Optimized Index Structures for Querying RDF from the Web

Andreas Harth Stefan Decker

Digital Enterprise Research Institute (DERI)
National University of Galway, Ireland
University Road
Galway, Ireland

firstname.lastname@deri.org

Abstract

Storing and querying Resource Description Framework (RDF) data is one of the basic tasks within any Semantic Web application. A number of storage systems provide assistance for this task. However, current RDF database systems do not use optimized indexes, which results in a poor performance behavior for querying RDF. In this paper we describe optimized index structures for RDF, show how to process and evaluate queries based on the index structure, describe a lightweight adaptable implementation in Java, and provide a performance comparison with existing RDF databases.

1. Introduction

In recent years RDF has emerged as the prevalent data format for the Semantic Web. RDF is a graph-based data format which is schema-less and self-describing, meaning that the labels of the graph within the graph describe the data itself. Many applications that deal with RDF have the need to store the data persistently and perform queries on the data set.

Systems such as Jena2 [17], Sesame [5], rdfDB [7], Redland [2], Kowari [1], FORTH RDF Suite [1] and others provide a storage infrastructure for RDF data. However, after looking at the index structures of the systems, most of the systems use an index structure which do not support typical query scenarios for data from the Web which results in poor query answering performance in some cases.

Furthermore, to judge a given piece of information we usually need to look at the context of the information. As an example for context consider the source of a piece of information. We might want to trust information coming from whitehouse.gov more than information from whitehouse.com (or vice versa). However, the notion of context is missing in most current RDF storage systems.

In this paper we adapt database techniques for RDF data storage and indexing, which results in improved query answering performance and capabilities compared to current RDF storage systems. Our paper identifies and combines several techniques from the database area to arrive at a system with improved efficiency for storing and retrieving RDF. Specifically, this paper makes the following contributions:

- We define and realize a complete index structure including full-text indexes for RDF triples with context.
- We describe YARS, a lightweight open-source implementation of the index structure in Java with small footprint which can be embedded into an application and adapted to special application needs.

The remainder of the paper is organized as follows: Section 2 reviews the data model for data and queries and introduces an example. In Section 3, we present optimized index structures tailored for speeding up queries on RDF data. We describe how to perform query processing for select-project-join (SPJ) queries in Section 4. Section 5 presents the architecture and implementation of Yet Another RDF Store (YARS), our prototype system. In Section 6 we evaluate YARS and in Section 7 we discuss related work. Section 8 concludes the paper.

2. Preliminaries

In this section we first give an example of RDF data found on the Web and then define the data model and query language used.

2.1 Example

FOAF (Friend of a Friend) is a vocabulary "describing people, the links between them and the things they create and

[1]`http://www.kowari.org/`

do" [2]. FOAF is frequently used to describe a person's homepage in machine-readable format, and is a good example for data that is distributed across the Semantic Web. Figure 1 shows a small RDF graph describing the authors of this paper using the FOAF vocabulary.

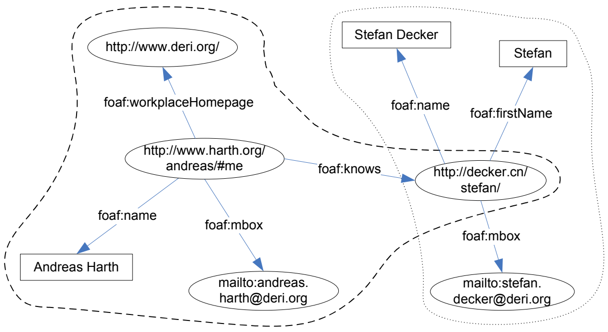

Figure 1: FOAF example originating from two different sources describing the authors. The contexts of the graphs are `http://sw.deri.org/~aharth/foaf.rdf` and `http://www.isi.edu/~stefan/foaf.rdf`.

2.2 Data Model

We begin with defining the standard RDF data model as described in various W3C Recommendations [13], [10].

Definition 2.1 *(RDF Triple, RDF Node) Given a set of URI references $\mathcal{R}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple (s, p, o) $\in (\mathcal{R} \cup \mathcal{B}) \times \mathcal{R} \times (\mathcal{R} \cup \mathcal{B} \cup \mathcal{L})$ is called an RDF triple, An element of an RDF triple is called an RDF node.*

In such a triple, s is called the subject, p the predicate, and o the object.

Although the RDF specification itself does not define the notion of context [8], usually applications require context to store various kinds of metadata for a given set of RDF statements. E.g. MacGregor and Ko [12] reported on an application that stored information about ships and their position and argued that quads are a suitable formalism to capture context.

The interpretation of context is depends on the application. For example, in an information integration use case, the context is the URI of the file or repository from which a triple originated. Capturing provenance is one of the fundamental necessities in open distributed environments like the Web, where the quality of data has to be judged by its origin. Contexts are useful in other application scenarios as well, such as versioning or access control.

[2]`http://www.foaf-project.org/`

Definition 2.2 *(Triple in Context) A pair (t, c) with t be a triple and c $\in (\mathcal{R} \cup \mathcal{B})$ is called a triple in context c.*

Please note that a triple ((s, p, o), c) in context c is equivalent to the quad (s, p, o, c).

2.3 N3 Syntax

We use Notation3 (N3) as a syntax for RDF. For a full description of N3 see [3]. To make this paper self-contained, we introduce the basic syntactic N3 primitives. Brackets (<>) denote URIs, quotes ("") denote Literals, and blank node identifiers start with "_ :". There exists a number of syntactic shortcuts, for example ";" to introduce another predicate and object for the same subject. Namespaces can be introduced with the @prefix keyword. Figures 2 and 3 show the N3 syntax for the two contexts in the example depicted in Figure 1.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/ .

<http://www.harth.org/andreas/#me>
 foaf:name "Andreas Harth" ;
 foaf:mbox <mailto:andreas.harth@deri.org> ;
 foaf:workplaceHomepage <http://www.deri.org/>;
 foaf:knows <http://decker.cn/stefan/> .
```

Figure 2: N3 syntax of context `http://sw.deri.org/~aharth/foaf.rdf` in Figure 1.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/ .

<http://decker.cn/stefan/>
 foaf:name "Stefan Decker" ;
 foaf:firstName "Stefan" ;
 foaf:mbox <mailto:stefan.decker@deri.org> .
```

Figure 3: N3 syntax of context `http://www.isi.edu/~stefan/foaf.rdf` in Figure 1.

N3 offers an extension to the RDF data model with primitives for variables and grouping of graphs. Variables in N3 are denoted using a question mark "?". Within N3, RDF subgraphs can become the subject or object of a statement, using "{}". N3 is able to represent lists of nodes using "()".

2.4 YARS Query Language

To be able to express queries, two sets of N3 language extensions are required. For the extensions, we introduce two namespaces, `ql`[3] and `yars`[4]. The first set of language extensions, namely the predicates `ql:where` and `ql:select` enable us to formulate queries. A query consists of a `ql:where` clause which comprises a set of statements that can contain variables. An optional `ql:select` clause determines the format of the result set.

The second set of language extensions define YARS-specific query primitives and are listed below.

- The `yars:context` predicate denotes that the subgraph grouped in the subject is occurring in the context provided as the object. The `yars:context` predicate enables us to express our notion of context within the RDF data model. In the example in Figure 4 the `yars:context` predicate specifies that we only want to query statements originating from `http://sw.deri.org/˜aharth/foaf.rdf`.
- The `yars:keyword` predicate allows to represent keyword-containment requirement for subjects. E.g., the `yars:keyword` in the example in Figure 4 requires that the variable `?n` needs to contain the string “Harth”.
- The predicate `yars:prefix` can be used to specify that a variable in the subject has to match on the prefix denoted in the object. For example, the `yars:prefix` predicate in the example in Figure 4 specifies that only statements using the FOAF predicates should be used for query answering.
- The predicate `yars:count` is used to query occurrence counts for statements quoted as subject. The object is a variable that is bound to the occurrence count of the access pattern during query evaluation.

3. Index Organization

The goal of the index is to support evaluation of SPJ queries. At the lowest level, the index structure enables fast retrieval of quads, given any combination of subject (s), predicate (p), object (o) or context (c). We want to avoid expensive joins wherever possible, and therefore trade index space for retrieval time.

All our persistent indexes use B+-trees [6], a well understood data structure which support insert, deletes, and lookups (especially range lookups). Conceptually, we have

[3]`http://www.w3.org/2004/12/ql#`
[4]`http://sw.deri.org/2004/06/yars#`

```
<> ql:select { ?x ?p ?z . };
   ql:where {
     { ?x foaf:name ?n .
       ?n yars:keyword "Harth" .
       ?x ?p ?z .
       ?p yars:prefix foaf: .
     } yars:context
        <http://sw.deri.org/˜aharth/foaf.rdf> .
   } .
```

Figure 4: N3 query to get all FOAF information about the person with the name containing “Harth” from a specified source.

(key, value) pairs where retrieval based on key yields the value using few disk operations.

Our index structure contains two sets of indexes:

- the *lexicon* covers the string representations of an RDF graph ($\mathcal{R}, \mathcal{L}, \mathcal{B}$).
- the *quad indexes* cover the quads.

3.1 Lexicon

The lexicon indexes operate on the string representations of RDF nodes, and enable fast retrieval of object identifiers (OIDs) for RDF nodes. OIDs are represented and stored on disk as 64 bit longs. Since we reference RDF nodes in multiple indexes the mapping from string values to OIDs saves space. Also, processing and comparing OIDs is faster than comparing strings.

The lexicon consists of three different indexes: the *nodeoid* and *oidnode* indexes map strings to OIDs, and the keyword index is an inverted text index.

3.1.1 NodeOID and OIDNode Index

The *oidnode* and *nodeoid* indexes are used to map OIDs to string values of RDF nodes and vice versa. OIDs are assigned increasingly monotonically for each unique node that is inserted. OID 0 is a special OID that denotes a variable.

An alternative to keeping the *nodeoid* index is to compute the hash of the node and use the resulting number as an OID. However, hash functions with a small probability of collisions such as SHA1 or MD5 produce at least 128 bit keys for OIDs, which would increase the index size considerably.

Keeping a separate index for mapping string values to OIDs and storing the mapping in B+-trees has the advantage that we are able to perform prefix queries on node values, given that the *nodeoid* index is sorted lexographically. Table 1 shows the *nodeoid* index created for the example graph provided by Figure 1.

Key	Value
“Andreas Harth”	3
“Stefan”	14
“Stefan Decker”	11
<http://www.harth.org/andreas/#me>	1
<http://decker.cn/stefan/>	10
<http://sw.deri.org/~aharth/foaf.rdf>	4
<http://www.deri.org/>	8
<http://www.isi.edu/~stefan/foaf.rdf>	12
<http://xmlns.com/foaf/0.1/firstName>	13
<http://xmlns.com/foaf/0.1/knows>	9
<http://xmlns.com/foaf/0.1/mbox>	5
<http://xmlns.com/foaf/0.1/name>	2
<http://xmlns.com/foaf/0.1/workplaceHomepage>	7
<mailto:andreas.harth@deri.org>	6
<mailto:stefan.decker@deri.org>	15

Table 1: OIDs for the node values in the example dataset in the *nodeoid index*

3.1.2 Keyword Index

The prevalent type of queries used today to explore Web data are keyword queries. To speed up these type of queries, the lexicon keeps an inverted index on string literals to allow for fast full-text searches. Each literal is tokenized into words. Each word represents a key in the index, with a sorted list of OIDs as occurrences. Table 2 shows such an index constructed for the example. We use the full list storage scheme as described in [15], but keep OIDs instead of document identifiers for the hitlist. Keeping the number of hits helps to determine join ordering during query processing.

Key	No of hits	List of hits
“Andreas”	1	3
“Decker”	1	11
“Harth”	1	3
“Stefan”	2	11,13

Table 2: Keyword/hitlist pairs for literals in the inverted text index.

3.2 Quad Indexes

The following section shows that we only need a restricted number of indexes to cover all possible access patterns for RDF data.

3.2.1 Access Patterns

We want to avoid expensive joins whenever possible. Thus, we need an index that allows to lookup any combination of s, p, o, c directly rather than joining the results from lookups in multiple indexes.

The quad indexes are based on the notion of access patterns.

Definition 3.1 *(Access Pattern) An access pattern is a quad where any combination of s, p, o, c is either specified or a variable.*

For example, an access pattern could be a quad where only s is specified, and p, o, and c are variables. The access pattern (s:?:?:?) denotes all quads where the subject equals to s, whereas the other nodes have unspecified value. To compute the total number of access patterns we just have to consider that for each element of the quad (4) there exist 2 possibilities (either a node is specified, or it is a variable). Therefore the total number of access patterns is $2 * 2 * 2 * 2 = 16$. Table 3 shows all possible access patterns for quad lookups.

No	Access pattern	No	Access pattern
1	(?:?:?:?)	9	(s:?:o:c)
2	(s:?:?:?)	10	(?:?:o:c)
3	(s:p:?:?)	11	(?:?:o:?)
4	(s:p:o:?)	12	(?:?:?:c)
5	(s:p:o:c)	13	(s:?:?:c)
6	(?:p:?:?)	14	(s:p:?:c)
7	(?:p:o:?)	15	(?:p:?:c)
8	(?:p:o:c)	16	(s:?:o:?)

Table 3: Possible quad patterns; in total, there are 16 different patterns to cover all possible access combinations.

A naive implementation of a complete index on quads would need 16 indexes, one for each access pattern. Implementing a complete index in the naive way is prohibitively expensive in terms of index construction time and storage utilization. In the next section we show how we can cover all access patterns with just six indexes.

3.2.2 Combined Indexes

To reduce the number of indexes needed, we leverage the fact that B+-tree provide support for range or prefix queries. A combined index on s, p, o, and c for a quad (s, p, o, c) is able to support queries for access patterns 1 through 5 in Table 3. For example, a lookup for the access pattern (s:p:?:?) resolves to a prefix query for s and p on the *spoc* index. Therefore, we do not need to keep a separate index of s and p but can reuse the *spoc* index.

Using combined indexes reduces the number of necessary indexes to implement a complete index on quads to six. Table 4 shows the six indexes and which access pattern they cover.

spoc	poc	ocs
(?:?:?:?) (s:?:?:?) (s:p:?:?) (s:p:o:?) (s:p:o:c)	(?:p:?:?) (?:p:o:?) (?:p:o:c)	(?:?:o:?) (?:?:o:c) (s:?:o:c)
csp	**cp**	**os**
(?:?:?:c) (s:?:?:c) (s:p:?:c)	(?:p:?:c)	(s:?:o:?)

Table 4: Six indexes needed to cover all 16 access patterns.

To simplify and speed up the lookup operations on access patterns, we implement every index containing the full quads as key (i.e. (c:p:s:o) for the cp access pattern), rather than having the index on (c:p) and keep the remaining elements of a quad in linked lists in the value part. That means we store each quad (s, p, o, c) as key (s:p:o:c) in the *spoc* index, as key (p:o:c:s) in the *pocs* index, and so on.

Table 5 shows the *spoc* and *pocs* indexes for the example dataset. For example, to retrieve all information about the predicate with OID 2 (which resolves to `http://xmlns.com/foaf/0.1/name`) we perform a range query on index *pocs* from (2:min:min:min) to (2:max:max:max), which is equivalent to a prefix query for all keys that start with OID 2. The result of the operation is an iterator over all keys with the specified predicate.

(s:p:o:c)	Value	(p:o:c:s)	Value
(1:2:3:4)	-	(2:3:4:1)	-
(1:5:6:4)	-	(2:11:12:10)	-
(1:7:8:4)	-	(5:6:4:1)	-
(1:9:10:4)	-	(5:15:12:10)	-
(10:2:11:12)	-	(7:8:4:1)	-
(10:5:15:12)	-	(9:10:4:1)	-
(10:13:14:12)	-	(13:14:12:10)	-

Table 5: *spoc* and *pocs* indexes for the example. The remaining four quad indexes are constructed accordingly.

3.2.3 Occurrence Counts

A large number of applications, such as data mining, ranking, and user interface generation, require to collect statistical information about the data set. To allow these applications to quickly access basic statistical information, we are able to store occurrence counts directly in the index.

For each key that is inserted into an index, we generate additional keys denoting access patterns. We utilize the unused value field in the B+-tree index to record occurrence counts of access patterns. For each quad in an index, we construct one key that contains node OIDs only consisting of "0"s, one key that contains the first node OID together with "0"s, one that contains the first and second node OID, and one that contains the first three node OIDs.

Using our example, consider the quad (<`http://www.harth.org/andreas/#me`>, `foaf:knows`, <`http://decker.cn/stefan/`>, <`http://sw.deri.org/~aharth/foaf.rdf`>), which resolves to the key (1:9:10:4) to insert into the *spoc* index. We generate four additional keys: (0:0:0:0), (1:0:0,0), (1:9:0:0), and (1:9:10:0), and insert them into the index as key. The value part is initialized with "1" if the access pattern was not present before, otherwise the value part is incremented by one. Table 6 illustrates how the *spoc* index looks like after adding occurrence counts. Note that the value part of the key (1:9:10:4) which denotes a quad still remains empty.

(s:p:o:c)	value
(0:0:0:0)	7
(1:0:0:0)	4
(1:9:0:0)	1
(1:9:10:0)	1
(1:9:10:4)	-
(1:5:0:0)	1
…	…

Table 6: *spoc* index with occurrence counts stored in the index.

Determining the result size of an access pattern is as simple as looking up the corresponding access pattern key in the index. The value associated with that key is the number of resulting quads for a given access pattern For example, an index lookup for the key (0:0:0:0) yields the total number of quads in an index (7 given the index in Table 6). An index lookup for the key (1:0:0:0) returns the number of quads with subject <`http://www.harth.org/andreas/#me`>, a lookup for (1:9:0:0) the number of quads with that subject and predicate foaf:knows, and so on.

Since counting and storing the occurrence counts is quite expensive, adding the counts can be performed as a batch operation once the quad indexes have been constructed.

4. Query Processing

The following section describes how to perform index lookups on the indexes and how to process and execute conjunctive queries.

4.1 Index Lookups

Given our indexing structure, there are a set of lookup operations that the lexicon and the quad indexes support. Table 7 lists the atomic lookup operations. For example, a prefix lookup for <`http://xmlns.com/foaf/0.1/`> returns OID 13 as a lower bound and OID 7 as an upper bound (see Table 1 for the content of the *nodeoid* index). The *quad* operation only returns OIDs of nodes that are needed in subsequent steps in the query processing. The other operations work analogously, for details please consult Table 7.

Operation	Parameter	Index deployed	Result
getOID	node value	nodeoid	OID
getNode	OID	oidnode	node value
getPrefix	prefix of node value	nodeoid	range from lower OID to upper OID
getKeyword	keyword	inverted index	iterator over OIDs
getCountKw	keyword	inverted index	result size
getQuad	access pattern	quad indexes	iterator over list of OIDs
getCountQ	access pattern	quad indexes	result size

Table 7: Lookup operations on the index structure.

4.2 Query Plan

This section describes how to combine lookup operations from the different indexes to be able to answer queries. In the query processing stage, results from the lexicon indexes are combined with results from the quad indexes, and results from the quad indexes among each other.

The simplest case for a query is to ask for quads matching one quad pattern, which involves looking up the OIDs for node values specified in the query to OIDs on the *nodeoid* index, constructing keys with lower and upper bounds that determine the result set, and ranging over the corresponding quad index to derive the matching quads. For example, consider a query that asks for all quads with the literal "Stefan Decker" as object, (?, ?, "Stefan Decker", ?). We first look up the OID of the literal in the *nodeoid* index. The result, OID 11, is then used to construct a set of keys with lower bound (11:min:min:min) and upper bound (11:max:max:max). Next, we perform a range query over the *ocsp* quad index, and the resulting quads are then translated to node values which are returned to the user.

```
<> ql:select { ?x ?y ?z . };
   ql:where {
     { ?x ?y ?z } yars:context ?c .
     ?c yars:prefix <http://www.cnn.com/> .
   } .
```

Figure 5: N3 query to return all statements from the contexts starting with <http://www.cnn.com/>.

Consider a query to retrieve the statements associated with a context prefix, for example return all statements that originate from URIs starting with http://www.cnn.com/. Figure 5 shows the query in N3 notation.

So far, we have only considered queries that involve one particular quad pattern. More complex queries consist of multiple quad patterns that are connected via logical AND (conjunctive queries), which are evaluated based on joins. To be able to efficiently answer conjunctive queries, we have to perform a query optimization step which involves reordering of quad patterns. In particular, the query execution step should start with the quad pattern that yields the smallest result set, which is then subsequently combined with other quad patterns that were specified in the query.

It is common to use heuristics to estimate the size of a result set. One example of a heuristics is to estimate the result size of an access pattern based on the number of variables specified in that access pattern. Rather than relying on heuristics, we can use the *getCountQ* operation that accesses occurrence counts in the quad indexes to accurately determine the result size of a given access pattern.

The result of the plan evaluation is a list of variable bindings with OIDs.

4.3 Result Construction

Our query language has closure; that means, results of a query are itself expressed in RDF and can be therefore fed into another query. The `ql:select` clause of a query specifies a template that is used to construct the results. One type of template can be a set of RDF lists of variable bindings that constitute the answer to a query. Another type of template can include RDF statements that contain variables which are bound during query execution. The result of the query processing is then a set of RDF statements with values filled in for the corresponding variables.

If no select clause is specified in the query, then a set of lists with variable bindings is returned as an answer to the query.

5. Architecture and Implementation

The goal of our implementation was to build a system that offers scalable query facilities suited for aggregated RDF data collected from the Web while providing a lightweight implementation that can be used in other systems that need to store and retrieve RDF data (which is almost every Semantic Web application). The prototype system is available for download [5] and is released under a BSD-style license.

5.1 Components

The basic architecture of the YARS system is depicted in Figure 6. YARS consists of an HTTP access interface, a storage component that handles both persistent and in-memory indexes, and a query handler to perform query processing and evaluation.

The system is implemented in Java as a Web application that runs inside a Tomcat application server and uses Jena's N3 parser to parse incoming data and queries. The size of the resulting war file including all packages is around 250k, making YARS applicable for applications where a small footprint is required. Excluding test cases and external components, the system currently consists of around 6000 lines of Java code.

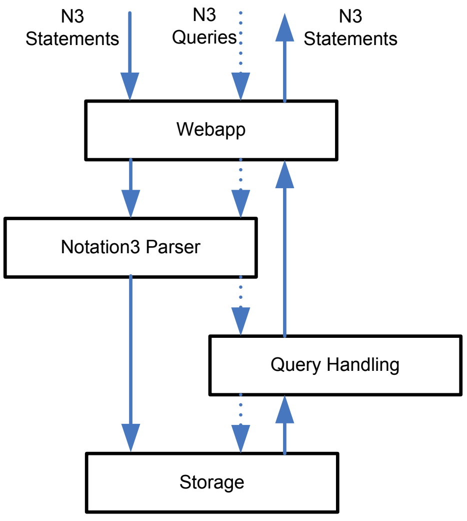

Figure 6: YARS components and data flow for N3 queries and N3 statements

5.2 Access Interface

We defined an HTTP access interface with operations for insertion (HTTP PUT), querying (HTTP GET), and deletion (HTTP DELETE). We chose to use HTTP as a network interface, which makes it possible to use standard HTTP browsers for HTTP GET operations or versatile HTTP clients such as curl[6] or wget to interface with YARS. The data format for input is N3, and results are returned in N-Triples format. We support N3 as a result format, because, given our index structure, we can return results in a streaming fashion without the need to construct the result set in memory first.

5.3 Storage

To be able to implement the index organization we chose to use JDBM [7], a lightweight open-source library that includes implementation of B+-trees for persistent storage of data on disk. JDBM consist of a record manager that offers caching and transactions. Both *oidnode* and *nodeoid* indexes are candidates for high number of buffers because lookups in these indexes are very frequent.

We chose to use a pre-existing B-tree implementation over developing our own B-tree for similar reasons as [15].

The additional layer we had to implement was the handling of concatenated keys in the quad indexes. The text inverted index is constructed and held in memory for better access performance. Both insert and delete operations are transactions. Indexes can be created as batch operations based on the *spoc* index.

6. Experimental Results

We conducted a performance evaluation based on a synthetic dataset from the Lehigh University Benchmark [9] containing 2.8 million triples for our experiments. We created a dataset, univ(20, 0), which file size in N-Triples format is 392 MB.

We considered the following RDF stores for evaluation: Sesame, Kowari, Redland and Jena2. [9] shows that Sesame generally supersedes Jena in performance results, therefore we did not include it in the benchmarking. We tried to install Kowari 1.0.2, but failed to get a running version. Therefore we concentrated in our efforts on Sesame 1.1RC2 and Redland 0.9.18.

As a test server, we used a Pentium-4 2.4 GHz machine with 4 GB main memory running Debian Sarge. YARS and Sesame were running inside a Tomcat servlet container with a VM memory size set to 1GB for index construction and 128 MB for retrieval tests. Both clients for YARS and Sesame were programmed in Java. In contrast to Sesame and YARS, Redland does not provide an HTTP access interface. Therefore we used the Perl API to add data to the repository and perform queries.

[5] `http://sw.deri.org/2004/06/yars/yars.html`

[6] `http://curl.haxx.se/`

[7] `http://jdbm.sourceforge.net/`

Sesame lacks context support; we refrained from using reification in Sesame to keep context information and just stored triples. The context mechanism was enabled in the Redland store.

6.1 Index Construction

For the index construction test, we loaded the files from N-Triples format into the repository. The index creation time for YARS consists of creating the quad index excluding statistical information, and the lexicon indexes without inverted index to be comparable to the other repositories that do not construct these indexes either. Figure 7 shows the performance measurements from the index construction process.

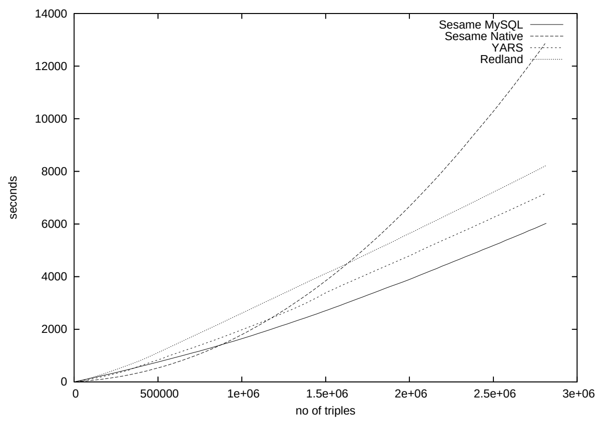

Figure 7: Index construction times for synthetic dataset

To be able to discuss the results, we first briefly introduce the indexing methods of the various data stores. Redland stores three indexes, based on hash tables. The po2s index maps a key on p and o to value s, the so2p index maps s and o to p, and the sp2o index maps s and p to o. The native store of Sesame has been recently included into the Sesame distribution. Internally, resources and literals are mapped to OIDs, and then separate indexes are kept on s, p, o. For queries specifying both s and p, an expensive join has to be performed. YARS keeps the complete index on quads without statistics as well as the *nodeoid* and *oidnode* indexes. Although we are keeping full indexes and have a sophisticated index structure, index construction times for YARS are comparable with the other systems.

YARS trades index space for query time. As Table 8 shows, YARS requires more space than the Sesame implementations. However, Sesame does not include the notion of context, which means that the index contains less information. In a triple store, triples that occur in multiple files/contexts are just stored once. In a quad store, those triples are stored as many times as they occur in different contexts. Please note that we keep 8 byte OIDs instead of Sesame's 4 byte OIDs, which roughly doubles our index size.

System	Index Size (bytes)
Redland	2.164.019.200
Sesame MySQL	340.381.636
Sesame native	39.997.992
YARS	1.090.002.944

Table 8: Index size for the synthetic Univ20 dataset.

6.2 Queries

Since the queries associated with the Lehigh benchmark take into account reasoning, we created four basic queries that test different access patterns and have different characteristics. Please note that we perform the query experiments on plain RDF semantics and do therefore not take any reasoning into account.

No	Query
1	?x rdf:type univ:UndergraduateStudent
2	?x ?p "UndergraduateStudent0"
3	<http://www.University965.edu> ?p ?o
4	?x univ:worksFor ?y

Table 9: Quad queries used in the evaluation.

Each query was executed against the repository after a random 300 MB file was copied to hard disk to flush buffers. We issued each query ten times, but only included the first result here since we wanted to test index lookup time and not the cache manager of the persistence layer.

Query	Redland	Sesame MySQL	Sesame Native	YARS
1	0:10.48	0:18.87	1:05.16	0:18.41
2	0:44.14	0:00.73	0:00.55	0:00.49
3	0:44.15	0:00.46	0:00.47	0:00.32
4	3:04.21	0:03.42	0:01.95	0:00.47

Table 10: Performance results for quad queries.

The results obtained in the query tests reflect the internal index structures of the various repositories. In query 1, Redland is very fast since the query is only a hash index lookup on the po2s index. Sesame/MySQL performs quite well here, probably due to extensive optimizations in MySQL. Sesame/Native needs to join results from the s index with the p index and is therefore slow returning results.

YARS performs just an index lookup and streams back the results. The result size for query 1 is around 160.000 triples.

Queries 2 to 4 are returning smaller result sets, usually only a few triples/quads. Here as well, the performance results reflect the index organization of the store. Since YARS keeps a complete index on quads, all quad queries can be mapped to simple index lookup operations.

YARS has some overhead for resolving the dependencies and order in the different indices as shown by the first query. However, as soon as in the other stores multiple indexes or table scans are involved, YARS shows a better performance. For query 4 YARS was 400x faster than Redland, and still 4x to 7x faster than the available Sesame implementations.

7. Related Work

[16] presents a path indexing schema for distributed RDF repositories for the Sesame system, but does not discuss how to improve retrieval performance for local storage. Their indexing schema can be combined with ours to enable distributed indexes.

[4] presents indexing techniques for object-oriented databases. These techniques are focussing on path expressions in combination with object-oriented primitives like inheritance and classes. Our index can be interpreted as an index for path length 1, but also allows multi-directional queries, e.g. our index also allows to query for properties pointing to certain objects. As an additional extension our index structure supports a context mechanism.

Lore [14] has an index scheme for semistructured data. However, in their data model a root node always exists, whereas RDF is just a directed labelled graph without any further requirements. Lore uses various path indexes which do not cover all access patterns, as opposed to our complete index.

The query facility we offer is in the tradition of RDF stores such as Jena and Sesame. These RDF repositories store their data in a relational database, and offer limited reasoning capabilities. In contrast, we focus on fast storage and retrieval only and describe indexing techniques based on multi-dimensional access methods that are B-Tree based, similar to [11]. Multi-dimensional indexing methods such as R-Trees and space-filling curves are not entirely suited for our problem because we often have queries along one particular dimension.

8. Conclusion

Query processing for RDF is an important issue for Semantic Web applications and we have determined a set of indexes required for efficient RDF query processing. In comparison with many other RDF indexing approaches that index only for a restricted set of access patters, our approach provides indexes for all access patterns on RDF with context. We utilize indexes on string representations and on the RDF graph to efficiently answer queries that are characteristic for data from the Web.

Acknowledgements

Thanks to Thomas Fanghänel for suggesting the use of concatenated keys, and to Carlos F. Enguix for discussion and comments.

This work is supported by Science Foundation Ireland (SFI) under the DERI-Lion project (SFI/02/CE1/l131) and by the European Union's IST programme under the DIP project (FP6-507483).

References

[1] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The rdfsuite: Managing voluminous rdf description bases. In *Proceedings of the 2nd International Workshop on the Semantic Web (SemWeb'01), in conjunction with WWW10, Hongkong*, May 2001.

[2] D. Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[3] T. Berners-Lee. Notation 3 – Ideas about Web architecture. http://www.w3.org/DesignIssues/Notation3.html.

[4] E. Bertino. An Indexing Technique for Object-Oriented Databases. In *Proceedings of the 7th International Conference on Data Engineering, Kobe*, pages 160–170. IEEE Computer Society, Apr. 1991.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proceedings of the 2nd International Semantic Web Conference, Sardinia*, pages 54–68. Springer, 2002.

[6] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11:121–137, 1979.

[7] R. Guha. rdfDB : An RDF Database. http://www.guha.com/rdfdb/.

[8] R. V. Guha, R. McCool, and R. Fikes. Contexts for the Semantic Web. In *Proceedings of the 3rd International Semantic Web Conference, Hiroshima*, Nov. 2004.

[9] Y. Guo, Z. Pan, and J. Heflin. An Evaluation of Knowledge Base Systems for Large OWL Datasets. In *Proceedings of the 3rd International Semantic Web Conference, Hiroshima*, pages 274–288. LNCS 3298, Springer, 2004.

[10] P. Hayes. RDF Semantics. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/rdf-mt/.

[11] H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai. Efficient Search of Multi-Dimensional B-Trees. In *Proceedings of the 21th International Conference on Very Large Data Bases, Zurich*, pages 710–719. Morgan Kaufmann, Sept. 1995.

[12] R. M. MacGregor and I.-Y. Ko. Representing Contextualized Data using Semantic Web Tools. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida*, Oct. 2003.

[13] F. Manola and E. Miller. RDF Primer. W3C Recommendation, Feb. 2004. http://www.w3.org/TR/rdf-primer/.

[14] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, 1997.

[15] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-Text Index for the Web. In *Proceedings of the 10th International World Wide Web Conference, Hong Kong*, pages 396–406, 2001.

[16] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of 13th International World Wide Web Conference, New York*, pages 631–639, May 2004.

[17] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of SWDB'03, 1st International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Berlin*, pages 131–150, Sept. 2003.