RDFBroker: A Signature-Based High-Performance RDF Store

Michael Sintek and Malte Kiesel

DFKI GmbH, Kaiserslautern, Germany
{sintek, kiesel}@dfki.uni-kl.de
http://www.dfki.uni-kl.de/~{sintek, kiesel}

Abstract. Many approaches for RDF stores exist, most of them using very straight-forward techniques to store triples in or mapping RDF Schema classes to database tables. In this paper we propose an RDF store that uses a natural mapping of RDF resources to database tables that does not rely on RDF Schema, but constructs a schema based on the occurring signatures, where a signature is the set of properties used on a resource. This technique can therefore be used for arbitrary RDF data, i.e., RDF Schema or any other schema/ontology language on top of RDF is not required. Our approach can be used for both in-memory and on-disk relational database-based RDF store implementations.

A first prototype has been implemented and already shows a significant performance increase compared to other freely available (in-memory) RDF stores.

1 Introduction

RDF has been developed to facilitate semantic (meta-)data exchange between actors on the (Semantic) Web [1]. Its primary design rationale was simplicity; therefore, a lowest common denominator of knowledge representation formalisms suited for this task has been chosen: triples, or statements of the form *subject, predicate, object*. But exactly for the same reason, being a lowest common denominator of knowledge representation formalisms, it is suited neither for internal use in general applications[1] nor for efficient handling in established databases which are optimized to handle tables or n-tuples, not ternary statements/binary predicates. Also, naive handling of triples leads to inefficient memory usage since data is implicitly duplicated.[2]

We therefore propose (and implemented a first prototype of) an RDF store that on the one hand allows *efficient import and export of RDF data*, but on the other hand allows *adequate and efficient access from applications* (i.e., from the applications that serve as actors on the Semantic Web, e.g., web services). Unlike

[1] Applications' data structures are typically object-oriented and not statement-oriented.

[2] A resource with four properties needs four statements, each having the resource's URI as object.

Y. Sure and J. Domingue (Eds.): ESWC 2006, LNCS 4011, pp. 363–377, 2006.

other similar approaches which restrict themselves to supporting exactly one (application-adequate) schema formulated in a schema language such as RDFS or OWL, we believe that we should stay *schema independent*, for two simple reasons: there is no "one-size-fits-all" schema/ontology language, and because of the distributed and chaotic nature of the Semantic Web, an application might not have full access to the schema, the schema might be incomplete or even non-existent, or the data is, at least at intermediate stages, simply not schema-compliant.

Our approach is based on the notion of *signatures*, where a signature of a resource is the set of properties used on that very resource (at a specific point of time in an application). This approach allows Semantic Web data to be represented as normal (database) relations (with signatures being the database column headings), which are much more application-adequate and, at the same time, much more efficient wrt. space and time than naive approaches that directly store triples. Especially queries that access multiple properties of a resource simultaneously (which, in our experience, form the vast majority of queries) benefit from this approach.

We furthermore *benefit from database technology* that has been optimized for performing queries on normal database tables, i.e., tables that group structurally similar objects, which in our case are resources with the same properties. Standard database technology is not very efficient (esp. wrt. queries) when you simply map triples to one large table (plus some tables for compressing namespaces etc.), since then data usually accessed together is arbitrarily distributed over the database, resulting in many (non-consecutive) parts ("pages") from hard disk being accessed for one query.

Apart from the obvious benefits of our approach when using on-disk databases, the approach also shows considerable *performance improvements* when storing RDF data in memory, esp. for queries accessing multiple properties for a single resource.

A first (in-memory) prototype, the *RDFBroker*, has been realized as part of the OpenDFKI open-source initiative.[3]

In the following, we will first explain the major concepts of RDFBroker (section 2), give some details on the implementation (section 3), show first results of evaluating our approach (section 4), list some related work (section 5), and finally conclude the paper and describe plans for future work (section 6).

2 RDFBroker Concepts

RDFBroker mainly relies on the concept of signatures and signature tables which are organized in a lattice-like structure. On these tables, normal operations known from relational algebra are applied. In the following, we will formally define these basic concepts.

[3] `http://rdfbroker.opendfki.de/`

2.1 Signatures

Definition 1. *The* signature $\Sigma_G(s)$ *of a resource s wrt. an RDF graph*[4] *G is the set of properties that are used on s in G:*

$$\Sigma_G(s) = \{p \mid \exists o : \langle s, p, o\rangle \in G\}$$

When it is understood from the context (or irrelevant) which graph is being referred to, we just write $\Sigma(s)$.

Definition 2. *The* signature set Σ_G *for an RDF graph G is the set of all signatures occurring in it, i.e.,*

$$\Sigma_G = \{\Sigma_G(s) \mid \exists p, o : \langle s, p, o\rangle \in G\}$$

Definition 3. *A signature* $\Sigma(s_1)$ subsumes *a signature* $\Sigma(s_2)$ *iff*

$$\Sigma(s_1) \subseteq \Sigma(s_2)$$

Definition 4. *The* signature subsumption graph $\mathcal{G}_G$ *for an RDF graph G is the directed acyclic graph with vertices* Σ_G *and edges according to the subsumes relation between signatures, i.e.,* $\mathcal{G}_G = (\Sigma_G, \subseteq)$*.*

The simplified signature subsumption graph $\mathcal{G}'_G$ *for an RDF graph G is the graph that results from the signature subsumption graph by deleting all edges that can be reconstructed from the transitivity and reflexivity of* $\subseteq$*.*

Note that the (simplified) signature subsumption graph has, in general, more than one "root." Adding $\emptyset$ and $\bigcup_s \Sigma_G(s)$ (for all subjects s in G) turns it into a lattice. We will see later (Sect. 3.1) that adding $\emptyset$ is used for implementing the basic operators on RDF graphs.

Example 1. Let's consider the simple RDF graph depicted in Fig. 1.

The signatures for the four subjects $\text{Person}, p1, p2, p3$ that occur in P are:

$$\begin{aligned}
\Sigma_P(\text{Person}) &= \{\text{rdf : type}\}\\
\Sigma_P(p1) &= \{\text{rdf : type, rdfs : label, firstName, lastName}\}\\
\Sigma_P(p2) &= \{\text{rdf : type, firstName, lastName, email, homepage}\}\\
\Sigma_P(p3) &= \{\text{rdf : type, firstName, lastName}\}
\end{aligned}$$

We therefore have the following signature set:

$$\Sigma_P = \left\{\begin{array}{l}\{\text{rdf : type}\}, \{\text{rdf : type, rdfs : label, firstName, lastName}\},\\ \{\text{rdf : type, firstName, lastName, email, homepage}\},\\ \{\text{rdf : type, firstName, lastName}\}\end{array}\right\}$$

The simplified signature subsumption graph $\mathcal{G}_P$ is shown in Fig. 2.

[4] For simplicity, an RDF graph G is a set of subject-predicate-object triples $\langle s, p, o\rangle$. Special aspects of RDF like BNodes, reification, containers, and datatypes are not handled in this paper.

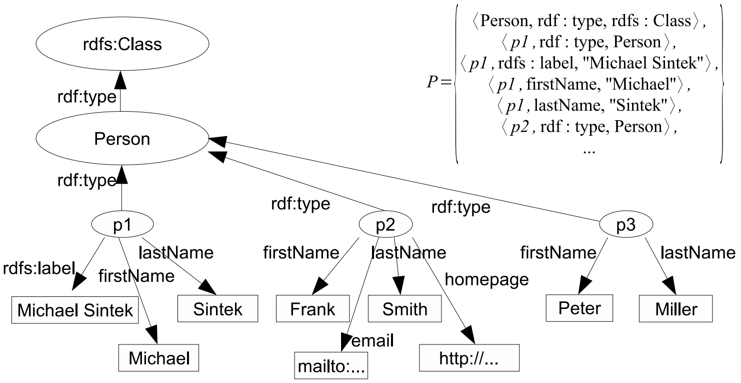

Fig. 1. A Sample RDF Graph: Persons

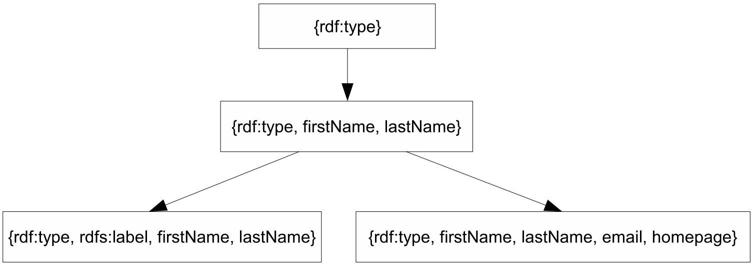
{rdf:type}
{rdf:type, firstName, lastName}
{rdf:type, rdfs:label, firstName, lastName}
{rdf:type, firstName, lastName, email, homepage}

Fig. 2. Sample Simplified Signature Subsumption Graph

2.2 Signature Tables

The basis of our approach is the storage of an RDF graph entirely in tables that correspond to the signatures occurring in the graph. These tables are defined in the following.

Definition 5. *The* signature table $\mathcal{T}_G(\{p_1, \ldots, p_n\})$ *for a signature* $\{p_1, \ldots, p_n\} \in \Sigma_G$ *for an RDF graph G is a two dimensional table with headings* $(\text{rdf:about}, p_1, \ldots, p_n)$ *(where the p_i are canonically ordered) and entries as follows: for each subject s in G with $\Sigma_G(s) = \{p_1, \ldots, p_n\}$, there is exactly one row in the table, where the rdf:about column contains s and column p_i contains the set of values for this property on s, i.e., $\{v \mid \langle s, p_i, v\rangle \in G\}$.*

Definition 6. *The* signature table set $\mathcal{T}_G$ *for an RDF graph G is defined as* $\mathcal{T}_G = \{\mathcal{T}_G(s) \mid s \in \Sigma_G\}$.

The signature table set for Ex. 1 ($\mathcal{T}_P$) looks like this:

rdf:about	rdf:type
Person	rdfs:Class

rdf:about	rdf:type	firstName	lastName
*p*3	Person	“Peter”	“Miller”

rdf:about	rdf:type	rdfs:label	firstName	lastName
$p1$	Person	"Michael Sintek"	"Michael"	"Sintek"

rdf:about	rdf:type	firstName	lastName	email	homepage
$p2$	Person	"Frank"	"Smith"	mailto:...	http://...

2.3 Algebraic Database Operations

Now that we have mapped an RDF graph to a set of tables,[5] we can lay the foundation for queries (and rules) by defining the algebraic operations used in (relational) databases. We define two sets of database operators, those directly operating on RDF graphs and those operating on the resulting tables.

On RDF graphs, we define only two operators, namely the projection $\dot{\pi}$ and a combined projection and selection $[\dot{\pi}\dot{\sigma}]$. On the resulting tables, we allow the usual set of algebraic operators known from relational databases, i.e., $\pi, \sigma, \times, \bowtie, \cup, \cap, -, \ldots$.

Definition 7. *The* projection $\dot{\pi}$ *on an RDF graph G for a property tuple $(p_1, \ldots, p_n)$ is defined as follows:*

$$\dot{\pi}_{(p_1,\ldots,p_n)}(G) = \bigcup \pi_{(p_1,\ldots,p_n)}(t)$$
$$\textit{for all } t = \mathcal{T}_G(s) \textit{ with } s \in \Sigma_G \textit{ and } \{p_1, \ldots, p_n\} \subseteq s$$

where π is the normal database projection operator slightly modified to work on non-normalized tables (since the entries are set-valued).

It would be sufficient to have only the projection operator $\dot{\pi}$ defined on RDF graphs, as we allow the full range of database operators to be applied to the resulting tables. Since all relational algebra expressions can be reformulated such that some projections occur first, we do not need any of the other operators to directly work on RDF graphs.

But this would mean that we have to copy all tuples from signature tables as the first step, which would not be very wise for efficiency reasons. Therefore, we also define a combined selection and projection operator, $[\dot{\pi}\dot{\sigma}]$.[6]

Definition 8. *The* projection-selection $[\dot{\pi}\dot{\sigma}]$ *on an RDF graph G for a property tuple $p = (p_1, \ldots, p_n)$ and a condition C is defined as follows:*

$$[\dot{\pi}\dot{\sigma}]^C_p(G) = \pi_p \bigcup (\sigma_C \circ \pi_{p'})(t)$$
$$\textit{for all } t = \mathcal{T}_G(s) \textit{ with } s \in \Sigma_G \textit{ and } p' \subseteq s$$
$$\textit{and } p' = \{p_1, \ldots, p_n\} \cup \text{properties}(C)$$

where π and σ are the normal database selection operators (modified as π above), and properties(C) *is the set of properties that occur in C.*

[5] or, in the case of multiple RDF graphs, to several sets of tables, allowing access to named graphs which are nicely supported by our approach

[6] Note that we do not define an operator $\dot{\sigma}$ on RDF graphs since signature tables that would naturally be involved in a single selection are of varying arity.

The essential parts of these two definitions are the subsumption restrictions ($\{p_1, \ldots, p_n\} \subseteq s$ and $p' \subseteq s$, resp.), i.e., in both cases we only consider the signature tables with signatures that are subsumed by the properties occurring in the operators.

2.4 RDF Schema Semantics

Although RDFBroker is designed to work efficiently on RDF graphs without RDFS (or any other schema), we believe that is very important to provide an efficient implementation of the RDFS semantics as RDFS is used as ontology language in most applications dealing with mass data and therefore being a target for our system.

In the following, we define the (simplified)[7] RDFS semantics G^{RDFS} of an RDF graph mainly with the help of the (conjointly computed) transitive closures of rdfs:subClassOf and rdfs:subPropertyOf, the class propagation for rdf:type, and the value propagation for rdfs:subPropertyOf,[8] which follows directly from the RDF/S model theory [2]. Note that the naive approach to compute the transitive closures and propagations separately (or in any fixed order) is not correct (e.g., this would not catch the case where one defines a subproperty of rdfs:subClassOf (or even rdfs:subPropertyOf itself)).

Definition 9. *The* (simplified) RDFS immediate consequence operator[9] T_{RDFS} *for an RDF graph G is defined as follows:*[10]

$$\begin{aligned} T_{\mathrm{RDFS}}(G) = G &\cup \{\langle p, \mathrm{sPO}, q\rangle \mid \{\langle p, \mathrm{sPO}, r\rangle, \langle r, \mathrm{sPO}, q\rangle\} \subseteq G\} \\ &\cup \{\langle p, \mathrm{sCO}, q\rangle \mid \{\langle p, \mathrm{sCO}, r\rangle, \langle r, \mathrm{sCO}, q\rangle\} \subseteq G\} \\ &\cup \{\langle s, \mathrm{type}, c\rangle \mid \{\langle c', \mathrm{sCO}, c\rangle, \langle s, \mathrm{type}, c'\rangle\} \subseteq G\} \\ &\cup \{\langle s, p, o\rangle \quad\; \mid \{\langle p', \mathrm{sPO}, p\rangle, \langle s, p', o\rangle\} \subseteq G\} \end{aligned}$$

Theorem 1. *The (simplified) RDF Schema semantics of an RDF graph G is the fixpoint of* T_{RDFS}*:*

$$G^{\mathrm{RDFS}} = \bigcup_{n \in \mathbb{N}_0} T^n_{\mathrm{RDFS}}(G \cup AP)$$

where AP are the axiomatic triples $\langle \mathit{rdf{:}type}, \mathit{rdfs{:}domain}, \mathit{rdfs{:}Resource}\rangle, \ldots$ *as defined in the RDF/S model theory.*

We will show in Sect. 3.4 that T_{RDFS} can be directly used to implement G^{RDFS}.

[7] We explicitly ignore here some details of the RDFS semantics, namely rdfs:domain and rdfs:range, as their correct handling is sometimes counterintuitive when coming from a database or logic programming perspective.

[8] $\{\langle p, \mathrm{rdfs{:}subPropertyOf}, q\rangle, \langle s, p, o\rangle\} \subseteq G^{\mathrm{RDFS}} \rightarrow \langle s, q, o\rangle \in G^{\mathrm{RDFS}}$

[9] which is similar to the normal immediate consequence operator T_P

[10] with sPO = rdfs:subProperyOf, sCO = rdfs:subClassOf, and type = rdf:type

2.5 Sample Queries

In the following, we give some sample queries for the RDF graph P from Exa. 1.

Example 2. ‘Return first name and last name for all persons.’

$$r = [\dot{\pi}\dot{\sigma}]^{\text{rdf:type=Person}}_{(\text{firstName,lastName})}(P)$$

Example 3. ‘Find first name, email address, and homepage for the person with last name “Smith”’:

$$r = [\dot{\pi}\dot{\sigma}]^{\text{rdf:type=Person}\wedge\text{lastName=''Smith''}}_{(\text{firstName,email,homepage})}(P)$$

The current implementation does not support complex selection conditions, we therefore have to evaluate the query in several steps:[11]

$$r = \pi_{(\bar{1},\bar{3},\bar{4})}(\sigma_{\bar{2}=\text{''Smith''}}([\dot{\pi}\dot{\sigma}]^{\text{rdf:type=Person}}_{(\text{firstName,lastName,email,homepage})}(P)))$$

3 Implementation

The in-memory variant of RDFBroker is currently being implemented with JDK 1.5. It uses Sesame’s RIO parser, which could easily be replaced by any streaming parser generating “add statement” events.

Most of the concepts of Sect. 2 have directly corresponding implementations plus appropriate index structures (e.g., there is an index for each column in a signature table, currently realized as a hash table).

Our approach benefits heavily from well-known database optimization techniques. E.g., queries are reformulated such that selections and joins on multiple columns access small tables and columns which hold many different values (and are therefore discriminating) first, thus reducing the size of intermediate results as fast as possible.

In the following, we describe some aspects of the implementation in detail (some of which have not yet been realized in our first prototype, like updates and the RDFS semantics).

3.1 The Operators $\dot{\pi}$ and $[\dot{\pi}\dot{\sigma}]$

The efficient implementation of $\dot{\pi}$ and $[\dot{\pi}\dot{\sigma}]$, which form the basis of all queries, is obviously vital for our RDF store. The implementation of these operators requires the lookup of the signature tables $\mathcal{T}_G(s)$ for all signatures s where the properties occurring in the operator subsume s, as defined in Def. 7 and Def. 8.

The signature table lookup is performed by first looking up the matching signatures in the simplified signature subsumption graph and then retrieving the associated signature tables.

The signature lookup for properties $p = \{p_1, \ldots, p_n\}$ is performed by the following (informally described) algorithm, which is also exempflified in Fig. 3:

[11] Column numbers for relational operators are marked with a bar on top: $\bar{1}, \bar{2}, \bar{3}, \ldots$.

(a) add $\emptyset$ as an artificial root to the simplified signature subsumption graph $\mathcal{G}'_G$ (making it a "meet-semilattice"[12])
(b) starting at $\emptyset$, find all minimal signatures s which are subsumed by p, i.e., for which $p \subseteq s$ holds
(c) add all signatures which are subsumed by these minimal signatures (simply by collecting all signatures reachable from the minimal ones using a depth-first walk)

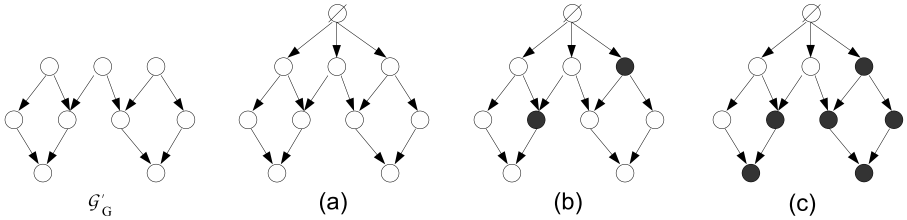

Fig. 3. Algorithm: Lookup of Signatures

3.2 Merging of Signature Tables

An important source for optimization are the signature tables and their organization in the subsumption graph. First tests with the system revealed that sometimes many small tables are generated which are responsible for overhead, which we wish to avoid. The obvious solution for this is pruning the subsumption tree by merging small adjacent signatures tables (i.e., which share many properties) or merging small tables with subsumed big ones. A sketch for a greedy algorithm (which tries to minimize the number of NULL values to be added and the number of merge operations) is as follows:

(a) pick the smallest signature table and mark it to be merged
(b) pick the smallest signature table adjacent to a signature table marked to be merged (and sharing substantially many properties) and mark it also
(c) repeat (b) until the size of all marked signature tables exceeds some threshold; if the threshold cannot be exceeded, mark the directly subsumed table with the smallest signature to be merged
(d) merge all marked signature tables
(e) repeat (a) – (d) until no single signature table exists that is smaller than some threshold

Fig. 4 shows the result of applying this algorithm on some sample signature subsumption graph.

The resulting signature subsumption graphs are often very similar to user defined schemas, which is what we expect since co-occurrence of properties is the basis for (manually) defining classes in an schema/ontology. We therefore expect our approach to perform similar to mapping an RDF Schema to an (object-) relational database directly.

[12] i.e., an partially ordered set where for any two elements there exists an infimum (greatest lower bound) but not necessarily a supremum in the set

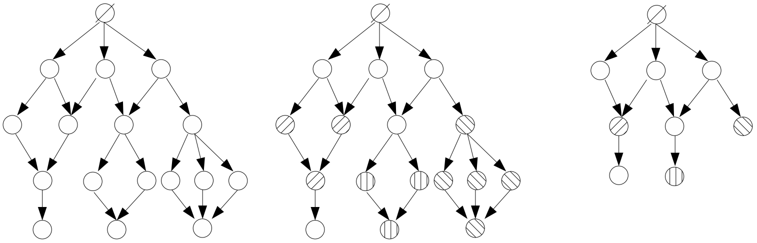

Fig. 4. Algorithm: Merging of Signatures

3.3 Updates

Updates (i.e., inserts, deletes, and value updates) can easily be realized on top of our RDF store, but some operations come with performance penalties when they change the signature of a subject resource, which then has to migrate from one signature table to another. To improve this, special methods will be used to allow mass updates to be handled efficiently. In the case of mass inserts, e.g., either a whole RDF graph is added all at once, thus allowing first the property values per subject to be collected (as it is done for parsing already), or applications use methods like `add(Resource, Map<URI,Value>)` to explicitly add many property values for one subject at once.

Since the merging of signature tables results in allowing NULL values in merged tables, this reduces the likelihood that database rows have to migrate from one table to another.

3.4 RDF Schema

In general, two main approaches exist to implement the RDFS semantics: compute the triples resulting from the RDFS semantics in advance and materialize them, or compute them on demand. Since our main goal currently is high-performance for queries, we decided to take the first approach.

Materializing the RDF Schema semantics G^{RDFS} of an RDF graph G can directly be based on Theorem 1, analogously to the realization of logic programming with the well-known immediate consequence operator T_P (for a deductive database / logic program P) [3]. For this, the semi-naive bottom-up evaluation is used, i.e., the T_{RDFS} operator is not evaluated on the full data set from all previous rounds in the fixpoint computation, but restricted in the sense that certain parts of T_{RDFS} are checked only against the data newly produced in the previous round. Furthermore, the materialization of the propagation for rdf:type is not necessary and can be handled by query rewriting, which will drastically reduce the number of additionally created data.

3.5 Natural Data Handling and Querying

One of our promises is that our RDF store is application-adequate. Therefore, we allow queries and data handling using traditional programming languages (in our prototype implementation, Java) in a very natural way.

Queries. Higher-level query languages like SPARQL have the disadvantage compared to using Java that they do not nicely cooperate with Java data: e.g., query parameters have to be translated into textual representations matching the query language syntax, which involves the typical problems of quoting special characters, character encoding, problems with malformed queries at runtime, no support from typical development environments,[13] etc.

Example 4. For the Exa. 3 query, the Java code looks like this (`C_equals` creates an equality selection condition and projection/selection indices are 0-based):

```
p.projectAndSelect(
     p.properties(FIRSTNAME, LASTNAME, EMAIL, HOMEPAGE),
     p.C_equals(p.RDF_TYPE, PERSON))
  .select(p.C_equals(1, p.literal("Smith")))
  .project(0,2,3);
```

Comparing Java-based queries (using algebraic operators) with standard declarative query languages is difficult. Both approaches have benefits and drawbacks: Java-based queries allow simple debugging since intermediate results are available. Manual optimization is easily possible without having to know much about the query engine's internals. On the other hand, declarative query languages are easier to read (since inherently they describe only the *goal* of the query in a simpler syntax), and automatic optimization can be done to some degree.

Data Handling. Normal RDF store APIs provide only triple-based methods for manipulating the RDF data which is uncomfortable for most applications as the typical view on an application's data is an object-oriented view. With additional tools such as RDF2Java [4] or RDFReactor [5] that introduce an abstraction layer this problem can be solved from the programmer's point of view. However, these approaches still map to triples internally. Using RDFBroker and its object/resource-centric data representation, it is possible to provide a natural data interface without mapping between data representations.

4 Evaluation

For our first in-memory prototype, we evaluated the (RDFBroker-specific) distribution of signature tables and load times, memory consumption, and query execution times by comparing them to the behavior of other freely available RDF stores, using several queries on a large database and measuring database load times on three different databases.

We used RDF data from TAP[14] which comes with RDF files of sizes up to about 300 MB. In particular, for evaluating load times

[13] Standard IDEs feature autocompletion which helps a lot in coding but does not work with queries which are just strings to the IDE.

[14] `http://sp11.stanford.edu/`

tuples per table	tuples	tables
1–10	8941	4137
11–100	15375	506
101–1000	13730	62
1001–10000	0	0
10001–100000	117288	2
100001–1000000	130939	1

Fig. 5. Signature Table Distribution

	1.7MB DB		24MB DB		298MB DB	
	load time	memory	load time	memory	load time	memory
RDFBroker	500ms	66MB	7,500ms	102MB	102,000ms	945MB
Jena	800ms	36MB	11,000ms	70MB	151,000ms	822MB
Sesame	300ms	28MB	4,500ms	83MB	74,000ms	408MB

Fig. 6. Load Times and Memory Consumption of RDFBroker, Sesame, and Jena

we used `swirl-SiteArchitectureEmporis.rdf` (1.7MB and 17,086 triples), `swirl-SitePlacesWorldAirportCodes.rdf` (24MB and 245,578 triples), and `swirl-SiteMoviesIMDB.rdf` (298MB and 3,587,064 triples). For testing query performance, we used `swirl-SiteMoviesIMDB.rdf` exclusively. The RDF store implementations we compared are Sesame [6], Jena [7], and of course RDFBroker.[15] The evaluation environment was an Athlon 64 3000+ (3530 bogomips) with 2GB RAM running Linux 2.6.12.3 i686 and Sun Java 1.5.0-2. The evaluation software can be found on the RDFBroker project website.

Signature Table Distribution. With `swirl-SiteMoviesIMDB.rdf` (298MB, 3,587,064 triples, 286,273 resources in subject position, 4,708 signature tables), we got the distribution of signature tables shown in Fig. 5, i.e., there are 4,137 tables of size 1–10 tuples that hold a total of 8941 tuples, . . . , and there is exactly one table that holds 130,939 tuples. This is exactly what we expect for mass data: most of the data is in very few tables (in this case, three tables hold 87% of all tuples).

Load Times and memory consumption for the three RDF files are shown in Fig. 6. Since RDFBroker currently uses Sesame's "Rio" parser, its load times are similar to Sesame's performance. The creation of signature tables and exhaustive indices explains the higher values.

Queries. We measured the execution times and memory consumption (see Fig. 7) for several queries, ranging from simple "retrieval" of property values over path expressions to joins that are not representable as path expressions. In Appendix A, we list the SeRQL queries for evaluating Sesame for completeness.

[15] For all implementations we enabled RDF validation on load and disabled inferencing.

	Query 1		Query 2		Query 3		Query 4	
	time	memory	time	memory	time	memory	time	memory
RDFBroker	70ms	4MB	1200ms	63MB	260ms	4MB	160ms	10MB
Jena	4300ms	82MB	8700ms	26MB	70ms	3MB	-	-
Sesame	1400ms	24MB	2200ms	46MB	50ms	2MB	-	-

Fig. 7. Query Times and Memory Consumption of RDFBroker, Sesame, and Jena

Query 1: 'return some interesting properties of all movies': This operates heavily on the queried instances' properties. As to be expected, RDFBroker performs very fine in this case since the signature tables nicely match the query's structure.

$$[\dot{\pi}\dot{\sigma}]^{\text{rdf:type=imdb:Movie}}_{\text{(rdf:about,rdfs:label,imdb:PropertyCountry,PropertySound\_Mix...)}}$$

Query 2: 'find names for persons casted in movies': This is a join query that in many high-level RDF query languages is expressed as a path expression. Since this is a very common query, most systems come up with optimized algorithms for it. Still, RDFBroker's performance was the best.

$$[\dot{\pi}\dot{\sigma}]^{\text{rdf:type=imdb:Movie}}_{\text{(rdf:about,rdfs:label,imdb:creditedCast)}} \bowtie_{\bar{3}=\bar{1}} [\dot{\pi}\dot{\sigma}]^{\text{rdf:type=imdb:Person}}_{\text{(rdf:about,rdfs:label)}}$$

Query 3: 'find persons playing in movies three cast hops separated from Kevin Bacon': This query is related to the Bacon Number.[16] Since it is a path expression using only one property, RDFBroker is not optimized for this kind of query, and has to walk over thousands of tables multiple times.[17]

Query 4: 'find movies with same title and return some useful properties on them, like release year, cast, genre, . . . ':

$$[\dot{\pi}\dot{\sigma}]^{\text{rdf:type=imdb:Movie}}_{\text{(rdf:about,rdfs:label,imdb:creditedCast,...)}} \bowtie_{\bar{2}=\bar{2},\bar{1}\neq\bar{1}} [\dot{\pi}\dot{\sigma}]^{\text{rdf:type=imdb:Movie}}_{\text{(rdf:about,rdfs:label,...)}}$$

RDFBroker evaluated this query in less than 200ms, while Sesame and Jena were not able to finish it (we stopped after 30 minutes). Probably joins that are not path expressions are not handled "properly" in the sense that they are evaluated by first computing the complete cartesian product.

Conclusion. For standard queries, the RDFBroker approach performs very well. The prototype's memory consumption is higher than that of other RDF stores. Both characteristics are most likely due to the fact that currently all database columns get indexed which leads to high performance but counteracts the potential benefit of small memory footprint that is inherent of the approach. We will address this in future RDFBroker versions, as well as implementing table merging to reduce the overhead of walking over thousands of tables for queries accessing only few properties.

[16] http://en.wikipedia.org/wiki/Bacon_number

[17] We expect this kind of query to perform much better in RDFBroker when we use table merging.

5 Related Work

Since most RDF frameworks such as Sesame [6] or Jena [7] allow using RDBMSs as storage backends, quite a lot of previous work on this area is available. An overview of different RDF frameworks can be found in [8]; an overview of different approaches of mapping RDF to standard DBs can be found in [9] and [10].

There are several RDF frameworks that rely on native storage such as YARS [11], Redland [12], or BRAHMS [13]. Often, these implementations perform superior with special types of queries. BRAHMS, for example, is very fast when searching semantic associations—semantic association paths leading from one resource to another resource.

In [10], several RDBMS mapping characteristics are presented along with a generic performance comparison of the approaches described. The approaches outlined use one or more tables with at most three columns, one table representing either triples, properties, or RDFS class instances. The drawback of this compared to our approach is that properties of one resource get scattered over multiple tables and/or rows—an advantage is that no support for sets in table cells is needed.

The definition of the mapping characteristics in the same paper are a bit too narrow and cannot be applied to our approach easily—while, for example, no schema is needed for our approach, it is not *schema-oblivious* in terms of the cited paper since we do not use only one table for storing triples.

In [14], an approach to derive table layout from the data or using machine learning-based query analysis approaches is described. This leads to high initial costs and requires a large amount of data for initial training. Our approach is much more lightweight especially concerning initial setup. However, we do not support query analysis at all.

For performance comparison and test data generation several tools have been described, mostly using using a Zipfian distribution when generating class instances. See [10] and Store-Gen [14] for a further description of synthetic data generators.

6 Conclusions and Future Work

In this paper, we introduced RDFBroker, an RDF store using signatures as the basis for storing arbitrary RDF data. Since signatures and their organization in a lattice-like structure approximate user-defined schemas/ontologies (and thus also make RDF data accessible from applications in a more natural way), RDFBroker performs similar to hand-coded (object-) relational databases. Comparison of our first prototype with other RDF stores showed that even for the in-memory case queries can be evaluated more efficiently than with standard techniques using triples as the basis for storage organization.

Our approach can handle, on standard hardware, already fairly large knowledge bases (RDF files with several hundred MBs) in main memory. For mass data, we will make use of on-disk databases (which requires support of multiplevalued attributes), which lets us expect even higher differences in performance compared to existing RDF stores which store RDF data in on-disk databases.

We also intend to support query and esp. rule language standards like SPARQL and the result of the W3C Rule Interchange Format Working Group that has just been founded, where we will profit from well-investigated deductive database technologies (like magic set transformation). We furthermore plan to provide a natural programming API for manipulation of the RDF data stored in RDFBroker, where the interface classes can be build as known from our RDF2Java [4] tool. Our future plans also include using a P2P network (or grid) to improve performance by using in-memory (instead of on-disk) stores in peers and using the signature subsumption graph for distributing the data, routing queries, and developing appropriate peer leave/join algorithms.

Acknowledgments. This work has been supported in part by the SmartWeb project, which is funded by the German Ministry of Education and Research under grant 01 IMD01 A, and by the NEWS project, which is funded by the IST Programme of the European Union under grant FP6-001906.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American (2001) 34–43
2. Hayes, P.: RDF semantics (2004) W3C Recommendation. `http://www.w3.org/TR/rdf-mt/`.
3. Lloyd, J.W.: Foundations of logic programming. Springer-Verlag New York, Inc., New York, NY, USA (1984)
4. Sintek, M., Schwarz, S., Kiesel, M.: RDF2Java (2005) `http://rdf2java.opendfki.de/`.
5. Völkel, M., Sure, Y.: RDFReactor - from ontologies to programmatic data access. Poster and Demo at ISWC2005 (2005)
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: An architecture for storing and querying rdf data and schema information. In Fensel, D., Hendler, J.A., Lieberman, H., Wahlster, W., eds.: Spinning the Semantic Web, MIT Press (2003) 197–222
7. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient RDF storage and retrieval in Jena2. In Cruz, I.F., Kashyap, V., Decker, S., Eckstein, R., eds.: SWDB. (2003) 131–150
8. SWAD: SWAD-europe deliverable 10.1: Scalability and storage: Survey of free software / open source RDF storage systems (2002) http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/.
9. SWAD: SWAD-europe deliverable 10.2: Mapping semantic web data with RDBMSes (2003) `http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report`.
10. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking database representations of RDF/S stores. In Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A., eds.: International Semantic Web Conference. Volume 3729 of Lecture Notes in Computer Science., Springer (2005) 685–701

11. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: 3rd Latin American Web Congress, Buenos Aires - Argentina, Oct. 31 - Nov. 2 2005. (2005)
12. Beckett, D.: The design and implementation of the redland RDF application framework. Computer Networks **39**(5) (2002) 577–588
13. Janik, M., Kochut, K.: BRAHMS: A workbench RDF store and high performance memory system for semantic association discovery. In: 4th International Semantic Web Conference. (2005)
14. Ding, L., Wilkinson, K., Sayers, C., Kuno, H.A.: Application-specific schema design for storing large RDF datasets. In Volz, R., Decker, S., Cruz, I.F., eds.: PSSS. Volume 89 of CEUR Workshop Proceedings., CEUR-WS.org (2003)

Appendix A. SeRQL Queries Used in the Evaluation

Query 1: ‘return some interesting properties of all movies’

```
SELECT MovieLabel, Year, Runtime, Color, Language, Country, Sound
    FROM {MovieURI} rdf:type {imdb:Movie}; rdfs:label {MovieLabel};
                    imdb:PropertyCountry {Country}; ...
    USING NAMESPACE imdb = <http://data.imdb.com/data/>
```

Query 2: ‘find names for persons casted in movies’

```
SELECT Movie, Title, Cast, PersonName
    FROM {Movie} rdf:type {imdb:Movie}; rdfs:label {Title};
                 imdb:creditedCast {Cast} rdfs:label {PersonName}
    USING NAMESPACE imdb = <http://data.imdb.com/data/>
```

Query 3: ‘find persons playing in movies three cast hops separated from Kevin Bacon’

```
SELECT DISTINCT PersonName
  FROM {StartPerson} imdb:PropertyActor_filmography {Movie1}
    imdb:creditedCast {Cast1} imdb:PropertyActor_filmography {Movie2}
      imdb:creditedCast {Cast2} imdb:PropertyActor_filmography {Movie3}
        imdb:creditedCast {Cast3} rdfs:label {PersonName}
    WHERE StartPerson = imdb:PersonKevin_Bacon_8_July_1958
    USING NAMESPACE imdb = <http://data.imdb.com/data/>
```