Christian Schönberg, Burkhard Freitag: Evaluating RDF Querying Frameworks for Document Metadata.
2009. Technical report, Number MIP-0903, University of Passau

Evaluating RDF Querying Frameworks for Document Metadata

Christian Schönberg and Burkhard Freitag

Department of Informatics and Mathematics, University of Passau
{Christian.Schoenberg, Burkhard.Freitag}@uni-passau.de

Technical Report, Number MIP-0903
Department of Informatics and Mathematics
University of Passau, Germany
February 2009

Abstract

An evaluation of different RDF querying methods in the context of document metadata is presented. Document metadata differs from general ontology data as it often has a more tree-like structure with linear cross-connections. We investigate how well current RDF frameworks can deal with this particular kind of RDF graphs. For comparison, we use a straight forward XML serialisation mechanism and standard path-based technologies like XQuery.

1 Introduction

Ontology querying, in particular RDF querying, is an integral part of current knowledge engineering. While ontologies are well covered by research, they are still not as widely used as expected [BT06]. Because ontology creation is only part of the work, and since ontology use has not yet had a complete breakthrough even for domains where many ontologies are available, we suspect that improved querying support could contribute.

Background of the research presented in this report is the *Verdikt*[1] project which aims at creating an integrated framework for checking document consistency [WJF09, Ver09]. The framework contains an information extraction component that can extract the document structure, narrative paths and other relevant metadata such as fragment type, fragment role, and topics addressed. Input documents may vary over a rather wide range of document formats. As extracting data from a document is a time consuming task, it is useful to store the document metadata in the form of RDF data. RDF querying can then provide the data to form a customised abstract document model for further use in the consistency checking framework.

Usually, RDF data forms a general graph. However, in the application domain of document management one often has a more specific metadata structure which can be exploited for optimization purposes. Basically, we can assume that metadata form a tree with linear cross references (cf. figures 3 and 4).
The contribution of this report is an evaluation of how this a-priori knowledge about metadata structure affects the performance of general graph-based RDF querying mechanisms, and how well these query languages reflect domain related requirements. We use XQuery for comparison, running on a serialised XML view of the test datasets. Since XML querying is generally optimised for tree structures, this will provide us with an interesting benchmark.

In [HBEV04], a number of RDF query languages have been evaluated in terms of expressiveness. However, there have been several new language developments since then, first of all the W3C standard SPARQL [SPQ08] and some significant changes to SeRQL [SeR08], which motivated the work reported on in this report. Furthermore, we present performance measurements particularly concerning tree-shaped metadata, which have apparently not been analysed before. Some other related work should be mentioned, in particular [FBB+06, FLB+06], which provide a more general classification of querying languages and [MKA+02], which contains an overview of early ontology frameworks.

The remainder of this report is organised as follows: Section 2 outlines the specific requirements of document metadata. Section 3 gives an overview of current RDF query languages and frameworks. In section 4, we describe the evaluation setup, while section 5 presents the preformance evaluation. The report concludes with a short summary.

[1]This work is part of a project funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under grant number FR 1021/7-1.

2 Requirements

Handling document metadata in the sense described above requires specific properties of both query languages and storage systems. In some instances, this results in less stringent demands than for general metadata. In other instances, the demands are actually higher than for the average case. Also, document metadata characteristics can offer a potential for optimisation not available in other cases. The specific requirements and their impact are discussed in sectsions 2.3 and 2.4. Section 2.1 provides a short overview of the document processing flow, and section 2.2 illustrates the RDF document model that is used.

2.1 Document Processing

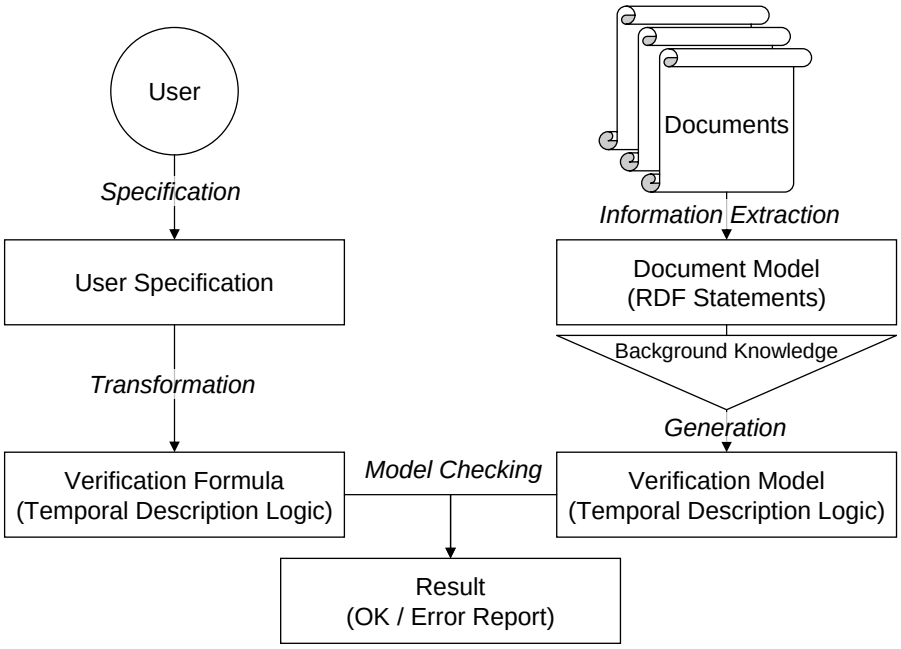

Figure 1: Verdikt overview (simplified)

A simplified overview of the Verdikt process for checking consistency criteria on documents is presented in figure 1 (see [WJF09] or [Ver09] for more details). The user (top left corner) uses some high-level specification technique to formulate consistency criteria for documents [JF08]. These criteria are converted into temporal description logical formulae (bottom left). The documents selected for consistency checking (top right corner) are converted into a generic RDF-based document model (see section 2.2). Combined with background knowledge, a

verification model is generated from the document model (bottom right). Both the formulae and the verification model are then processed by the model checker, which determines the validity of the formulae and generates an error report for any violations of consistency criteria.

The document processing component of the Verdikt framework is split into three parts:

(a) Information Extraction. Relevant metadata is extracted from the source documents by means of information extraction techniques. The documents are preprocessed and converted into valid XML code, e.g. from HTML, LaTeX, Microsoft Word, or other source formats. Structural metadata is identified using code analysis, stylesheet data (CSS), or keyword analysis. Content information is found by similar means, bolstered by dictionaries and thesauri, as well as grammar rules for word form recognition. Background knowledge, e.g. about relevant terms, can be very helpful at this stage.
The extracted metadata information is converted into RDF statements for further processing.

(b) Database Storage. Of course, metadata collections can be rather large. In some cases persistent storage is required. This means that RDF statements need to be stored in a database system.

(c) Verification Model Generation. The internal verication model is generated from the RDF metadata (see figure 1, bottom right). To this end a query language is required. Not only the document metadata but also the available background knowledge can be taken into account. Background knowledge may be used to adapt metadata-queries to the particular domain or the specific context of the document at hand. Background knowledge can also explicitly contribute to the document model, or help to infer new statements or relationships that have an impact on the verication model.

A more detailed view of the document modelling process is shown in figure 2. The information extraction returns a list of RDF statements (top right corner), from which RDF XML code is generated. This RDF code has to be loaded into the RDF framework that holds the document model (centre). The detour over the XML code is due to the limitations of the RDF frameworks, as discussed in sections 3.2.1 and 3.2.2. Following the load step, RDF XML code enriched with background knowledge is exported from the document model, from which the verification model is extracted using XQuery. Using the XML code is necessary due to the limitations of the RDF querying languages, as described in section 3.1. Since there is no unique way of generating RDF XML code from an RDF graph, the XQuery program has to be adapted to each framework and can only work with RDF XML code generated in the particular way of that framework. A method to generate well-defined XML code from RDF is described in section 3.2.3.

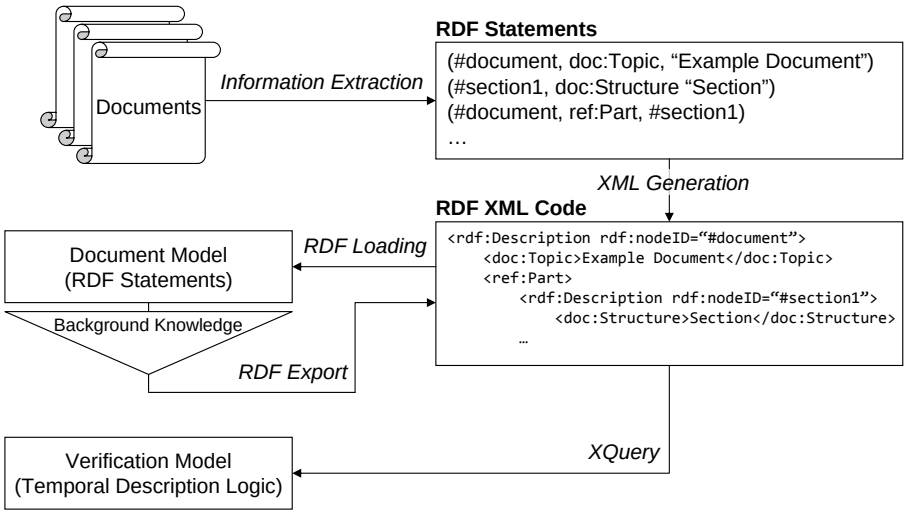

Figure 2: Detailed view of the document model component

2.2 Document Model

The Verdikt document model is represented by a collection of statements in RDF form, i.e., as a collection of $(Subject, Predicate, Object)$ triples. Figure 3 shows an example document, while figure 4 presents the corresponding RDF graph, i.e., the document model, derived from this document. The example document has a title “Example Document”, the question “Is this really a SubTopic?” as a subtitle, and two sections, the first of which contains two paragraphs.

The document model uses a custom RDF vocabulary. Resources representing structure objects like corpus, document, chapter or section are annotated with RDF statements such as $(\#document, doc{:}Structure, \text{“}Document\text{”})$, having `doc:Structure` predicates and target “Corpus”, “Document”, “Chapter” or “Section” literals, respectively. Additional information on the nesting level of sections can be expressed by the predicate `doc:Level`. Resources representing fragment type objects like example, list, question or others are annotated with statements having `doc:Fragment` predicates and target literals like “Example” or “List” (not shown in the example document). Structural hierarchies (e.g. relations between sections and subsections) are modelled using `ref:Part` predicates, while cross references are modelled using `ref:Reference` predicates, and references concerning the document order are represented by `ref:Successor` and `ref:Predecessor` predicates. Topics and subtopics are described by `doc:Topic` and `doc:SubTopic` predicates, respectively. Initial sections or paragraphs (in linear document structures such as DocBook) can be annotated with boolean literals and a `doc:Initial` predicate. Thus, the first text unit a reader sees can be identified (cf. *#para_1* in figure 4). Docu-

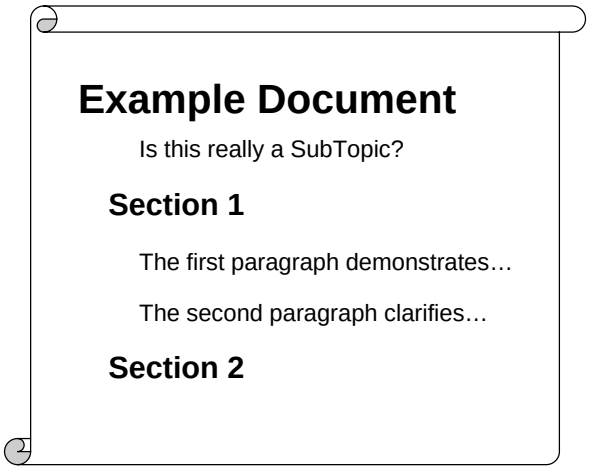

Figure 3: Example document

ments existing in several versions, e.g. a version for students and a version for teachers, can have more than one initial section, while other documents, like a collection of wikipedia articles, have no clearly defined starting point. Quality information about certain statements can be annotated with `eval:Probability` predicates and target literals having floating point values in the range of 0 to 1, as depicted in the bottom left corner of figure 4. In this case, the statement $(\#document, doc{:}SubTopic, \text{“}Is\,this\,really\,a\,SubTopic?\text{”})$ is attached a probability value of 0.3, meaning that the original statement only has about a one in three probability of being true. So “Is this really a SubTopic?” might have been mistaken for a subtopic by the information extraction component. When dealing with error-prone information extraction on large and complicated documents, this kind of quality annotation can be critical in later stages of the document verification process: Either metadata of low quality is disregarded entirely, or any conclusions drawn from it must be regarded as questionable when presented to the end user. Otherwise, correct documents can be mistakenly marked as erroneous or – worse – erroneous documents can be marked as valid.

2.3 Query Language Requirements

Figure 5 shows a list of queries often used on document metadata. The first query is a simple structural distinction. The next four queries are based on that distinction, but use it as a starting point to extract further information: The title (query (2)), a list of terms used in that particular structural unit (query (3)), an optional introductory subsection (query (4)), and all structural descendants of this unit if any exist (query (5)). The query in line (5) is more

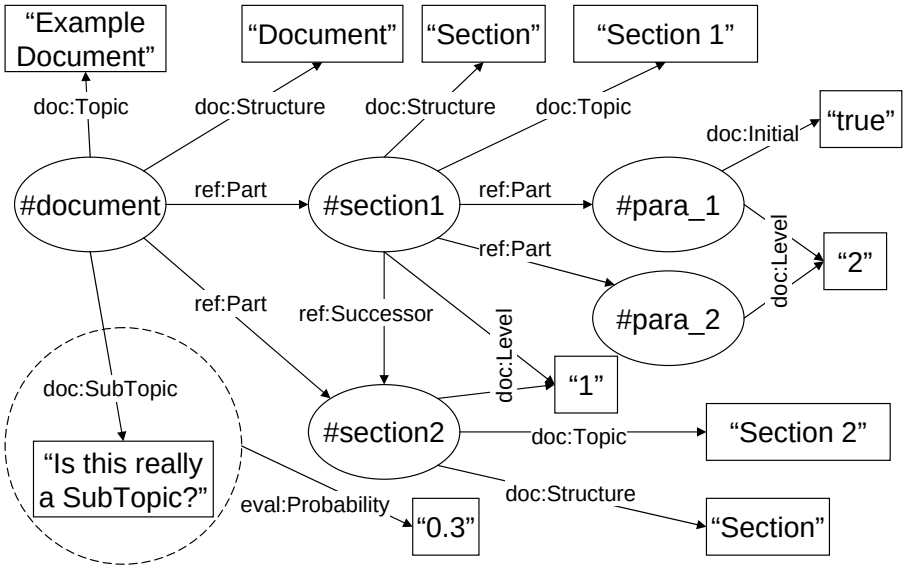

Figure 4: RDF graph of the example document

complicated than those above it. It has to return all subchapters, sections, subsections, paragraphs and other structural units that are part of the chapter selected in the first part of the query. However, since the structural depth of a document is not generally known, the exact formal query cannot be coded in advance. Instead, a recursive querying component is required that can traverse the graph and find all matches reachable from a certain node. Regular path expressions are the usual way to address this in graph-based query languages. The query in line (6) determines the structural type of each text unit. Again, regular path expressions or some other recursive querying construct are required for this query. The query in line (7) can be used as an addendum to the previous queries, limiting the result sets of these queries to data with a certain minimum

(1) Find every **chapter**.
(2) For each **chapter**, find the **title**.
(3) For each **chapter**, find a list of used **terms**.
(4) For each **chapter**, find the **introduction** (if any).
(5) For each **chapter**, find all its **descendants** (if any).
(6) For each **structural unit**, find its **type** (chapter, section, subsection, etc.).
(7) In all queries, find only metadata that has a **quality value** of **0.75** and above.

Figure 5: List of common queries on document metadata

of quality.

An important aspect of document metadata is that they are often incomplete. This can have different reasons. Documents can be fragmented or under development and therefore not be available completely. Maybe the metadata have been automatically extracted from a document (see section 2.1). Information extraction algorithms are usually incomplete which can cause some metadata to be missing. As another reason, the confidence value for some data may fall below a certain threshold causing some data to be disregarded.
RDF supports the handling of incomplete data directly by its semantics which is based on the open world assumption: It is not assumed that the negation of a statement is true just because it is not stored in the database. However, query languages need to support incomplete data as well by allowing for partial matches. This is usually accomplished through optional subqueries in the following manner: The main query is formulated as usual, while the data that may be missing is declared optional. When the query is executed, the result will then contain all full matches for the main query, combined with matches on the optional query where data is available (cf. figure 6 and section 3).

RDF statements form a graph, so the query language should reflect this particular structure. Most RDF query languages are syntactical heirs of SQL, but are adapted to be better suitable for matching triples. By combining several triple matches, it is possible to query entire subgraph structures. Some languages provide abbreviated syntax options to facilitate a more convenient mapping of graph structures to linear query strings. However, different from XML there is no generic concept of querying RDF graphs in a path-based manner.

Queries on document metadata frequently follow a specific pattern: They isolate one or a few nodes in the document tree and then focus on the subtrees below these nodes (cf. figure 5). For a single query, it is thus useful to be able to reference a specific node, which has been identified in the main query, from different subqueries. It would also be useful to be able to reference nodes identified in previous queries, so that the subtree identification does not have to be reiterated. With the `nodeID` attribute, RDF provides the necessary mechanism. The `nodeID` attribute uniquely identifies an RDF resource in the RDF metadata collection. If a node is returned as the result of a query, if should be possible to reference this particular node from other queries by referencing its `nodeID`. In practice, however, some RDF frameworks assign temporary `nodeID`s to RDF nodes returned as query results, which are unique (and valid) only in the context of this single result set.

Another query pattern often found in document queries is recursion of an unknown depth: If no upper bound for the document depth is known, then without this ability it is impossible to specify a query reaching every part of a document.

An additional requirement of query languages is the support of reification. Reification means having statements about other statements. Document metadata is frequently annotated with quality information from the information extraction (see figure 4), or with further clarification on relationships between

document fragments. This is usually expressed via reification. As an example, let A, B and C be statements. A reads "Section 2 references section 1.", B reads "Statement A has only a 5 percent probability of being true.", and C reads "The reference in A is a textual reference only.". The data from statements B and C would be lost if reified statements could not be accessed in queries.

2.4 Storage Requirements

The first requirement of an RDF storage solution for document metadata is of course to satisfy the specific query requirements (see section 2.3 above).

While pure RDF triple stores are easy to implement, a more efficient method for data retrieval is desireable [SGK+08]. Current research has shown some interesting solutions, but most are only available as prototypes if at all [AMMH07, UPS07, SJ07]. There are also questions about the actual effectiveness of these new approaches: Results seem to vary widely depending on the query [SGK+08].

Document metadata offer some potential for optimised data storage. Apart from the tree-like structure, two starting points shall be named here: First, a path-based index representing the potential paths through a document. Second, a context-based index, which takes advantage of the fact that document data is usually used in some specific combinations, but not in others. For example, title information is usually queried in combination with structural data (cf. 5, second query), but title information and data about references are rarely needed in combination.

3 RDF Querying Languages and Frameworks

In this section, we will give an overview of currently available RDF query languages and of state-of-the-art RDF frameworks that support querying. We will describe their properties and evaluate their usefulness for RDF metadata representing documents.

3.1 Languages

There are several different query languages [FBB+06, FLB+06, HBEV04, MKA+02] that can operate on RDF. The query languages RDFPath, RPath and RxPath[2] use a data model similar to XPath [Swa01, MKKS04]; however, until today no implementation is available for RDFPath and RPath, and only an outdated (early 2006) and rather limited alpha version of RxPath. These languages will not be considered here. Other query languages like TreeHugger or RDF Twig are implemented as an XSLT extension [Ste03, Wal03]; however, they have not been maintained since mid 2004 or mid 2003 respectively. Therefore, and because both are available only as prototypes, we will disregard them as well.

[2] http://rx4rdf.liminalzone.org/RxPath

Figure 6 shows an abstract graphical representation of the basic query used as an example in the following sections. It retrieves all topics of documents that are part of a larger corpus, and tries to retrieve a subtopic as well, if one exists. The result of this query, when applied to the short example document shown above in figure 3 is listed in figure 7. Figure 7 also includes a second result set that would apply if the query had been extended with a quality clause as mentioned in figure 5, query (7).

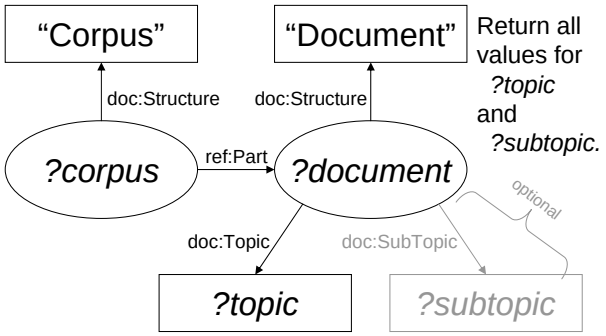

Figure 6: Graph representation of an example query

(a) Standard Query

#	Variables
1	*?topic* = "Example Document", *?subtopic* = "Is this really a SubTopic?"

(b) Query limited to metadata with probability values of 0.75 and above

#	Variables
1	*?topic* = "Example Document", *?subtopic* = { }

Figure 7: Result of the example query applied to the example document

3.1.1 SquishQL and RDQL

SquishQL is an RDF query language with a simple, SQL-like syntax [MSR02]. RDQL is based on and implements SquishQL, and is thus very similar regarding both syntax and semantics. Since RDQL is supported by the Jena Framework (see below), we will concentrate on RDQL.

In RDQL, queries consist of four parts. A `SELECT` clause, which lists all variables that should be returned; a `WHERE` clause, which holds the actual query;

an optional `AND` clause, which lists constraints on variables used in the query; and an optional `USING` clause, which declares abbreviations for URIs [Sea04].

The `WHERE` clause can consist of multiple triple patterns, each of the form `(subject predicate object)`. Each `subject`, `predicate` and `object` can be a variable (denoted by a question mark "?") or a URI, while the `object` can also be a literal. Variables can be used to identify RDF nodes common to more than one subquery, similar to a join condition in standard SQL or common variables in logic programming (e.g. the *?document* variable in the following code). The total set of all possible variable assignments conforming to the pattern defined by the query is the possible result set. The actual result set can be obtained by restricting the set of variables, similar to a projection in the SELECT clause of standard SQL. URIs can identify resources. This is used mostly for RDF vocabulary like `doc:Structure` or to identify web resources.

The following example query returns the topic of all documents that are part of a larger corpus (see figure 6):

```
SELECT ?topic
WHERE (?corpus doc:Structure ‘‘Corpus’’),
      (?corpus ref:Part ?document),
      (?document doc:Structure ‘‘Document’’),
      (?document doc:Topic ?topic)
USING doc FOR <http://.../document> ref FOR <http://.../reference>
```

The first triple pattern `(?corpus doc:Structure ‘‘Corpus’’)` matches the variable `?corpus` against all resources that are subjects of statements with the URI `doc:Structure` as the predicate and the literal `‘‘Corpus’’` as the predicate. The second triple pattern takes all matches found for `?corpus` in the first pattern and matches them against all statements having a `ref:Part` predicate and one of the `?corpus`-values just found as a subject. The object of each of these statements is bound to the variable `?document`. For the next pattern, the `?document`-values are used and further matches are found, until all possible values of the variable `?topic` have been determined. These are then returned as specified in the `SELECT` clause.

RDQL does not support inference, so to be recognised, any RDF statement has to be explicitly mentioned in the RDF data. RDQL also does not support optional patterns. This makes it impossible to deal with incomplete data in a sensible manner. If, for example, we do not only want to return the topics of documents, but also their subtopics, we would add a triple pattern `(?document doc:SubTopic ?subtopic)`, and add the variable `?subtopic` to the `SELECT` clause. However, now only documents actually having a subtopic will be regarded, while those that do not will not be included in the result. To express that subtopics should be included *if* they are available, we would like to declare the subtopic pattern to be optional, which RDQL does not allow. For querying document metadata, where the availability of specific metadata can vary from file to file, this is a serious drawback. Other language constructs such as `UNION` or `INTERSECT` operators, which could be used as a work-around, are not supported either. In principle, it would be possible to formulate two separate queries, one with and one without the subtopic clause. Lacking a `UNION`

operator, the result sets could still be combined externally. This would require, however, that the IDs of nodes remain unchanged between query results. As described below (section 3.2), this is not generally the case.

As of early 2005, the Jena Framework, while still supporting RDQL, recommends using SPARQL instead.

3.1.2 SPARQL

Since early 2008, SPARQL [SPQ08] is an official W3C recommendation. It became a candidate recommendation in 2006, but was pushed back to working draft status until mid 2007, when it became a candidate recommendation again. It can be seen as an amalgamation of several predecessors, most notably RDQL.

Basic SPARQL queries look similar to RDQL queries, with a few syntactical exceptions. Namespace abbreviations are not defined in an appended `USING` clause, but in a `PREFIX` clause at the beginning. Triple patterns are not enclosed in parentheses any more, and they are not separated by comma "`,`", but by periods "`.`". The content of the `WHERE` clause is enclosed in curly breackets, and multiple patterns can be grouped in curly brackets as well. Filters are now part of the `WHERE` clause, with the keyword `FILTER` instead of `AND`. Variables are denoted as above and have the same uses.
Optional patterns or pattern groups are indicated by the keyword `OPTIONAL`. Marking patterns as optional means that they will be part of the result set if a match is found, but no data are excluded from the result set if no match can be found. Results of patterns can be combined using the `UNION` command.
SPARQL also offers commands such as `DISTINCT`, `LIMIT` or `ORDERBY`, known from standard SQL, as well as many syntactical abbreviations.

In SPARQL, the (extended) example from above would look like this:

```
PREFIX doc: <http://.../document> PREFIX ref: <http://.../reference>
SELECT ?topic ?subtopic
WHERE { ?corpus doc:Structure ‘‘Corpus’’ .
        ?corpus ref:Part ?document .
        ?document doc:Structure ‘‘Document’’ .
        ?document doc:Topic ?topic .
        OPTIONAL { ?document doc:SubTopic ?subtopic }
}
```

Again, variables are used to specify join conditions like in RDQL (see section 3.1.1). Resolving the triple patterns works in the same way as for RDQL, except for the additional `OPTIONAL` pattern, which is simply added to the result set where available.

SPARQL does not support regular path expressions, making queries over a graph of unknown depth impossible. It is, for example, not possible to return all descendants of an object (unless one knows an upper bound of their depth and is prepared to formulate a huge query):

```
?object ref:Part ?part .
?object ref:Part ?tmp1 . ?tmp1 ref:Part ?part .
?object ref:Part ?tmp1 . ?tmp1 ref:Part ?tmp2 . ?tmp2 ref:Part ?part .
...
```

3.1.3 RQL and SeRQL

As with RDQL and SPARQL, the RQL syntax is based on SQL. The original RQL was introduced in [KAC+02]. However, the development seems to be stagnant since 2003. The Sesame Framework supports a revised and extended version of RQL, SeRQL [BK03], which we will analyse here instead. SeRQL also solves several problems of RQL, such as distinguishing between URIs and variables, literals, and a few syntax issues [Fra03].

The general capabilities of SeRQL are close to those of SPARQL, both languages having influenced each other in their development. The SeRQL syntax is quite different, though (but still similar to SQL). It has a `SELECT` clause, which lists the variables to be returned. The triple patterns are listed in the `FROM` clause, separated by comma ",". Subjects and objects of triples are enclosed in curly brackets; literals are also enclosed in quotes, URIs in angle brackets, and – in contrast – variables are not marked at all. Despite the syntactical differences, variables are used exactly the same way as in RDQL and SPARQL (sections 3.1.1 and 3.1.2). Optional patterns are enclosed in square brackets. Namespace abbreviations are declared similarly to RDQL, with a `USING NAMESPACE` command. As opposed to SPARQL, filters can be defined in the `WHERE` clause.
SeRQL supports many syntactical abbreviations, including narrowing the focus to subgraphs, as well as many advanced commands such as `LIKE`, `LIMIT`, `INTERSECT` or `UNION`, offering a broader variety than SPARQL. However, it does not support regular path expressions, thus causing the same problem as with SPARQL [HBEV04].

The example from above written in SeRQL now reads:

```
SELECT topic, subtopic
FROM {corpus} doc:Structure {''Corpus''},
     {corpus} ref:Part {document},
     {document} doc:Structure {''Document''},
     {document} doc:Topic {topic},
     [ {document} doc:SubTopic {subtopic} ]
USING NAMESPACE doc: <http://.../document>, ref: <http://.../reference>
```

Resolving the triple patterns works the same way as for SPARQL.

SeRQL also supports the construction of RDF graphs as the result of a query, but cannot insert this graph into the database or modify the original data in any way. This capability can be understood as providing a static view of part of the current data.

Despite the re-advent of SPARQL, SeRQL still remains an important rival, owing to its extended syntax and semantics, as well as to its efficient implementation.

3.2 Frameworks

Currently, few RDF frameworks have survived the development rush of 2000 through 2006 [FBB+06, MKA+02]. The two main frameworks that have been

firmly established are Jena[3], developed by Hewlett-Packard, and Sesame[4], maintained by Aduna. Both are open source frameworks and freely available. Redland [Bec01] is a set of C libraries for RDF. Since it currently supports only RDQL and parts of SPARQL, we will postpone its examination.
For the evaluation, we will compare both the Jena and the Sesame frameworks, as well as a simple XQuery applied to an XML view of the data for comparison.

3.2.1 Jena

RDF data is stored in Jena either in main memory, or in a persistent model using an RDBMS. In the database, triples are stored in a single table, only literals or URIs longer than a predefined threshold are moved to a separate table. This method reduces the number of joins neccessary for a query, but increases the database size. Property tables are used to group statements for specific properties, again increasing database size but making query evaluation more efficient [CDD+04]. [AMMH07] states that this approach is superior to a generic triple store, but also proposes another, possibly even more advantageous method. However, there is not yet a general consensus as to the optimal solution [SGK+08]. Further optimisations for indexing RDF data in relational databases are proposed in [MAYU05].

Jena provides three methods for querying RDF data. First, a Java API can be used to access the RDF graph directly. Second, the RDQL query language and third, the SPARQL query language are supported. The latter two allow only read access. Even though SPARQL queries can construct and return new RDF triples, these cannot be inserted into the existing RDF data. A serious drawback of the Java API is its lack of support for reification: Statements about other statements are impossible to represent. The only way to correctly deal with reification is to work around the API. To insert new reificated data, RDF code can be generated using some external means and then imported into the data store. For retrieval or querying, the RDF export (see section 2.1) or one of the supported query languages can be used. A workaround using proprietary context information for statements is available, but violates the RDF standard. Since RDF `nodeID`s are not maintained (or at least not published), referencing nodes accross queries is impossible without reiterating the previous query that returned the node in the first place. Jena assigns each blank RDF node a temporary ID that is valid (and unique) for the current query only. As of this moment, the authors are not aware of any efficient workaround for the missing persistence of `nodeID`s.

The current version of Jena referred to in this evaluation is Jena 2.5.6.

3.2.2 Sesame

Sesame supports three ways to store data – in main memory, or in a persistent binary storage being part of the file system (called "native store"), or in a

[3] http://jena.sourceforge.net/
[4] http://www.openrdf.org/

database system. There are actually a number of implementations for various database engines, such as PostgreSQL, MySQL or Oracle (version 9i and above). The implementation for PostgreSQL, for example, makes use of subtables, which are not part of the SQL standard. Each class or property is stored in a separate table, and the RDF subclass and subproperty relations are represented as ProstgreSQL subtable relations, thus projecting a feature of the RDF language directly onto a feature of the database. For other database management systems, a fixed schema is used with tables for classes, triples, properties, and relations between classes and properties [BKH02]. The current implementation of Sesame officially supports MySQL and Oracle, but can be used with other DBMS as well. Further optimisations for indexing RDF data are proposed in [HLQR07].

Querying a Sesame repository can be done either via a Java API or by using the SeRQL query language. Both methods support data retrieval, but only the Java API supports data modification. Unfortunately, Sesame suffers from the same detriments as Jena, namely the non-support for reification and for persistent `nodeID`s. This makes handling reified statements rather tedious and inefficient. However, the same reification workaround available for Jena can be used for Sesame as well. For `nodeID`s there does not appear to be a viable solution for Sesame either.

The current version of Sesame referred to in this evaluation is Sesame 2.2.

3.2.3 XML View

The XML view is a serialisation of the RDF graph into an XML document, using the standard names from the RDF vocabulary for naming the corresponding XML elements. As mentioned before, this is straightforward because the RDF graph is in fact very close to a tree in our particular case. As can be seen in figure 4, metadata graphs for documents tend to have a structure reminiscent of trees, except for shared literal values and for references. Cross references are usually the main reason why document metadata graphs cannot be modelled as plain trees.
Queries can be formulated in XQuery and answered using an XQuery engine. As such, this method is not a framework in itself, but rather serves for comparison.

For a given RDF graph the generation of the associated XML view follows these steps:

1. List all root objects (objects with statements going out, but none coming in).

2. For each root object do: Create an empty `object` element with an `id` attribute representing its name, URI or identifier.

3. For each outgoing statement pointing to a literal: Create a subelement using as a tag name the original predicate name, with a `data` attribute holding the literal's value, and a `format` attribute holding the literal's type.

4. For each outgoing statement pointing to a resource: Create a subelement using as a tag name the original predicate name, with an `id` attribute holding the resource's name, URI or identifier.

5. Reification: If an outgoing statement $s1$ is the subject of another statement $s2$, create a subelement `statement` as a child of the element created for $s1$. This `statement` element then gets a new subelement according to steps 3 or 4.

6. Repeat recursively from step 3, but do not follow statements that have already been serialised on the current path (thus preventing infinite recursion).

The idea underlying the XML views is that they more closely represent the tree-shaped RDF graphs of documents, thus making path-queries simpler and more efficient than in the standard W3C RDF/XML formulation. The corresponding XML view for the RDF graph in figure 4 about the example document is:

```
<object id=‘‘document’’>
    <doc:Structure data=‘‘Document’’/>
    <doc:Topic data=‘‘Example Document’’/>
    <doc:SubTopic data=‘‘Is this really a SubTopic?’’>
        <statement><eval:Probability data=‘‘0.3’’/></statement>
    </doc:SubTopic>
    <ref:Part id=‘‘section1’’>
        <doc:Structure data=‘‘Section’’/>
        <doc:Topic data=‘‘Section 1’’/>
        <doc:Level data=‘‘1’’/>
        <ref:Part id=‘‘para_1’’>
            <doc:Initial data=‘‘true’’/>
            <doc:Level data=‘‘2’’/>
        </ref:Part>
        <ref:Part id=‘‘para_2’’>
            <doc:Level data=‘‘2’’/>
        </ref:Part>
        <ref:Successor id=‘‘section2’’>
            <doc:Structure data=‘‘Section’’/>
            <doc:Topic data=‘‘Section 2’’/>
            <doc:Level data=‘‘1’’/>
        </ref:Successor>
    </ref:Part>
    <ref:Part id=‘‘section2’’>
        <doc:Structure data=‘‘Section’’/>
        <doc:Topic data=‘‘Section 2’’/>
        <doc:Level data=‘‘1’’/>
    </ref:Part>
</object>
```

Since XPath does support recursive queries of unknown depth, and because the implementation of the XML view allows both referencing of any node previously identified as well as reification, this approach does not suffer from any of the drawbacks described for Jena or Sesame.

The XML view has been generated using Qexo[5], the GNU Kawa implementation of XQuery, using a custom implementation of a DOM tree to generate the actual view.

4 Evaluation Setup

The evaluation, which uses five different RDF datasets and six different queries, has been executed on an Intel Core 2 Duo machine running at 2.13 GHz, with 2 GB of RAM, under Windows XP SP2.

4.1 Data

VDK1108 The first dataset is a representation of a technical documentation which has been created artificially but modelled realistically, closely resembling a real documentation of a real product. It describes the use of an imaginary HDTV satellite receiver "VDK1108". The document consists of 24 chapters on 88 pages. The RDF metadata contains a total of 297 resources and 650 statements, including 75 `ref:Part` statements.

DocBook The second dataset is a document in DocBook[6] format, the "Bash Guide for Beginners" Version 1.9 by Machtelt Garrels, available under the GNU Free Documentation License, Version 1.1. The document consists of 17 chapters on 173 pages. The RDF metadata contains a total of 1.987 resources and 6.679 statements, including 1.585 `ref:Part` statements.

DITA The third dataset is an artificially created document in DITA[7] format. It consists of 2.000 DITA Topics which are linked both in a linear fashion and in a more complex pattern of back- and forward links, including cycles. The RDF metadata contains a total of 7.484 resources and 17.819 statements, including 4.605 `ref:Part` statements.

Wiki10 The forth dataset is a snapshot of 10 English Wikipedia[8] articles focused on human anatomy. The articles are linked to each other and contain various sections and subsections. The RDF metadata contains a total of 1.527 resources and 3.059 statements, including 752 `ref:Part` statements.

Wiki100 The fifth dataset is a larger snapshot of 100 English Wikipedia articles, again focused on human anatomy. The RDF metadata contains a total of 13.166 resources and 29.431 statements, including 7.231 `ref:Part` statements.

For single documents, our test datasets contain realistic amounts of metadata. Each dataset was subjected to our information extraction process, and the resulting metadata was written to an XML RDF file.

[5] http://www.gnu.org/software/qexo/
[6] http://www.oasis-open.org/docbook/
[7] http://www.oasis-open.org/committees/dita/
[8] http://en.wikipedia.org

4.2 Queries

All test queries are formulated in such a way that their answers have the form of an XML representation of an abstract model of the original document for further use in the course of document consistency checking. This does not, however, lower the generality of the test queries.

Query 1 The first query simply returns the resource representing the entire data corpus.

Query 2 The second query returns a list of all chapters/sections of the document.

Query 3 The third query annotates this list with information about which chapter is the starting point of the document. Note that since there is no "initial" Wikipedia article, query 2 is substituted in place of query 3 for the evaluation of the Wiki10 and Wiki100 datasets.

Query 4 The fourth query adds a list of all successors and references to each chapter.

Query 5 The fifth query adds a list of all examples and other fragment types contained in each chapter.

Query 6 The sixth query adds a list of all topics dealt with in any fragment of each chapter.

4.3 Document Processing Tests

For further tests, we have also prepared a real document processing example from the Verdict project. A set of eight e-learning documents[9] is used to extract relevant data and generate document models: *ESmkA* (real-time scheduling with critical sections), *ESokA* (real-time scheduling without critical sections), *EZKom* (real-time communication), *MLP* (multilayered perceptrons), *Fuzzy* (fuzzy systems), *SN* (combinatorial circuits), *SOM* (self-organising maps) and *SW* (sequential circuits). Each model contains between 25.000 and 45.000 statements, but only a few thousand resources, limiting the memory requirements. This is due to the nature of e-learning documents, where topics are frequently repeated and occur in different contexts.

5 Experimental Results

In this section, the experimental results for the initial data processing (e.g. loading the RDF data) and for the actual query processing will be presented. Afterwards, we will run an additional evaluation on the e-learning data set, using those storage method(s) that have exhibited the best results in the previous test runs. The results of this second experiment are then shown in section 5.3.

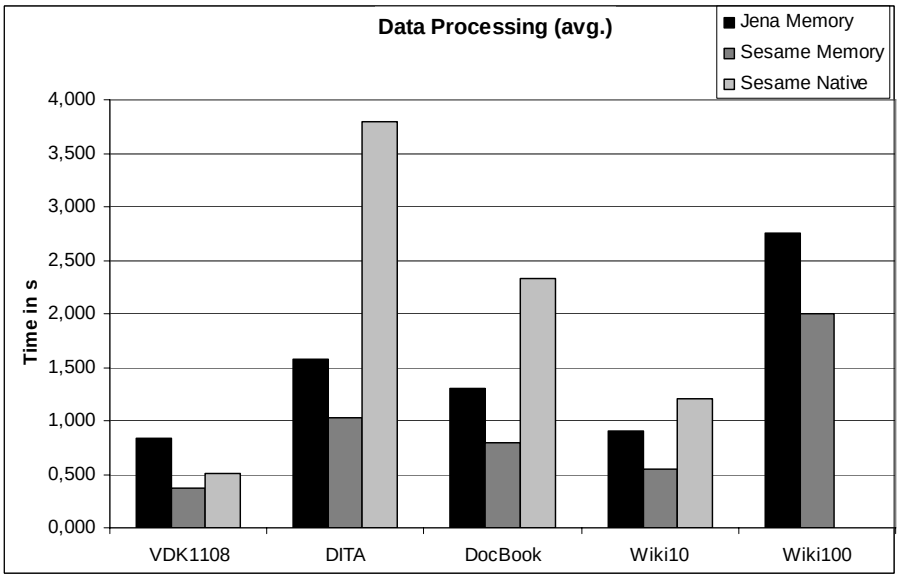

Figure 8: Data processing times: Storage in main memory and file system

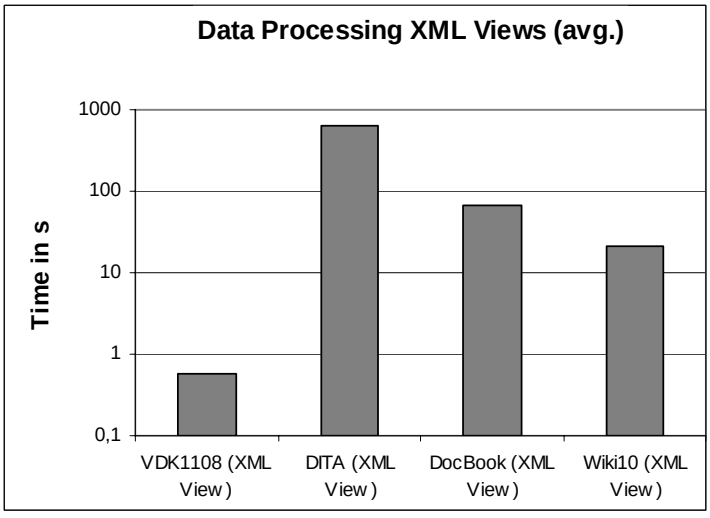

Figure 9: Data processing times: XML view

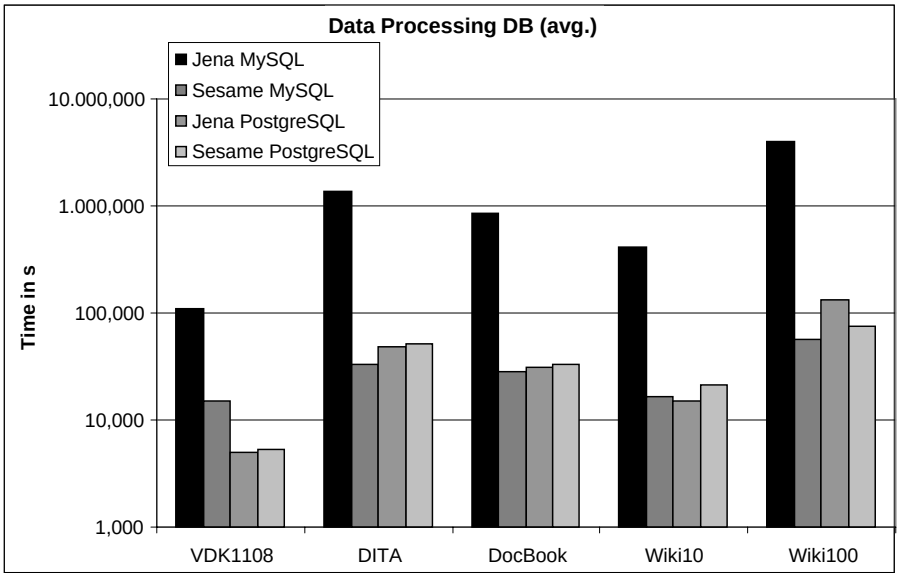

Figure 10: Data processing times: Storage in database system

5.1 Data Processing

As shown in figure 8, both the Jena and the Sesame framework create the data repository and load the RDF data in under four seconds. To provide a stable and easily replicable base for comparison, the information extraction for the five datasets has been done in a preprocessing step, and the metadata have been stored in an RDF XML file. The load time is the time required to load one of these XML files into the RDF repository (see section 2.1 and figure 2). The largest of the datasets, Wiki100, could not be loaded by the Sesame native store. It is noteworthy that the native, file-based Sesame repository, took between two and four times as long for set up as the main memory-based repository. The straightforward method creating XML views is completely outclassed for the data sets (cf. figure 9). Wiki100, for instance, could not be loaded at all using this method.

Figure 10 shows that loading times for repositories using a database for storage are much higher than for those storing their data in main memory or even the file system. The Jena framework in combination with MySQL exhibited by far the longest load times. On average, Sesame together with MySQL has been the fasted combination, followed by Sesame in combination with PostgreSQL.

[9] http://lrs2.fim.uni-passau.de/online/index.html

5.2 Query Processing

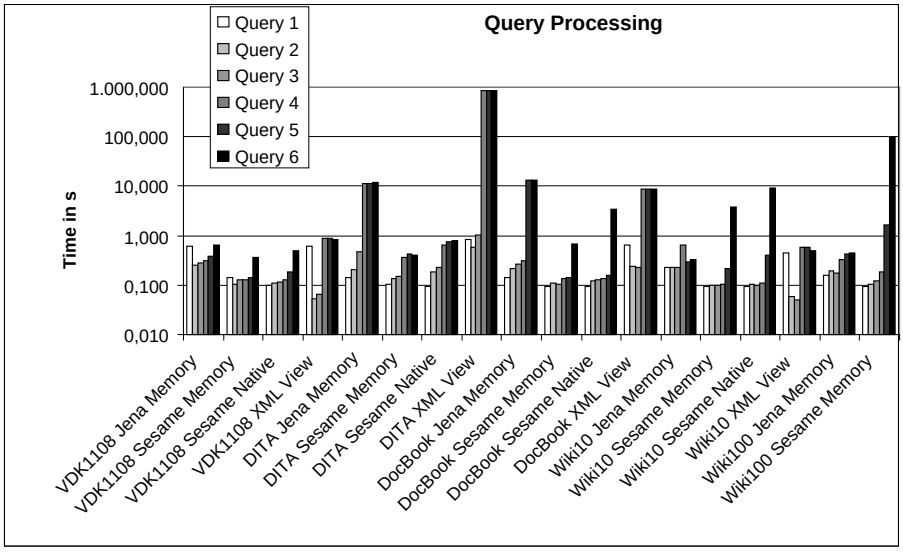

Figure 11: Query processing times: Storage in main memory and file system

Figures 11 and 12 show overviews of all query execution times. The y-axis shows the time in seconds on a logarithmic scale, while the x-axis lists all datasets in combination with each framework.

It is interesting to observe that the query processing time of the XML view method applied to the DITA dataset is much higher than when applied to the other datasets. The reason lies most likely in the very complex structure of the DITA document, leading to many references. These have to be resolved by matching IDs, which is not a strong point of XPath. Using an XQuery of comparable complexity but without dereferencing IDs took only a fraction of the run times, thus supporting our hypothesis.

As another interesting result, the query times for the DITA and DocBook datasets do not differ significantly, even though the DocBook dataset contains several times as many resources and statements. The more complex structure of the DITA dataset likely cancels out any advantage caused by its smaller size. This conjecture is supported by the fact that the DocBook dataset takes more time on the sixth query, which adds complexity at a local level, independent of the complexity of the document structure.

Again, the XML view shows an interesting effect: The execution times of the first query are higher than those of the second query, and the times of the sixth query are actually slightly lower than those of the (less complex) fifth query. This is likely due to caching of XML subgraphs and XPath results.

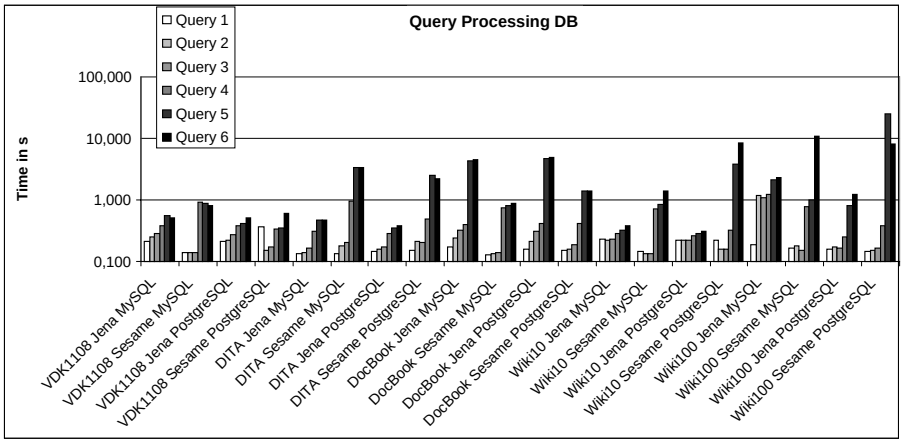

Figure 12: Query Processing times: Storage in database system

The results obtained by queries five and six on both Wiki datasets are incomplete for the Jena framework in combination with main memory-based repositories. Jena in combination with a database also returned empty results for all queries on the DITA dataset, and empty results for queries four through six on both Wiki datasets. Since the Jena query is basically the same as the Sesame query (just in SPARQL instead of SeRQL), and literally the same for the different Jena repositories, we suspect an inconsistency in the language implementation or an internal error. Given the difference in performance between Jena and Sesame, this has no bearing on the conclusions of this evaluation.

Comparing the results for the structurally simple VDK1108 dataset with those for the more complex DITA dataset shows that this structural advantage cannot be exploited effectively. It might be possible to extend current frameworks with a path index that makes use of the additional information.

The Sesame native (file-based) repository does not only take longer to initialise, but also performs worse than the main memory-based repository. This result was of course to be expected and, in combination with the native store's inability to load the largest dataset at all, confirms our assumption that for RDF graphs of document metadata, a main memory-based store might be preferrable.

According to the results of the evaluation, the Sesame main memory-based repository is recommended for querying tree-like RDF structures. For very large documents that rarely change, a database-driven Sesame store can be a good alternative.

5.3 Document Processing and RDF Generation

Summarising the results of our experiments so far, we decided to use both the Sesame RDF database and a main memory repository to perform a complete

run of the information extraction component of the Verdikt project on several relevant documents (see section 2.1). This entails the actual metadata extraction and RDF import into the respository. Since the available query languages were insufficient (cf. section 3.1) to facilitate the selection and filtering of the metadata as required for the generation of a verification model, we opted to test exporting the complete RDF data to an RDF XML file instead. Evaluation results for eight documents are shown in figures 13 and 14.

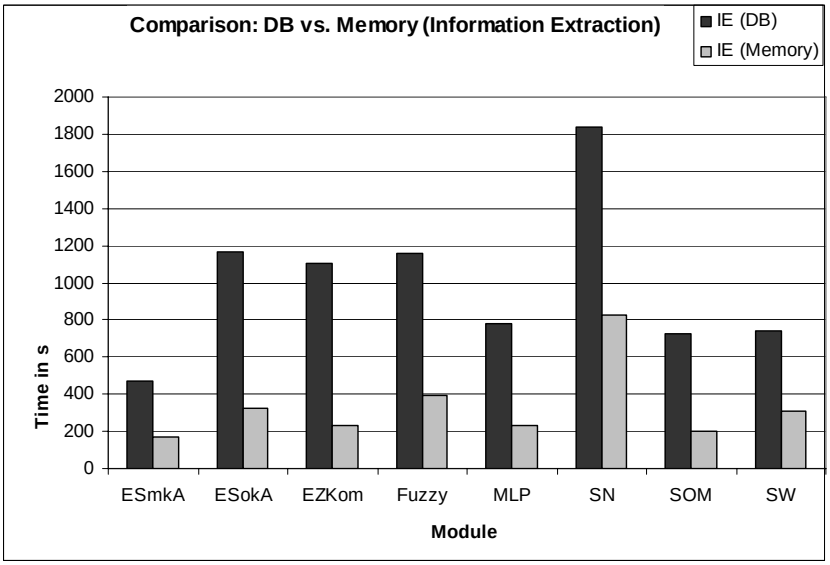

Figure 13: Information extraction: Comparison of performance for RDF database and main memory model

It is obvious and not entirely surprising that the main memory model outperforms the database repository by a considerable margin for the information extraction/data import (see figure 13). The database repository needs to reformat the statements and distribute them over the neccessary tables, while the memory model can simply keep them "as-is". Even more importantly, the database storage engine writes the data on a slower hard disc drive. This is neccessary for very large document models not fitting into main memory or when persistent storage is required, but has to be paid for in runtime performance. In absolute figures, information extraction time for the *SN* module was about thirty minutes using the database repository, and less than fifteen minutes for the main memory model. For the *EZKom* module, the values diverge even more, with about 18 minutes and just under four minutes (less than a quarter) respectively.

For RDF export (figure 14), the differences are less pronounced yet more interesting. It would have seemed likely that a database, using index structures

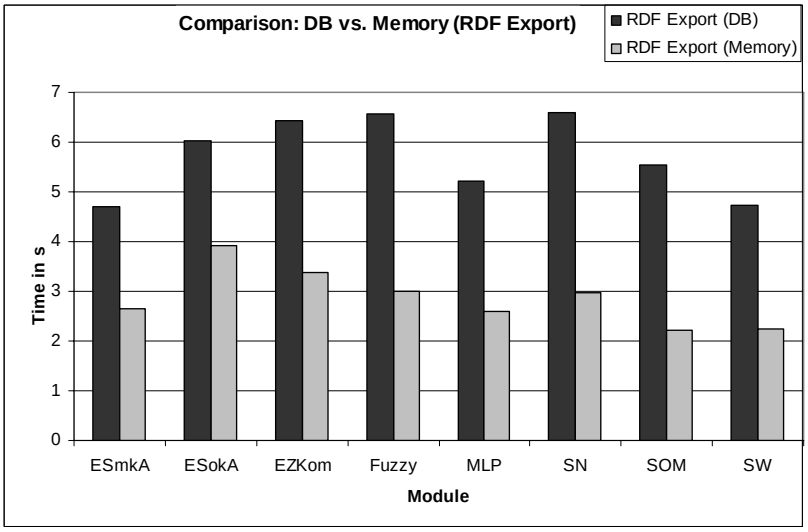

Figure 14: RDF export: Comparison of performance for RDF database and main memory model

and other optimisations, could easily outperform a main memory model. The latter has yet to generate the data representing the relationships between different statements, which are required to create an effective RDF model from its list of separate statement triples. Grouping statements by subject – as is necessary for a concise RDF model – can be carried out by a simple join operation. However, in this case, the RDF-specific optimisations of Sesame (see section 3.2) can decrease the performance, because all the predicate tables have to be joined to get all subjects. While this disadvantage could have been avoided through additional subject-centred tables, the cost in storage space and duplication effort would have been prohibitive. Apparently, the superior coherence information of the database does not compensate for the speed advantage of pure main memory data retrieval. It is also possible that the model size is still too small to exhibit possible performance advantages of the database system. If this is the case, it might be evident in a relative performance increase (database vs. main memory) for the larger models, esp. *Fuzzy* and *SN*. However, such an increase could not be observed.

These findings affirm and reemphasise the results of the previous parts of the evaluation, that a database-driven model should be used for large and static documents, while a main memory repository should be preferred whenever possible, namely for smaller document models.