	L2	*L3*	*L4*	*L9*	*S1*	*S2*	*S4*	*L1*	*L5*	*L6*	*L7*	*L8*
SAINT-DB	116	5	163	18	18	36	64	238	39	47	38	7
RDF-3X	89	5	123	12	16	35	53	194	132	39	268	7
Speed-up	0.77	1.00	0.75	0.67	0.89	0.97	0.83	0.82	3.38	0.83	7.05	1.00
	C3											
	L10	*L11*	*L12*	*L13*	*L14*	*L15*	*L16*	*S3*	*S5*	*S6*	*S7*	
SAINT-DB	25	41	0	53	1519	352	288	48	410	173	175	
RDF-3X	21	30	281	109	2668	2178	1224	33	424	316	236	
Speed-up	0.84	0.73	∞	2.06	1.76	6.19	4.25	0.69	1.03	1.83	1.35	

Table 1 shows the performance of RDF-3X and SAINT-DB on the CHAIN dataset. We see that SAINT-DB requires over 10 times less I/Os up compared to RDF-3X, and this reduction in I/Os increases as query length increases. This is due to the rich structures inside the data set and the queries. The structural index can hence significantly eliminate the search space of the later index scans. These results demonstrate the tremendous potential of structural indexing over value-based indexing.

Table 2 shows the I/O costs of RDF-3X and SAINT-DB, for the LUBM and SOUTHAMPTON datasets. We have grouped queries into three categories:

– (C1) Queries without structure. These queries consist of a single triple pattern, and hence do not exhibit any structural information to be exploited.
– (C2) Structured queries over highly specific information. These queries have many triple patterns, and at least one triple pattern is very selective.
– (C3) Structured queries. These queries have many triple patterns, with rich structural information.

By leveraging exhaustive value-based indexes and various optimization strategies, RDF-3X can efficiently answer queries in category C1 and C2. In particular, for C2 the technique of *sideways information passing* [14] allows efficient computation of bindings in RDF-3X for the less selective patterns. Hence, as we can expect, the structural indexes of SAINT-DB provide no advantage. Nevertheless, even though these queries represent the worst-case scenario for structural indexes, SAINT-DB generally exhibits comparable query evaluation costs.

Further study is nevertheless warranted to bridge the gap between value-based and structure-based indexing for such query types (e.g., compression techniques in index blocks, to offset the overhead introduced by moving from triples to quads).

For the queries of category C3, we see that structural information does increase selectivity significantly. For example, queries *L5*, *L7*, *L14*, *L15*, and *L16* do benefit from the increased selectivity, with SAINT-DB having as little as 15% of the query evaluation costs of RDF-3X. The query benefiting the most from structural information is *L12*. The result is detected as empty at the structural index level in SAINT-DB, and hence no read request is issued. For this same query, RDF-3X needs to perform a number of joins to find the same empty result. The structural-approach avoids these I/Os completely.

This initial empirical study indicates that there are indeed general situations where SAINT-DB can clearly leverage structural information for significant reduction in query evaluation costs. Furthermore, for queries without significant structure, or with highly selective triple patterns, SAINT-DB is competitive with RDF-3X. Our next steps in this study are a finer analysis of query categories, and their appropriate indexing and query evaluation strategies in SAINT-DB.

6 Concluding Remarks

In this paper, we have presented the first results towards triple-based structural indexing for RDF graphs. Our approach is grounded in a formal coupling between practical fragments of SPARQL and structural characterizations of their expressive power. An initial empirical validation of the approach shows that it is possible and profitable to augment current value-based indexing solutions with structural indexes for efficient RDF data management.

In this first phase of the SAINT-DB investigations, we have focused primarily on the formal framework and design principles. We are currently shifting our focus to a deeper investigation into the engineering principles and infrastructure necessary to put our framework into practice. Some basic issues for further study in this direction include: alternates to the B^+-tree data structure for physical storage and access of indexes and data sets; more sophisticated optimization and query processing solutions for reasoning over both the index and data graphs; efficient external memory computation and maintenance of indexes; and, extensions to richer fragments of SPARQL, e.g., with the OPTIONAL and UNION constructs.

Acknowledgments. We thank Paul De Bra for his critical feedback on these investigations. The research of FP is supported by a FNRS/FRIA scholarship. The research of SV is supported by the OSCB project funded by the Brussels Capital Region. The research of GF, JH, and YL is supported by the Netherlands Organization for Scientific Research (NWO).

References

1. Abadi, D., et al.: SW-Store: a vertically partitioned DBMS for semantic web data management. VLDB J. 18, 385–406 (2009)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Arias, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: USEWOD (2011)
4. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern XML databases. WWW 11(1), 117–151 (2008)
5. Brenes Barahona, S.: Structural summaries for efficient XML query processing. PhD thesis, Indiana University (2011)
6. Bröcheler, M., Pugliese, A., Subrahmanian, V.S.: DOGMA: A Disk-Oriented Graph Matching Algorithm for RDF Databases. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 97–113. Springer, Heidelberg (2009)
7. Fletcher, G.H.L., Beck, P.W.: Scalable indexing of RDF graphs for efficient join processing. In: CIKM, Hong Kong, pp. 1513–1516 (2009)
8. Fletcher, G.H.L., Hidders, J., Vansummeren, S., Luo, Y., Picalausa, F., De Bra, P.: On guarded simulations and acyclic first-order languages. In: DBPL, Seattle (2011)
9. Fletcher, G.H.L., Van Gucht, D., Wu, Y., Gyssens, M., Brenes, S., Paredaens, J.: A methodology for coupling fragments of XPath with structural indexes for XML documents. Information Systems 34(7), 657–670 (2009)
10. Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation: Coarsest partition problems. J. Autom. Reasoning 31(1), 73–103 (2003)
11. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. Web Sem. 3(2-3), 158 (2005)
12. Luo, Y., Picalausa, F., Fletcher, G.H.L., Hidders, J., Vansummeren, S.: Storing and indexing massive rdf datasets. In: De Virgilio, R., et al. (eds.) Semantic Search over the Web, Data-Centric Systems and Applications, pp. 29–58. Springer, Heidelberg (2012)
13. Milo, T., Suciu, D.: Index Structures for Path Expressions. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 277–295. Springer, Heidelberg (1998)
14. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: SIGMOD, pp. 627–640 (2009)
15. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
16. Picalausa, F., Vansummeren, S.: What are real SPARQL queries like? In: Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, pp. 7:1–7:6. ACM, New York (2011)
17. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, W3C Recommendation (2008)
18. Sidirourgos, L., et al.: Column-store support for RDF data management: not all swans are white. Proc. VLDB Endow. 1(2), 1553–1563 (2008)
19. Tran, T., Ladwig, G.: Structure index for RDF data. In: Workshop on Semantic Data Management, SemData@ VLDB (2010)
20. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A graph based RDF index. In: AAAI, Vancouver, B.C., pp. 1465–1470 (2007)

21. van Glabbeek, R.J., Ploeger, B.: Correcting a Space-Efficient Simulation Algorithm. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 517–529. Springer, Heidelberg (2008)
22. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: VLDB, Auckland, New Zealand (2008)
23. Wylot, M., Pont, J., Wisniewski, M., Cudré-Mauroux, P.: dipLODocus[RDF]—Short and Long-Tail RDF Analytics for Massive Webs of Data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 778–793. Springer, Heidelberg (2011)
24. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: Answering SPARQL queries via subgraph matching. Proc. VLDB Endow. 4(8), 482–493 (2011)