The Sesame LuceneSail: RDF Queries with Full-text Search

NEPOMUK Technical Report 2008-1

Enrico Minack[1], Leo Sauermann[2], Gunnar Grimnes[2], Christiaan Fluit[3], and Jeen Broekstra[3]

[1] L3S Research Center / Leibniz Universität Hannover, 30167 Hannover, Germany, minack@L3S.de
[2] Knowledge Management Dept., DFKI, 67663 Kaiserslautern, Germany, {firstname.lastname}@dfki.de
[3] Aduna, 3817 CS Amersfoort, The Netherlands, {firstname.lastname}@aduna-software.com

Abstract. With the growth of the Semantic Web, the requirements on storing and querying RDF has become more sophisticated. When a larger amount of data has to be managed, queries in structured query languages, such as SPARQL, are not always powerful enough. Use of additional keywords for querying can further reduce the result set towards the actual relevant answers, however, SPARQL only provides complete string matching or filtering based on regular expressions, which is a very slow operation. In contrast, state of the art Information Retrieval (IR) techniques provide sophisticated features such as keyword search, lemmatisation, stemming and ranking. In this paper we present a combination of structured RDF queries and full-text search. It is implemented as an extension of an established RDF store (*Sesame*) with IR capabilities using the text search library *Lucene*, without requiring modifications to existing RDF query languages.

1 Introduction

The World Wide Web with its billions of web resources made clear the need for methods to find web resources with queries that are as simple as a hand-full of keywords. However, finding resources like that on the Semantic Web, *i.e.* in a large distributed RDF graph containing millions of RDF resources with links to other RDF resources and RDF literals, is surprisingly poorly supported by today's RDF stores and query languages.

The research community addressed this shortcoming by employing simple full-text indices for RDF literals. Though effective, this still lacks lots of features provided by classical IR systems. Unlike previous approaches which built new RDF storage systems with IR features from scratch, we show how to simply combine and benefit from well-known established systems: *Sesame* and *Lucene*.

Our full-text extension for Sesame RDF storages is called *LuceneSail*. The extension facilitates pure *Lucene* queries (IR) within pure RDF queries (structured SPARQL queries), taking full advantage of the expressiveness of each of

them. This was achieved without any modifications of existing RDF query language syntaxes. The *LuceneSail* can be re-used for many existing mature "classic" Sesame RDF storages, *e.g. File-*, *MySQL-* and *PostgreSQL*-based ones. Our principal concept can also be adopted to other existing RDF storages.

The *LuceneSail* is motivated by the requirement of including effective search into Semantic Desktop productivity tools, specifically the *NEPOMUK Project*[4] and *Aduna AutoFocus*[5]. However, the design and implementation of the *LuceneSail* is in no way particular to these projects, and the method is applicable to any application that requires full-text indexing over an RDF graph.

The rest of this paper is organised as follows. Section 2 introduces into *structured* and *full-text indexing and search*, which are the two complementary topics we are addressing here, as well as two well-known representatives — *i.e.* implementations — of each area: *Sesame* and *Lucene*. Section 3 describes our combination of both in theory, whereas Section 4 further describes implementation details. Section 5 presents a performance evaluation of our approach. In Section 6 we discuss decisions taken and further open questions, Section 7 examines related work, and Section 8 makes concluding remarks.

2 Structured and Full-text Indexing and Search

In the following section we briefly introduce the two research areas this work is addressing, *i.e.* structured search — coming from the *Database* (DB) community — and full-text indexing and search, an *Information Retrieval* (IR) topic.

2.1 Structured Indexing and Search

Structured data are any kind of data that perfectly fit into a schema, or, more visual, can be put into a table of however defined columns. Talking about the Semantic Web, we primarily focus on structured data described in RDF [1]. This basically is a directed graph where RDF resources are nodes and RDF predicates are edges. Usual RDF query languages as SPARQL [2], RDQL [3] or SeRQL [4] query a graph by defining graph patterns with wildcards and variables[6]. The result set is either a set of graph fragments that exactly match the patterns, or a set of variable bindings (values). As usual in structured data domain, results are always exactly matching the query.

Influenced by the fact that an RDF graph is by definition a composition of RDF statements (triples) [5], most RDF systems handle RDF as a set of statements. Different indices on structure (schema) and data (instance) level are employed to quickly retrieve the relevant triples that satisfy a graph pattern. Since nodes in RDF can also be RDF literals (strings), inverted indices supporting full-text search have also been employed [6] [7]. However, these indices provide only simple full-text related features or non at all. Furthermore, RDF

[4] http://nepomuk.semanticdesktop.org
[5] http://www.aduna-software.com/solutions/autofocus/overview.view
[6] Note that without wildcards the query degrades to an *exists* function.

query languages just support exact string matching, which is bulky when you do not exactly know your data or desired information. Some languages provide filter functions (*e.g.* SPARQL provides `FILTER REGEX(...)`) that evaluate regular expressions on literals in the result set. This of course is not efficient since firstly a preliminary result set is fetched from disk and secondly only a filtered sub-set of it is returned.

Concerning RDF literals, today's RDF query languages and RDF stores lack sophisticated full-text search. This is surprising since literals are giving meaning to the Semantic Web, they are the connection to the outer world, to humans. Without literals, an RDF graphs is just a set of interconnected nodes, one element out of a set of isomorphic graphs where nodes are practically name- and meaningless. Searching the Semantic Web would benefit greatly from good literal search: finding the most relevant resources regarding a desired meaning. Literal search filters out relevant resources from all the resources that match specific structure constraints. When relevant literals have been found the structure becomes important for giving more meaning to the relevant resources.

2.2 Full-text Indexing and Search

In Information Retrieval, resources (documents) are usually represented by their plain-text content, *i.e.* a long string of characters. While indexing, this string is tokenised into terms, stemmed or lemmatised (normalising the terms) and finally transformed into a term vector. Each field represent one *term frequency (TF)*, *i.e.* the number of this term's occurrences divided by the sum of all term occurrences. The TF is a measure for the relevance of a document for a given term. The term-resource relations are then indexed in an inverted index.

On query-time, a given keyword is firstly transformed into a term again via stemming or lemmatisation, and secondly the inverted index quickly provides a ranked list of documents containing that term. The TF and the powerful relevance measure $TF \times IDF$ (see Equation 1) provide valuable scoring functions to find the most relevant documents out of all matching ones.

$$\text{tfidf}_i = \frac{n_i}{\sum_k n_k} * log \frac{|D|}{|\{d : t_i \in d\}|} \tag{1}$$

Where D is the set of all documents and n_i is the frequency of the i-th term.

For Information Retrieval, this relevance measure gives relevance to the query terms, expressed by a floating point *score* between 0 and 1. In contrast, results matching a structured search are determined by binary decisions.

Besides the common keyword queries, today's IR systems usually provide phrase (a term consists of more than just one word), wildcard (* and *?*), fuzzy (finds terms that are similar to the given one), proximity (terms appear in a specified distance) and range queries (all terms alphabetically between two given terms). These queries can even be combined to more complex queries by using boolean operators. Further, terms of a query may be weighted differently to distribute importance among them.

2.3 *Sesame*: an RDF store

Our *LuceneSail* is based on *Sesame2*, the second release of an open-source framework for storage, inference and querying of RDF data, developed by the software company Aduna[7]. *Sesame2* is very flexible and can be configured to use different storage backends, including a memory based, a relational database, or a native RDF storage file format based. This flexibility is achieved through the stacking of Storage And Inference Layer objects (SAILs). Working with RDF in *Sesame* is based on a JDBC-like[8] model and all operations are done through a `SAILConnection` acquired through the SAIL objects. This connection-centred approach ensures clean handling of transactions and concurrent access to an RDF store. The `SAILConnection` objects provide methods for adding, removing and querying RDF triples, as well as transaction management.

Sesame2 supports both, the recent SPARQL standard, as well as the SeRQL standard that was used in *Sesame1*. On the SAIL level, the RDF queries are already parsed from their original format and they are represented as instances of the internal *Sesame* query model which is identical for all query languages.

2.4 *Lucene*: a Full-text Search Engine Library

Apache Lucene[9] is a pure-Java, high-performance, full-featured text search engine library. It has excellent performance characteristics, while keeping requirements of resources low. It implements all common IR features described in Section 2.2, which are

- stemming and lemmatisation
- phrase, wildcard, fuzzy, proximity and range queries
- boolean operators and term boosting

Lucene's underlying concept is a *Lucene* `Document`, which simply consists of a number of fields, each field having a string name and string value. The `Documents` are indexed by applying different strategies on each single fields. There are various options for how to store and index the values, the most relevant ones are shown in Table 1. *Lucene* employs inverted indices on each indexed field. Each retrieved `Document` provides a score value reflecting its relevance based on the *TF*×*IDF* measure. If fields are stored, previews — so called snippets — of the matching section can also be provided by *Lucene*. On query-time, the field to be queried must be specified, so it is not possible to simply query all fields at once. To work around this problem it is considered best-practise to store all text again in an indexed *all* field, which then enables quick queries over all fields at the cost of increased storage space.

[7] *Sesame2*, http://openrdf.org/

[8] The Java abstraction layer for relational database connections, *i.e.* Java DataBase Connections.

[9] Lucene, http://lucene.apache.org/

option	value	description
store	no	The value is not stored.
	yes	The value is completely stored in the index.
	compressed	The value is stored in a compressed way.
index	no	The value is not indexed and cannot be searched.
	not tokenised	The value is indexed as one single term.
	tokenised	The value is tokenised and indexed.

Table 1. Lucene options to index or/and store `Document`'s fields.

2.5 The need for a combined solution

Search on structured data allows to formulate very specific and complex queries on structure properties. The corresponding results are always a set of exact matches, rather than a ranked list. Since RDF resources typically provide a lot of text information, one should also be able to formulate complex queries on those properties. Studies have shown that users most probably start searching by file location or classification in an ontology and then use simple keyword searching [8], which gains importance if the Semantic Web shall get widely used. However, current graph query languages strongly focus on structural queries and neglect the expressiveness of textual queries. To support users both in file location (ontology) and full-text search, a combination is needed.

In IR, textual queries can contain mandatory, optional and prohibited terms. Further compared to stemming/lemmatisation, phrase, wildcard, fuzzy, proximity and range queries, IR textual queries show this advance in contrast to the exact match queries of today's graph query languages. Functions that filter the preliminary result set using regular expression are not realistic alternatives for any non-trivial amount of data.

In the following two sections we describe our approach of integrating *Lucene* queries into graph query languages in general and specifically how it was implemented in *Sesame*, enabling both, complex textual queries and structural queries on RDF graphs.

3 Combining Structured with Full-text Queries

The main goal while integrating full-text into structured queries was to maintain the syntax of the latter so that existing query language parser can be reused. The recently standardised RDF query language *SPARQL* is expandable with *extension functions*[10] that can have multiple RDF terms as input parameters and one output value. They can be used to test conditions for optional matching or to convert data from one format to another. A problem with such functions is, that they return only one value, thus we cannot return the rank of documents

[10] SPARQL Query Language for RDF - Extensible Value Testing, http://www.w3.org/TR/rdf-sparql-query/#extensionFunctions

and we will also miss other rich information from the full-text search, such as the snippets.

Another way to extend SPARQL (as any other graph query language) is to use what we have named *virtual properties*. These RDF properties form a distinct full-text query resource inside the structured query, which is actually not stored in the RDF store. Its purpose is to annotate RDF resources with full-text queries and to provide means to return various information from the full-text search, as opposed to one or no variables using extension functions. An example of such a combined query is shown in Fig. 1. The full schema for our virtual properties is shown in Appendix A.

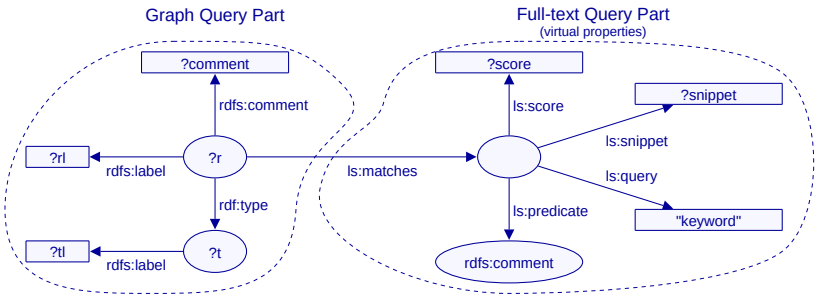

Fig. 1. An example structured and full-text query.

In a layered architecture as shown in Fig. 2, those virtual properties are extracted and corresponding actions on the full-text index are performed. The non-virtual part of the query is sent to an underlying "classical" RDF store and the results from both parts are merged into the combined result satisfying the structured and the full-text query parts.

The main advantage of this two-layer approach is clearly that existing RDF stores can be reused and transparently extended with full-text capabilities. Different query languages and existing parser can further on be used without modification.

4 Integrating *Lucene* into *Sesame*: the *LuceneSail*

We have chosen to implement our full-text search solution as a *Sesame2* SAIL. This approach has several advantages: Firstly, the *LuceneSail* will be independent from any particular underlying RDF storage, as it can be stacked on top of any existing *Sesame2* SAIL. It will work equally well on top of the *Sesame* native-store as on a relational data-base. Secondly, our *LuceneSail* will be query-format agnostic, since we only need to hook into the query model for implementing the full-text queries to *Lucene*, and *Sesame* will handle parsing of the SPARQL queries. This also has the added advantage that by letting the *Lucene* queries

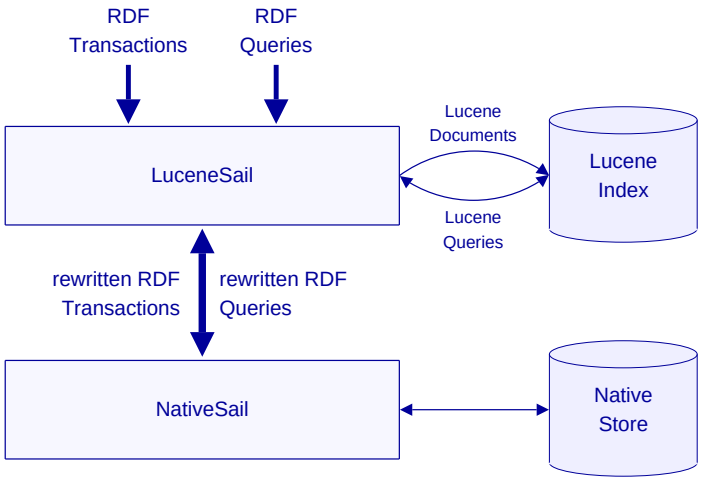

Fig. 2. Layered architecture of the full-text extension.

form a part of the query model the merging of results is handled intelligently by the existing *Sesame* query processor, and it might also optimise the queries.

The *LuceneSail* has two modes of operation within *Sesame*: the initial indexing of the RDF data and the handling of the virtual properties describing above occurring in queries. The indexing logic in the *LuceneSail* is built around *Sesame* transactions, and resources are only indexed once a transaction has been successfully committed to the underlying SAIL. Section 4.1 describes the details of how RDF triples are mapped to *Lucene* `Documents`. The second task of the *LuceneSail* is to inspect any incoming queries for use of the virtual RDF predicates associated with full-text queries. The `GraphPatterns` for the virtual properties are removed from the query model and replaced with `GraphPattern` objects that will query the *Lucene* index and return matching `Documents`, as well as score and snippets if requested.

4.1 Storing RDF in *Lucene* `Documents`

Storing RDF in *Lucene* `Documents` needs a mapping from the graph-structure of RDF to the *Lucene* `Document` format. Such a `Document` contains `Fields`, each `Field` has a name and a text-value. For full-text indexing of RDF data we identified two possible strategies for mapping RDF to *Lucene* `Documents`[11]:

- resource-based: each *Lucene* `Document` represents a single RDF resource and all its predicates.
- triple-based: each RDF triple becomes one *Lucene* `Document`.

[11] Note that there are more ways to store the complete RDF in a *Lucene* index, however, we only want to investigate scenarios where we extend an existing RDF store.

Each approach has its advantages and disadvantages. At first sight it might seem that both alternatives are equivalent, since both, storing each triple, and storing each (predicate, value) tuple for every resource looks equivalent. However, two bits of information are lost in the resource-centric approach: firstly, in modern RDF stores it is common to extend the RDF data-model with a fourth element to each triple, this is known as the context, and is used to split a large RDF store into several *named graphs*. In the triple-based approach this can be included by adding an additional field for storing the context, but for the resource-based approach this is not feasible, since there is no way to know if all triples describing one resource are all in the same context. A possible work-around for retaining the context while using the resource-based approach is to generate one *Lucene* `Document` for every context the resource appears in. However, this introduces a problem when at query time several `Document`s corresponding to the same resource may be returned. Although the *LuceneSail* can attempt to merge these `Document`s before returning the results, computing the correct relevance score for the merged `Document` is non-trivial. The second bit of information lost in the resource-centric approach is the additional meta-data about RDF Literals contained in the language and data-type tags available in RDF. Again, these can be added as additional fields when using the triple-based approach.

For the resource-based approach it is also problematic to do queries for a keyword occurring in *any* of the predicates. Since the potential number of RDF predicates (and therefore fields) can be very large it is not feasible to do a joined query over all fields, and, as detailed above, an *all* field storing all text of all predicates has to be maintained. Moreover, it is not possible to do boolean queries for the presence of a field using the resource-based method, because unlike for the triple-based method the predicate URI itself is not stored.

However, the additional representational power of the triple-based approach comes at the cost of having to index many more *Lucene* `Document`s, it is not uncommon to have RDF stores with several million triples. Furthermore, in order to query for statements having a specific predicate and a literal matching a specific keyword two indices must be queried and joined, compared to one lookup and no join in the resource-based approach. Also, querying for resources that match in multiple specific predicates cannot be merged by *Lucene* directly, as it can with the resource-based approach, and finally since all literals are indexed in the same inverted index (field "literal") there is only one global score over all literals, and no relevance score for a single predicate can be computed.

We decided to use the resource-based approach for our *LuceneSail*, despite the reduced representational power. We evaluated that the context information is not essential for implementing useful full-text search. An an early version of the *LuceneSail* was deployed in the Semantic Desktop prototype *Gnowsis* [9], the Semantic Desktop search engine *Beagle*$^{++}$[12] and in the DataWrapper-Component of the *NEPOMUK Project* and this feature was never required. The missing lan-

[12] http://beagle.l3s.de/

guage and data type tags seem to be a larger problem, but it is always possible to use the underlying RDF store to retrieve them.

Based on this resource-based approach the *LuceneSail* generates *Lucene* `Document`s with the following fields:

- The *uri* field stores the URI of a resource (stored and indexed).
- The *all* field stores all literals of a resource (only indexed).
- For each triple about this resource, where the object is a literal, store and index a field where the predicate is the field name and object is the field value.

Storing each triple again in the *Lucene* index may seem unnecessary, but it is required for supporting incremental indexing. If triples about a subject are added in two transactions the *LuceneSail* needs to update the `Document`. Since *Lucene* has no concept of update, this is done through deleting the `Document` and adding the updated version, and to allow reconstruction of the `Document` storing the fields is necessary. Of course, one could also query the underlying RDF database for this information, but this would incur a large performance penalty. An alternative and quicker approach for handling updates is to just store the new information in a separate *Lucene* `Document` and merge any multiple `Document`s at query time, however, computing the correct relevance score for a query is then non-trivial and we decided against this approach. Again, based on experience from the *Gnowsis*, *Beagle*$^{++}$ and *NEPOMUK projects* we observe that the deletion and re-adding of `Document`s does cause an unacceptable performance penalty.

Finally, if the application domain and the structure of the RDF to be stored is known it is possible to configure the *LuceneSail* to only full-text index certain RDF predicates. Conversely, it is also possible to specify predicates that should only be indexed, but not propagated to the base-SAIL. Such a predicate selection can drastically reduce the amount of storage space required by the *Lucene* index, and was for instance used in the Semantic Desktop System *Gnowsis* [9].

5 Performance Evaluation of the *LuceneSail*

In this section we present performance evaluation results of our first *LuceneSail* prototype implementation. For this evaluation, existing RDF storage and querying benchmarks such as the *Lehigh University Benchmark* (LUBM) [10] cannot be used: they do not consider complex full-text queries. This benchmark, for instance, generates artificial RDF graphs of any size. However, the RDF literals are simple one-term strings like `"Course26"` or `"Publication3Common"`. By using such graphs, keyword search would be degraded to simple complete string matching.

To our knowledge, no performance evaluations focusing on full-text have been done before. Here, we simply want to provide an estimate of the typical performance of the full-text search capabilities of the *LuceneSail*. We took a large RDF graph that primarily contains literals: the *Wordnet* (1 February 2001)

RDF graph[13]. This graph represents the relations between English nouns, verbs, adjectives and adverbs, organised into sets of synonyms, each representing one underlying lexical concept. Nodes usually have multiple literals containing one or only a few terms like `"cognition"` or `"knowledge"` and one longer multi-term literal like `"the psychological result of perception and learning and reasoning"`. Further, this graph contains a lot of structural information. Altogether, the *Wordnet* RDF graph contains over 473 000 triples, including over 273 000 triples with literal object. The corresponding NTriples file was 71 MByte in size. For storage we employed an *Sesame* RDF native store, which had a size of 52 MByte. The corresponding *Lucene* index was 47 MByte in size. For our experiments we indexed and stored all predicates.

We then issued 100 000 queries, each containing one, two or three random terms that occur in the indexed RDF literals. Each query is issued only once to avoid caching effects. As the query response time we measured the time between issuing the query and fetching *all* results. We evaluated pure full-text queries and retrieved the score and URI of the matching resources. In Fig. 3 you can see that with increasing result set size the average response time increases linearly, probably mainly caused by the linear complexity of fetching the results from disc. However, the response time is always below 50 ms for less then 1 000 results. The number of terms does not have a significant influence on the overall response time, so the performed joins must be negligible in time. Note that the results of the 2 term queries do not significantly differ from the shown results and are therefore omitted. The area between the σ- and the 2σ-lines, respectively, illustrate the deviation of the response time: it always stays at a moderate level.

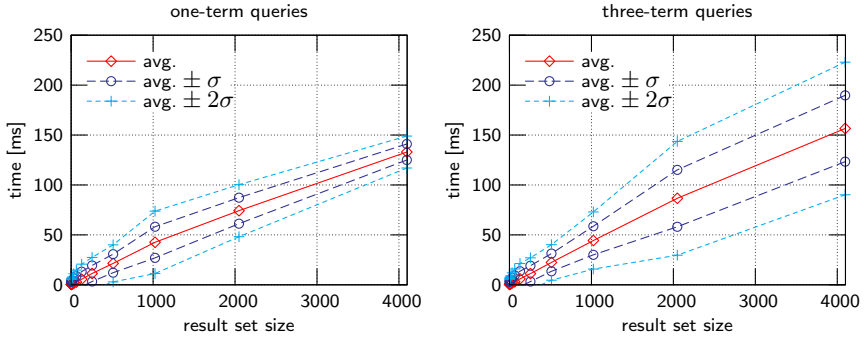

Fig. 3. Performance evaluation results.

[13] *Wordnet* RDF, http://www.semanticweb.org/library/

6 Future Work

Integrating *Lucene* into an RDF store certainly suffers from some limitations implicated by the different approaches and designs of both systems. This section highlights these issues, outlines some ideas to overcome them, which can therefore be considered as *future work*.

Our approach combines an RDF storage with a *Lucene* index, each having their own index structures. This combined system is queried with one comprehensive query (see Fig. 1). The two sub-sets of results retrieved from each query part need to be joined before all relevant resources that match the whole query are known. This join is performed on the URI of the resources which refers to the full-text query part via the `ls:matches` predicate. Unfortunately, the URI needs to be read from the *Lucene* `Document`s matching the full-text query part, which implies I/O operations. However, not all of these fetched URIs will match the graph query part and are therefore irrelevant and cause unnecessary I/O operations. In contrast, the *Lucene* `Document` IDs can be retrieved from the matching sub-set without further I/O operations, so the join could be performed on them. But they firstly — due to context information or updates — do not match with URIs one-by-one, and secondly they can change during a *Lucene* index optimisation process. An ID-to-URI mapping, updated after each insert, deletion and optimisation of the *Lucene* index would provide a much faster (due to less I/O operations) access to the URIs for joins.

Lucene does not support querying multiple or all fields at once, natively. We have implemented a `MultiFieldQueryParser` that can transparently rewrite a *Lucene* query to search over multiple fields, which implies consecutive access of several inverted indices. Alternatively, as stated above, the consequence of the best-practise approach of using an *all* field is that the information which field actually matched the query is lost. However, if *Lucene* would natively provide matching position feedback, this issue could easily be solved. Let *Lucene* `Document` D consist of N_D fields $F_i : i = 1, \ldots, N_D$. Each field has string value v_i containing l_{v_i} characters. Then, the *all* field's value would be constructed by concatenating all field values $v_{N_D+1} = v_1 \cdot v_2 \cdot \ldots \cdot v_{N_D}$ having the length $l_{v_{N_D+1}} = \sum_{i=1}^{N_D} l_{v_i}$. Considering $p : 1 \leq p \leq l_{v_{N_D+1}}$ being the position where the *all* field matches the query (first matching character position), the following function $f(p)$ provides the index of the corresponding field:

$$f(p) = \begin{cases} 1 : & p \leq l_{v_1} \\ 2 : & l_{v_1} < p \leq l_{v_1} + l_{v_2} \\ 3 : & l_{v_1} + l_{v_2} < p \leq l_{v_1} + l_{v_2} + l_{v_3} \\ & \vdots \\ N_D : & l_{v_1} + \cdots + l_{v_{N_D-1}} < p \leq l_{v_1} + \cdots + l_{v_{N_D}} \end{cases} \tag{2}$$

or

$$f(p) = \left\{ i : \sum_{j=1}^{i-1} l_{v_i} < p \leq \sum_{j=1}^{i} l_{v_i} \right. \tag{3}$$

All field length values l_{v_i} can be stored along with the `Document` in a not-index stored $(N_D + 2)$-th field. This approach can also be applied in a term-based way.

7 Related Work

In the last decade — since the Semantic Web was announced — a large number of RDF storage systems were developed. The most promising and most developed ones have already been reviewed in several surveys. A first and general survey can be found in [11], a more RDBMS-focused survey is [12]. A performance analysis of some file, *MySQL* and in-memory based RDF storages was done in [13]. However, our work focuses on incorporating full-text capabilities into an RDF storage. Probably the first store employing an inverted index on literals was the *RDFStore* [6]. Full-text search is incorporated into RDQL and their *RDFStore*-API. They maintain one inverted index for all literals, together with simple stemming (first and last character stemming). Predicate specific full-text search is achieved by joins, which are expected to be cheap since the *RDFStore* uses compressed sparse bitmaps. There is no ranking provided and only simple keyword queries can be issued. The *RDFStore* is written in Perl and C.

Yet Another RDF Store (YARS) [7] also uses one inverted index for all terms, but provides only simple queries and neither stemming nor a scoring measure. Since there is only one index for all literals, full-text queries specifying a predicate require one additional join operation. As our approach, *YARS* uses virtual properties, but without any feedback from the full-text search. *YARS* only uses queries in N3 notation, it is written in Java and runs inside a *Tomcat 5* server.

The University of Southampton developed 3store [14], an *MySQL* based triple store implemented in Core C. *MySQL* provides a full-text index for each column, allows simple keyword queries, boolean operators, phrase queries and some kind of query expansion[14], as well as a relevance measure. Thus, the *3store* gains its full-text capabilities from *MySQL*, which is therefore not general enough to be applied on other RDF stores.

An RDF storage that incorporates *Lucene* quite similarly to our work is *Kowari* [15]. When the *Lucene* support is used, all literal values in queries are interpreted as *Lucene* queries and a matching score can then be received. Due to virtual properties, in *LuceneSail* the *Lucene* queries are strictly distinguished from the exact matches. *Kowari* introduces the *Interactive Tucana Query Language (iTQL)* and implements a subset of RDQL only.

8 Conclusion and Outlook

In this paper, we have presented *LuceneSail*, a combination of structured queries (SPARQL) with information retrieval (IR). The essence of the approach was to

[14] MySQL 5.0 Reference Manual - Full-Text Searches with Query Expansion, http://dev.mysql.com/doc/refman/5.0/en/fulltext-query-expansion.html

embed queries for the full-text of RDF literals into structured queries using *virtual properties*. The results of these queries are computed by combining of an RDF indexing store (*Sesame2*) and a highly optimised full-text search engine (*Apache Lucene*). The score and text snippet results from the IR query can be returned in combination with the results from the structured query, whereas in related work the IR results are often omitted. Several optimisations were done regarding the *Lucene* document format and the query language. Due to a lack of comparable full-text search implementations and a standardised document corpus, we carried out a performance evaluation based on the *Wordnet* RDF graph to estimate the performance of our approach. Our implementation is available as free software and was used in three research projects, *Gnowsis*, *Beagle*$^{++}$ and *NEPOMUK*, as well as in the commercial software *Aduna Autofocus*.

The main contribution of this work is the RDFS vocabulary for the virtual properties used in the queries. Combined with the suggested overall architecture and document format for *Lucene* documents, this work can be replicated by others. Within 2007, we are communicating with developers of other desktop search engines to standardise this query language as best practice extension of SPARQL. The community work is part of the *NEPOMUK* EU project. We envision that future Desktop and Web Search Engines will be based on a combination of SPARQL and full-text retrieval, for which this work is a needed input.

A LuceneSail Vocabulary

```
@prefix : <http://openrdf.org/projects/contrib/lucenesail/schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

:LuceneQuery a rdfs:Class ;
      rdfs:label "A Lucene Query".

:score a rdfs:Property ;
      rdfs:label "score" ;
      rdfs:comment "The Lucene relevance score of this query" ;
      rdfs:domain :LuceneQuery.
      rdfs:range xsd:float ;

:query a rdfs:Property ;
      rdfs:label "query" ;
      rdfs:comment "The keywords to match" ;
      rdfs:domain :LuceneQuery.
      rdfs:range xsd:string ;

:snippet a rdfs:Property ;
      rdfs:label "snippet" ;
      rdfs:comment "The snippet of matching text" ;
      rdfs:domain :LuceneQuery.
      rdfs:range xsd:string ;

:matches a rdfs:Property ;
      rdfs:label "matches" ;
      rdfs:comment "Connecting the resource to the query" ;
      rdfs:domain rdf:Resource ;
      rdfs:range :LuceneQuery.
```

References

1. (W3C) Semantic Web Activity: Resource Description Framework (RDF). http://www.w3.org/RDF/ (29 January 2007)
2. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (26 March 2007)
3. Seaborne, A.: RDQL - A Query Language for RDF. http://www.w3.org/Submission/RDQL/ (9 January 2004)
4. Broekstra, J., Kampman, A.: SeRQL: An RDF Query and Transformation Language. (5 August 2004)
5. (W3C) Semantic Web Activity: RDF Primer. http://www.w3.org/TR/rdf-primer/ (10 February 2004)
6. Reggiori, A., van Gulik, D.W., Bjelogrlic, Z.: Indexing and retrieving Semantic Web resources: the RDFStore model. SWAD-Europe Workshop. http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/asemantics.html (October 2003)
7. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. 3rd Latin American Web Congress (2005)
8. Barreau, D., Nardi, B.A.: Finding and Reminding: File Organization from the Desktop. (1995)
9. Sauermann, L., Grimnes, G.A., Kiesel, M., Fluit, C., Maus, H., Heim, D., Nadeem, D., Horak, B., Dengel, A.: Semantic Desktop 2.0: The Gnowsis Experience. In: Proc. of the ISWC Conference. (November 2006) 887–900
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics **3**(2) (2005) 158–182
11. W3C: Survey of RDF/Triple Data Stores. http://www.w3.org/2001/05/rdf-ds/DataStore (April 2003)
12. Beckett, D., Grant, J.: SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/ (Jan. 2003)
13. Lee, R.: Scalability Report on Triple Store Applications. http://simile.mit.edu/reports/stores/ (July 2004)
14. Harris, S., Gibbins, N.: 3store: Efficient Bulk RDF Storage. In: Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA (2003)
15. Wood, D., Gearon, P., Adams, T.: Kowari: A Platform for Semantic Web Storage and Analysis. In: Conference Proceedings XTech 2005, Amsterdam, Netherlands (25–27 May 2005)