Scalable Indexing of RDF Graphs for Efficient Join Processing

George H. L. Fletcher
Department of Mathematics & Computer Science
Eindhoven University of Technology, The Netherlands
g.h.l.fletcher@tue.nl

Peter W. Beck
School of Engineering & Computer Science
Washington State University, Vancouver, USA
pwbeck@wsu.edu

ABSTRACT

Current approaches to RDF graph indexing suffer from weak data locality, i.e., information regarding a piece of data appears in multiple locations, spanning multiple data structures. Weak data locality negatively impacts storage and query processing costs. Towards stronger data locality, we propose a Three-way Triple Tree (TripleT) secondary memory indexing technique to facilitate flexible and efficient join evaluation on RDF data. The novelty of TripleT is that the index is built over the atoms occurring in the data set, rather than at a coarser granularity, such as whole triples occurring in the data set; and, the atoms are indexed regardless of the roles (i.e., subjects, predicates, or objects) they play in the triples of the data set. We show through extensive empirical evaluation that TripleT exhibits multiple orders of magnitude improvement over the state-of-the-art, in terms of both storage and query processing costs.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Query processing

General Terms

Performance, Experimentation

1. INTRODUCTION

RDF [9] is becoming the data model of choice in many emerging data generation and sharing scenarios. RDF data sets (often referred to as "graphs" of "triples") are typically generated in social semantic domains where often a fixed schema is not available a priori. The natural flexibility and expressivity of triples was recognized early in the development of modern logic [11]. Triples, which treat both objects and relationships as first-class citizens, blur the traditional divide between data and metadata, allowing for freer on-the-fly generation of data. This is in contrast to previous graph-oriented data models, which have more rigidly maintained the distinction between data and metadata [2].

CIKM'09, November 2–6, 2009, Hong Kong, China.

The format and flexibility of RDF call for new query paradigms and processing techniques [6]. At the heart of many RDF query languages, including the W3C standard language SPARQL [12], are *basic graph patterns* [7]. Recent research has begun to make headway on understanding fundamental theoretical (e.g., [7]) and engineering (e.g., [10, 14]) aspects of basic graph pattern query processing.

In this paper, we focus on the problem of scalable indexing of RDF data, to support efficient processing of basic graph patterns. As we discuss below, current indexing techniques suffer from weak data locality, in the sense that information about a piece of data can appear in multiple locations, possibly spanning several different data structures. Weak data locality negatively impacts storage and query processing costs. Towards stronger data locality, we propose a *Three-way Triple Tree* (TripleT) secondary memory index. The novelty of TripleT is that, in contrast to state of the art approaches, (1) the index is built over the atoms occurring in the graph, rather than at a coarser granularity, such as whole triples occurring in the graph; and (2) the atoms are indexed regardless of the roles they play in the triples of the graph. TripleT is robust in the face of the dynamic unstructured nature of typical RDF graphs. Furthermore, the proposal is conceptually quite simple. We show through extensive empirical evaluation that TripleT exhibits multiple orders of magnitude improvement over the state of the art, in terms of both storage and query processing costs.

2. BACKGROUND

We begin with a brief presentation of basic notation and definitions used in the paper, a statement of the indexing problem considered, and an overview of the state of the art.

Data model. Full details of the RDF data model can be found in the W3C standards [9, 12]. For our purposes here, let $\mathcal{A}$ be an enumerable set of atoms (e.g., Unicode strings). A *triple* is an element $(s, p, o) \in \mathcal{A} \times \mathcal{A} \times \mathcal{A}$, typically interpreted as a statement to the effect that "object o stands in relationship p to subject s." Hence, the first element s is called the *subject* of the triple; p is called the *predicate*; and o is called the *object*. An RDF *graph* is a finite set of triples. In a graph G, let $\mathcal{S}(G)$ denote the set of atoms appearing as subjects; $\mathcal{P}(G)$ the set of atoms appearing as predicates; $\mathcal{O}(G)$ the set of atoms appearing as objects; and $\mathcal{A}(G) = \mathcal{S}(G) \cup \mathcal{P}(G) \cup \mathcal{O}(G)$. For example, in graph G of Figure 1, we have $\mathcal{S}(G) = \{\texttt{Yamada}, \texttt{Herzog}, \texttt{McShea}, \texttt{doc1}, \texttt{doc2}, \texttt{doc3}, \texttt{knows}\}$.

Basic graph patterns. Let $\mathcal{V}$ be an enumerable set of variables, disjoint from $\mathcal{A}$. A *simple access pattern* (SAP) is an element of $(\mathcal{A} \cup \mathcal{V}) \times (\mathcal{A} \cup \mathcal{V}) \times (\mathcal{A} \cup \mathcal{V})$. In other

$\{\langle$`Yamada, authored, doc1`$\rangle$,
$\langle$`Yamada, knows, McShea`$\rangle$,
$\langle$`knows, is a kind of, social action`$\rangle$,
$\langle$`Herzog, authored, doc2`$\rangle$,
$\langle$`Herzog, authored, doc3`$\rangle$,
$\langle$`McShea, performed, doc3`$\rangle$,
$\langle$`McShea, past action, authored`$\rangle$,
$\langle$`doc1, type, PDF`$\rangle$,
$\langle$`doc1, rating, 4/5`$\rangle$,
$\langle$`doc2, type, MP3`$\rangle$,
$\langle$`doc3, type, MP3`$\rangle$,
$\langle$`doc3, createdOn, 29.6.09`$\rangle\}$

Figure 1: A small triple graph.

words, an SAP is a triple in which roles (i.e., subjects, predicates, and objects) may be either atoms or variables. A *basic graph pattern* (BGP) is a conjunction of one or more SAPs: $(s_1, p_1, o_1) \wedge \cdots \wedge (s_n, p_n, o_n)$. Let $\mathcal{V}(P)$ denote the set of variables occurring in BGP P, and consider the set $M(P)$ of functions f which map $\mathcal{A} \cup \mathcal{V}(P)$ into $\mathcal{A}$, such that f is the identity on $\mathcal{A}$:

$$M(P) = \{f \mid f : (\mathcal{A} \cup \mathcal{V}(P)) \rightarrow \mathcal{A} \text{ and } f|_{\mathcal{A}} = Id_{\mathcal{A}}\}.$$

Then, the *semantics* of BGP P on a graph G is the set

$$P(G) = \{f \in M(P) \mid \forall (a, b, c) \in P : (f(a), f(b), f(c)) \in G\}$$

where $(a, b, c) \in P$ means that (a, b, c) is an SAP occurring in P. In other words, $P(G)$ is the set of bindings for the variables of P, such that each binding maps all of the SAPs of P into triples of G.

EXAMPLE 1. *Consider the query "What are the dates and types of documents on which McShea was a performer?" over the graph G given in Figure 1, as a BGP:*

$$P = (\texttt{McShea}, \texttt{performed}, ?doc) \wedge (?doc, \texttt{createdOn}, ?date) \wedge (?doc, \texttt{type}, ?type).$$

On G, we have only one valid binding, $P(G) = \{\langle ?doc : \texttt{doc3}, ?date : \texttt{29.6.09}, ?type : \texttt{MP3}\rangle\}$.

BGPs are essentially conjunctive queries tailored for the RDF data model [7]. Joins between the SAPs of a BGP are induced by the co-occurrence of variables and atoms. There are six native BGP join types: subject-subject, subject-predicate, subject-object, predicate-predicate, predicate-object, and object-object joins. In Example 1, there is a subject-object join between the SAP $(\texttt{McShea}, \texttt{performed}, ?doc)$ and both of the other SAPs, due to the co-occurrence of variable $?doc$. Likewise, there is a subject-subject join between $(?doc, \texttt{createdOn}, ?date)$ and $(?doc, \texttt{type}, ?type)$.

The problem. How can we index a graph G to support efficient evaluation of BGPs on G? We specifically focus on the design of *native* RDF index data structures, i.e., indexes which support the full range of BGP join patterns.

State of the art solutions. In what follows, we use the B+tree secondary-memory data structure [3] to implement the various indexing techniques considered. We assume familiarity with this data structure and its use in conjunctive query processing. To the best of our knowledge, the two major competitive proposals for native RDF indexing are multiple access patterns (MAP) and HexTree.

MAP. In this approach, all three positions of triples are indexed: subjects (S), predicates (P), and objects (O), for some permutation of S, P, and O. MAP requires up to six separate B+trees, corresponding to the six possible orderings of roles: SPO, SOP, PSO, POS, OSP, OPS. For example, for each $(s, p, o) \in G$, it is the case that $o\#p\#s$ is a key in the OPS index on G (where "#" is some reserved separator symbol). A BGP join evaluation requires two or more look-ups, potentially in different trees, followed by sort merge joins. Major systems employing the MAP technique include Virtuoso, YARS, and RDF-3X [5, 8, 10].

HexTree. Recently in the Hexastore system, Weiss et al. [15] have proposed indexing two roles at a time. This approach requires up to six separate data structures corresponding to the six possible orderings of roles: SO, OS, SP, PS, OP, PO. Payloads are shared between indexes with symmetric orderings. For example, for each $(s, p, o) \in G$, it is the case that $s\#p$ is a key in the SP index on G, $p\#s$ is a key in the PS index on G, and both of these keys point to a shared payload of $\{o \in \mathcal{O}(G) \mid (s, p, o) \in G\}$. As with MAP, join evaluation requires two or more look-ups, potentially in different trees, followed by sort merge joins. Hexastore has only been proposed and evaluated as a main-memory data structure [15]. We propose *HexTree* as an effective secondary-memory realization of the Hexastore proposal using the B+tree data structure.

Limitations of current solutions. In both MAP and HexTree, information about a piece of data can appear in multiple locations, possibly spanning several different data structures. For example, consider the atom `doc1` in the graph of Figure 1. In the MAP indexing scheme, locating all triples related to `doc1` requires lookups in three different data structures: the SPO B+tree (or the SOP B+tree) to locate those triples in which the atom occurs as a subject (e.g., $\langle \texttt{doc1}, \texttt{type}, \texttt{PDF}\rangle$), the PSO (or POS) B+tree to determine that the atom does not occur as a predicate, and the OSP (or OPS) B+tree to locate those triples in which the atom occurs as an object (e.g., $\langle \texttt{Yamada}, \texttt{authored}, \texttt{doc1}\rangle$). Similarly, reconstructing `doc1` in the HexTree indexing scheme also requires three separate lookups.

This loss of *data locality* is a primary limitation of the MAP and HexTree indexing schemes. Weak data locality leads to (1) redundant storage (e.g., each B+tree in the MAP scheme contains a separate copy of essentially the same set of data), and (2) increased query processing costs (e.g., performing a join on an atom can require multiple independent index look-ups). We next present an indexing scheme designed with an eye towards strengthened data locality.

3. A THREE-WAY TRIPLE TREE INDEXING SCHEME

Towards stronger data locality, we propose indexing the key-space $\mathcal{A}(G)$, regardless of the particular roles the atoms of $\mathcal{A}(G)$ play in the triples of G. For a key k, the payload is all triples of G in which atom k occurs. In particular, the payload for k consists of three "buckets": one for all pairs (p, o) where $(k, p, o) \in G$, one for all pairs (s, o) where $(s, k, o) \in G$, and one for all pairs (s, p) where $(s, p, k) \in G$. In other words, there is one bucket apiece for all those triples where k occurs as a subject, for all those triples where k occurs as a predicate, and for all those triples where k appears as an object. For example, on the graph of Figure 1, the payload for `doc1` would consist of an object bucket $\langle(\texttt{Yamada}, \texttt{authored})\rangle$, a subject bucket $\langle(\texttt{4/5}, \texttt{rating}), (\texttt{PDF}, \texttt{type})\rangle$, and a predicate bucket $\langle\rangle$.

To facilitate query processing (e.g., for merge joins), we keep the pairs in each of the buckets sorted. By default, the subject bucket is sorted in OP order, the predicate bucket in SO order, and the object bucket in SP order. These choices are based on the assumption that subjects are more selective than objects, and objects are more selective than predicates. Of course, selectivities will be dependent on the characteristics of G, and sort orders can be chosen as necessary (or, even both sort orders can be materialized if warranted).

We note two features of TripleT. First, the keys in TripleT are 1/3 the length of those in MAP and 1/2 those in HexTree. Consequently, there is a higher branching factor in the TripleT B+tree. Furthermore, the key-space indexed by TripleT, $\mathcal{A}(G)$, itself is potentially much smaller that those indexed in the MAP and HexTree schemes. Taken together, these reductions lead to shallower indexes and hence lower lookup costs, relative to MAP and HexTree.

Second, TripleT requires just one index, leading to stronger data locality and non-trivial reduction in storage costs relative to MAP and HexTree, while efficiently supporting all BGP join types. For example, a subject-object join induced by the co-occurrence of an atom k can be evaluated by a single look-up on k followed by a merge-join between the subject and object buckets of k's payload. A join induced by the co-occurrence of a variable is implemented as multiple look-ups followed by sort merge joins, just as with MAP and HexTree.

EXAMPLE 2. *Consider the BGP* $P_1 = (\texttt{Herzog}, \texttt{authored}, ?d_1) \wedge (\texttt{Herzog}, \texttt{performed}, ?d_2)$. *To compute* $P_1(G)$ *using TripleT requires a single lookup of* `Herzog` *followed by a lookup in the resulting payload's subject bucket for* `authored`$\#?d_1$ *and* `performed`$\#?d_2$. *The crossproduct of all matches is then formed.*

Next, consider the BGP $P_2 = (\texttt{McShea}, \texttt{performed}, ?d_1) \wedge (?d_1, \texttt{createdOn}, ?d_2)$. *To compute* $P_2(G)$ *with TripleT requires: (1) a lookup of* `performed`*; followed by (2) a lookup of* `McShea`$\#?d_1$ *in the predicate bucket of the resulting payload; (3) a lookup of* `createdOn`*; followed by (4) the retrieval of the predicate bucket of the resulting payload; and finally (5) a direct merge join of the results of (2) and (4) on* $?d_1$.

In general, we have the TripleT join algorithm for two SAPs S_1 and S_2 presented in Algorithm 3.1. Note that we (1) abuse our notation, and let $\mathcal{A}(S)$ denote the set of atoms occurring in SAP S and, (2) set aside the degenerate case which occurs when either of the SAPs is atom-free, which necessitates an (index) nested loops join.

TripleT exhibits the same advantages as MAP and HexTree, as compared to earlier RDF storage and indexing proposals, e.g., robustness in the face of dynamic data (e.g., schema independence); concise handling of multivalued resources; avoidance of nulls; and, all pairwise joins are fast merge-joins (see the detailed discussion of these advantages in [1, 15]). Furthermore, the additional benefits of simplicity, reduced size, and strengthened data locality distinguish the TripleT proposal from the state of the art.

Input: • a basic graph pattern $P = S_1 \wedge S_2$
• I_G, the TripleT index on graph G
Output: $P(G)$, the evaluation of P on G
begin
  if $\mathcal{A}(S_1) \cap \mathcal{A}(S_2) \neq \emptyset$ **then**
    • choose an arbitrary $\texttt{a} \in \mathcal{A}(S_1) \cap \mathcal{A}(S_2)$
    • lookup `a` in I_G
    • compute $S_1(G) \bowtie S_2(G)$ on retrieved payload using sort merge, and return result
  else
    • choose arbitrary $\texttt{a}_1 \in \mathcal{A}(S_1)$ and $\texttt{a}_2 \in \mathcal{A}(S_2)$
    • lookup $\texttt{a}_1$ and $\texttt{a}_2$ in I_G
    • evaluate $S_1(G)$ and $S_2(G)$ on retrieved payloads, respectively
    • compute $S_1(G) \bowtie S_2(G)$ using sort merge, and return result
end

Algorithm 3.1: TripleT join algorithm for SAPs S_1, S_2

4. EMPIRICAL STUDY

We implemented MAP, HexTree, and TripleT using 8K blocks and 32-bit references, in virtual memory, using Python 2.5.2. All experiments were executed on a pair of 2.66 GHz dual-core Intel Xeon processors with 16 GB RAM running Mac OS X 10.4.11.

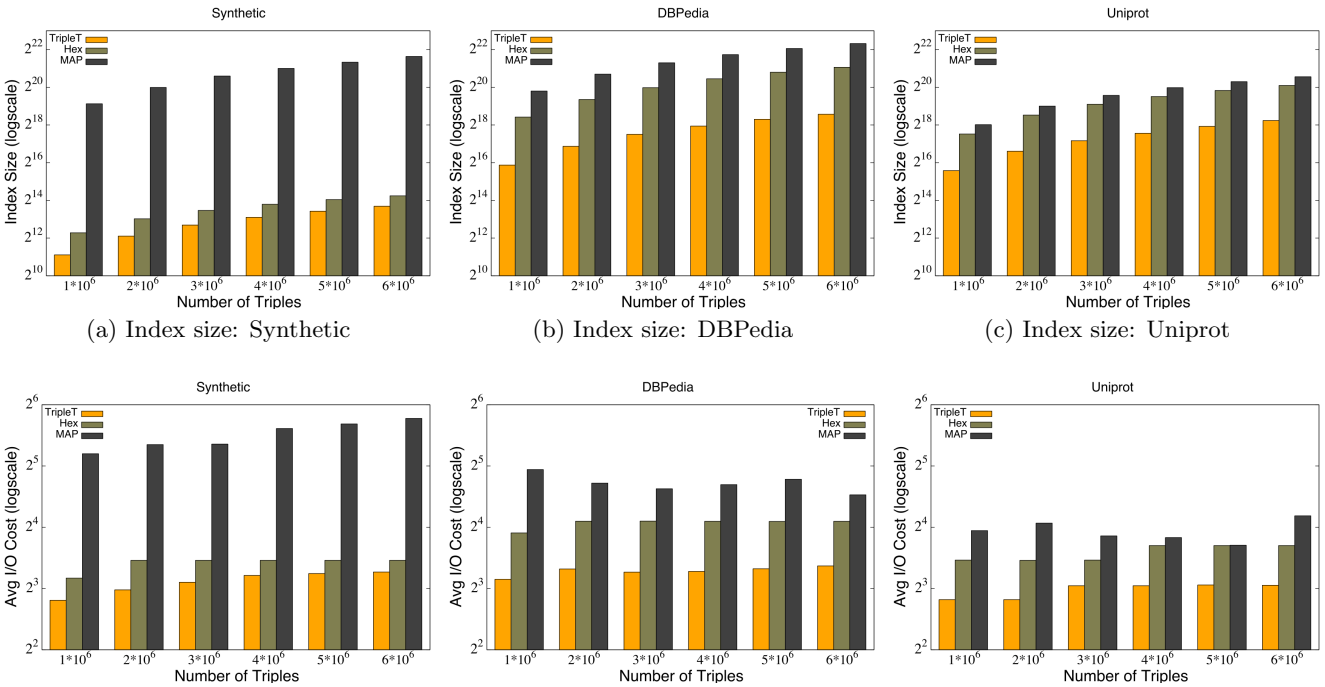
Each experiment was performed using (1) synthetic data, (2) the DBPedia data set;[1] and (3) the Uniprot data set.[2] DBPedia is an extraction of the well-known Wikipedia online encyclopedia. Uniprot is a comprehensive collection of protein sequence and annotation data. For (1), we built two synthetic data sets (the results presented below are the averages over these two sets). In the first set, we randomly generated n triples over $n^{1/3}$ unique atoms, for $n = 1,000,000$, to $n = 6,000,000$, in increments of one million, where repetitions of atoms were allowed within triples. In the second set, we randomly generated n triples over $ceiling(n^{1/3}) + 2$ unique atoms, for $n = 1,000,000$, to $n = 6,000,000$, in increments of one million, where repetitions of atoms within triples were disallowed. For (2) and (3), we took an arbitrary sample of 10,000,000 triples from each data collection. After cleaning, we kept 6,000,000 triples in each collection.

Comparing index size. In increments of 1 million triples, from 1 to 6 million triples, we built the three index types. The plots of the resulting index sizes, in 8K blocks, are shown in Figures 2(a)-2(c).

TripleT was up to 259 times smaller, with a typical two orders of magnitude savings in storage cost. This cost reduction is due to (1) TripleT uses just one B+tree, whereas MAP and HexTree both require three B+trees, and (2) the key size in TripleT is 1/3 that of MAP and 1/2 that of HexTree, leading to significantly higher branching factor of the B+tree (and hence shallower trees).

Comparing query performance. We measured the I/O cost for evaluating BGP join patterns between two SAPs. Since a join is induced by virtue of the SAPs sharing either an atom, a variable, or both, we considered four subscenarios, covering the basic ways in which SAPs may be joined: (1) computing the join of two variable-free SAPs having one atom in common, e.g., $(\texttt{a}, \texttt{b}, \texttt{c}) \wedge (\texttt{a}, \texttt{d}, \texttt{e})$; (2) computing the join of two SAPs having one atom in common, one SAP having a single variable and the other variable-free, e.g., $(\texttt{a}, \texttt{b}, ?v) \wedge (\texttt{a}, \texttt{c}, \texttt{d})$; (3) computing the join of two SAPs having no atoms in common, each having a single variable, which they share, e.g., $(\texttt{a}, \texttt{b}, ?v) \wedge (?v, \texttt{c}, \texttt{d})$; (4) computing the join of two SAPs having one atom in com-

[1] `http://wiki.dbpedia.org`
[2] `http://dev.isb-sib.ch/projects/uniprot-rdf`



(a) Index size: Synthetic (b) Index size: DBPedia (c) Index size: Uniprot

(d) Query evaluation: Synthetic (e) Query evaluation: DBPedia (f) Query evaluation: Uniprot

Figure 2: Index size in 8K blocks (top), and average I/O cost of query evaluation (bottom).

mon, each having one variable, which they also share, e.g., $(\mathtt{a}, \mathtt{b}, ?v) \wedge (\mathtt{a}, \mathtt{c}, ?v)$. These scenarios cover the whole range of basic BGP join types, and were chosen to give the indexes a complete work out.

For each data set, for each size (1-6 million), we generated ten random BGPs of each of these four scenarios, and measured their evaluation cost using MAP, HexTree, and TripleT. The average I/O costs are given in Figures 2(d)-2(f). We observe from these experiments that TripleT always out-performed MAP and HexTree, with down to only 17.6% of their I/O costs for query processing. This performance improvement is due to (1) the smaller keyspace and size of TripleT, and (2) reduced lookup costs for some basic types of BGPs, due to stronger data locality.

5. DISCUSSION

As demonstrated on both synthetic and real data sets, TripleT is multiple orders of magnitude smaller than MAP and HexTree, and exhibits significantly reduced I/O costs for join processing across the full range of RDF join patterns. Both of these improvements are due in large part to the improved data locality of the TripleT indexing scheme, as discussed above.

We have focused in this work on scalable indexing, specifically to support efficient join processing. Based on the results of this study, there are several complementary directions for further research on TripleT: (1) Recently a full-scale benchmark suite for RDF data management research has been developed [13]. Further analysis of TripleT on such benchmarks may suggest useful refinements or extensions of the data structure. (2) As a role-free approach to indexing the relationships of data in an RDF graph, it might be profitable to store additional information in the TripleT payload structure. For example, we are currently investigating payload structures to efficiently support keyword and path queries [4] with TripleT. (3) Finally, we are currently investigating heuristics, statistics, and algorithms for BGP join-order optimization (e.g., [10, 14]), especially tailored for TripleT.

Acknowledgments. We thank Sriram Mohan, Michael Schmidt, and Melanie Wu for their many helpful comments.

6. REFERENCES

[1] D. J. Abadi *et al.* SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *VLDB J*, 2009.
[2] R. Angles and C. Gutiérrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1–39, 2008.
[3] D. Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.
[4] X. Dong and A. Y. Halevy. Indexing Dataspaces. In *ACM SIGMOD*, pages 43–54, Beijing, 2007.
[5] O. Erling. Towards Web Scale RDF. In *SSWS*, Karlsruhe, Germany, 2008.
[6] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Reasoning Web*, pages 1–52, Lisbon, 2006.
[7] C. Gutiérrez *et al.* Foundations of Semantic Web Databases. In *ACM PODS*, pages 95–106, Paris, 2004.
[8] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC*, Busan, Korea, 2007.
[9] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Rec., 2004.
[10] T. Neumann and G. Weikum. RDF-3X: A RISC-Style Engine for RDF. In *VLDB*, Auckland, New Zealand, 2008.
[11] C. S. Peirce. Description of a Notation for the Logic of Relatives. *Memoirs of the American Academy of Arts and Sciences*, 9:317–378, 1870.
[12] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.
[13] M. Schmidt *et al.* SP^2Bench: A SPARQL Performance Benchmark. In *IEEE ICDE*, Shanghai, 2009.
[14] M. Stocker *et al.* SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *ACM WWW*, Beijing, 2008.
[15] C. Weiss *et al.* Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, 2008.