A lookup index for Semantic Web resources

Eyal Oren and Giovanni Tummarello

Digital Enterprise Research Institute
National University of Ireland, Galway
Galway, Ireland

Abstract. Developers of Semantic Web applications face a challenge with respect to the decentralised publication model: where to find statements about encountered resources. The "linked data" approach, which mandates that resource URIs should be de-referenced and yield metadata about the resource, helps but is only a partial solution and not followed widely. We present a simple lookup index that crawls and indexes resources on the Semantic Web. Our index allows applications to automatically retrieve sources with information about a certain resource. In contrast to more feature-rich Semantic Web search engines, our index is limited in scope and functionality and is therefore simple, small, and scalable.

1 Introduction

The Semantic Web can be seen as a large knowledge-base of statements about resources. These statements form an interconnected graph since statements may mention the same resources as other statements. A fundamental feature of the Semantic Web, as opposed to older forms of semantic networks, is that the graphs are decentralised: there is not one single knowledge-base that contains the graph of statements but instead anyone can contribute statements on his "personal" web-space. The complete graph is only visible after crawling and integrating the fragments mentioned on these personal subspaces.

This decentralised nature of the Semantic Web is well-known and, similarly to the decentralised nature of the ordinary Web, actually one of its benefits: anyone can make any statement about anything without the need for centralised control or authorisation. But for developers of Semantic Web applications, which operate on Semantic Web data, the decentralisation poses a challenge: *how and where to find statements about certain resources*?

This paper introduces Sindice, a simple lookup index that helps application developers answer exactly this question. The index crawls the Semantic Web and indexes the resources encountered in each source. A simple HTTP API returns sources containing statements about a given resource, ranked in order of relevance. In contrast to full-blown "Semantic Web search engines" we have a purposely limited scope: we only index RDF documents and we only index occurrences of resources; as a result, our code is simple and our index is relatively small.

2 Sindice architecture

Our Sindice index maintains a list of resources and the sources in which these resources appear (either as subject or as object of a statement). The basic functionality of the index is to return such a list of sources for any given URI, so as to allow the requesting application to visit these sources and fetch more information about the resource in question.

The service is provided for usage in Semantic Web applications, to implement for example a "find more information" button which would find additional sources of information through Sindice. Our lookup service augments the "linked data model" which mandates that information about an RDF resource should be available on the URI identifying the resource. Sindice promotes this approach (in its ranking algorithm) but also supports URIs that are not URLs and cannot be dereferenced (such as telephone numbers, isbn numbers or general concepts); most importantly, Sindice helps locating statements made outside the "authoritative" source for a resource.

2.1 Design requirements

The architecture for our Sindice index consists of three components for indexing, lookup, and ranking of resources. These three components provide the three methods in the Sindice API:

- `lookup(uri) => url[]`: lookup of a URI returns a ranked list of URLs in which the given URI has been mentioned,
- `index(url) => nil`: parses and indexes document at given URL,
- `refresh() => nil`: refreshes the index by re-visiting and updating all known sources.

We have two design requirements: first, we want to minimise the index size, so as to allow indexing of the whole Semantic Web without requiring complex disk solutions such as networked storage, disk-arrays or clustered machines; secondly, we want to minimise lookup times, so as to allow applications to use Sindice by default to lookup more information for any encountered resource.

To fulfil these requirements, we focus on simplicity: we only index occurrences of resources but not the actual statements in which they occur; we only allow lookups from resources to sources but not in the other direction (i.e. one cannot ask which resources are mentioned by a particular source); we use a simple ranking algorithm that produce useful results without requiring much computation time or background knowledge.

2.2 Index design

We aim for storing at least 300 million resources, mentioned on maybe 3 million sources. These numbers serve only as an indication of our scalability target and

are informed by earlier crawling results from SWSE[1] and Swoogle[2] @@ding/finin "characterising semantic web on the web". In designing the index, we optimise for disk space and lookup times. Since the only required access pattern is from resource to mentioning sources, a hashtable-like index seems natural. The key to such a hashtable would be the URI of the resource and the value of the hashtable would be the list (array) of mentioning sources.

In Listing 1.1 we describe the basic indexing algorithm, while abstracting for a moment from the exact implementation of the hashtable-like occurrence index. As shown in the listing, when indexing a new source, we extract each mentioned URI in that source and add it to the "occurrence" index to indicate that the indexed source mentions the found URI.

Listing 1.1. Indexing algorithm pseudo-code

```
def index(source)
  resources = subject_and_objects_in(source)
  for resource in resources
    occurrence[resource] += source
  end
end
```

With such a hashtable-like index, lookups are implemented by simple lookups in the hashtable and ranking of the results, as shown in Listing 1.2. Since the hashtable-like index uses the resource URI as key and the mentioning sources as values, average lookup complexity is $O(1)$. Worse-case lookups of frequently mentioned resources, especially frequent classes such as `foaf:Person` or `sioc:Forum`, would cause a higher runtime simply to read and return the large set of mentioning sources, but we do not envision these to be requested often.

Listing 1.2. Lookup algorithm pseudo-code

```
def lookup(resource)
  sources = occurrence[resource]
  return rank(sources)
end
```

2.3 Index implementation

In abstract sense our index design is clear, but we have several choices for concrete implementation of such a hashtable-like index. We have experimented with the following options:

Database: using a database table with the resource URIs as primary keys. The advantage is simplicity of implementation, the disadvantages are: that databases typically incur a slight overhead in query processing which we would not use since we have only simple key lookups, that many databases have upper limits on the amounts of rows which we could possibly reach, and most importantly, that databases typically need to have the complete

[1] `http://swse.deri.org`
[2] `http://swoogle.ubmc.edu`

list of primary keys in memory to ensure fast access to the data; in our case 300 million primary keys (resources that are each maybe 128 bytes to 500 bytes long) would imply main memory requirements of 38GB-150GB which is rather more than our standard available hardware offers.

Filesystem: implementing our own hashtable using the filesystem with one file for each resource containing the list of mentioning sources. The advantage of this solution is scalability: storing only a list of sources and compressing the contents of each file, each resource-file would occupy between 50 bytes to 300 bytes, resulting in an index size (disk-space) of 30GB which is well inside the range of current typical hardware. The disadvantage however, is that filesystems typically have a minimal block-size (configured at formatting time) which is an lower limit on occupied disk-space for each file. On for example the ext3 filesystem, typical block-size is 2K, so even if our files are only 50 bytes long, they would each occupy 2KB in diskspace; in the future we will experiment with filesystems without such limitations, such as Reiser4[3].

Persistent hashtable: using an existing library for persistent hashtables such as Berkeley DB. This solution is very similar to the second (persisting the hash-table directly onto the filesystem) except that Berkeley DB allows us to configure various parameters of the hash-table (bucket size, overflow rate, hash-function, etc.) and that Berkeley DB manages all the necessary internals to allow transaction management, concurrent access, locking etc.

After experimenting with these options and considering their characteristics, we implemented Sindice using the Berkeley DB persistent hashtable. We are still in the process of tuning it for best concurrent performance and disk space usage (e.g. bucket size and fill factor).

2.4 Source metadata

Additionally to the resource occurrence hashtable, we maintain a small hashtable with metadata for each visited source. This hashtable is used for managing the crawling and fetching process, and for the ranking (explained in the next section). To prevent over-requesting metadata sources and to follow the Robot (crawling) guidelines[4], we store visiting times and content hash for each source. We only revisit sources after a threshold waiting time and we only reparse the source's response if the content hash has changed. We do not yet use E-Tags and HTTP "last-modified" headers but consider doing so in the future.

2.5 Ranking phase

Since for popular resources our index could easily return many hundreds or thousands sources, a good ranking algorithm is crucial. But on the other hand,

[3] `http://www.namesys.com/v4/v4.html`
[4] `http://www.robotstxt.org/wc/guidelines.html`

we want to keep the index small, and thus amount of metadata minimal, we want to rank resources fast.

We have designed a ranking function that requires only little metadata for each source and is relatively fast to compute; in contrast to more optimal ranking functions such as HITS (Kleinberg, 1999), PageRank (Brin and Page, 1998), or ReconRank (Hogan *et al.*, 2006), it does not construct a global ranking of all sources but ranks sources based on their own metadata and external ranking services. We currently use the following metadata in ranking:

Hostname: we prefer sources whose hostname is the same as the resource's hostname. For example, we consider that more information on the resource `http://eyaloren.org/foaf.rdf#me` can be found at the source `http://eyaloren.org/foaf.rdf` than at an arbitrary "outside" source, such as `http://g1o.net/g1ofoaf.rdf`. We thus reuse the Internet's own authority mechanism and favour the notion of linked Semantic Web data, as advocated by e.g. Sauermann *et al.* (2007), in which resources use URIs that are resolvable and return valuable information.

Relevant statements we prefer sources that mention this resource more often than sources that mention this resource less times (considering that both sources mention the resource at all).

Source size: we prefer sources with more information than sources with little information.

External rank: we prefer sources hosted on sites with a high ranking in existing ranking services (such as Google's PageRank). We thus use techniques @@cite to estimate a ranking for each source at index time and assume that highly-ranked websites also produce valuable RDF data. For example, using an external rank the source `http://www.deri.ie/fileadmin/scripts/foaf.php?id=95` would rank higher than the source `http://g1o.net/g1ofoaf.rdf` for the same example resource `http://eyaloren.org/foaf.rdf#me`, because `http://deri.ie` has a higher estimated PageRank than `http://g1o.net`.

3 Current implementation

The current implementation is online at `http://sindice.com/`, the source code is available[5] under LGPL license.

To fetch and transform all RDF to canonical ntriples format, we use the Redland (Beckett, 2002) "rapper" utility: although we do not need to parse the RDF but only extract the resources mentioned, such extractions are difficult in XML since URIs can be mentioned without being true resources (e.g. as namespace declarations). For this reason, we employ rapper to parse all RDF into ntriples, and use a simple regular expression to extract all resource subjects and objects.

[5] `https://launchpad.net/sindice/`

To update our index, we use the pinging service `http://pingthesemanticweb.com`: we ask it periodically (currently every hour) for recently updated RDF documents and index each of these sources if they have changed since the last time we saw them. We also offer a Web interface and an HTTP API where people can submit their own RDF documents for indexing, but we prefer to reuse the `http://pingthesemanticweb.com` service since it already receives many pings from updated RDF documents.

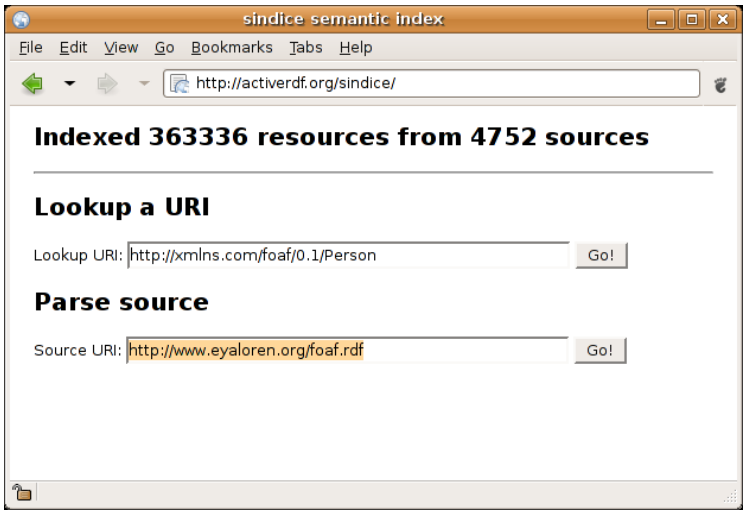

Fig. 1. Overview page of Sindice

Figure 1 shows the human-readable Web interface for Sindice. It shows some index statistics and access to the two API functions: lookup of a resource and parsing or updated source. The third API function, refreshing all sources, is not accessible over the Web interface to prevent denial-of-service attacks. Apart from the Web interface Sindice offers an HTTP API interface that returns XML, JSON, or plain text results, through which developers can lookup sources from within their programs.

Figure 2 shows the resulting lookup of a sample resource, in this case `http://www.w3.org/People/Berners-Lee/card#i`. As can be seen in the shown results, ranking was not yet taken into account at the time of the screenshot. Lookups on the current prototype take around @@@0.025s in average (variance probably due to disk cache, network lag, processor load and the amount of sources returned).

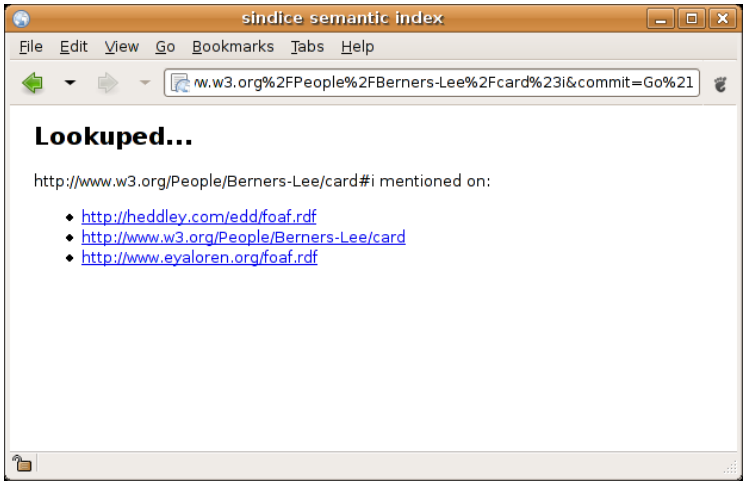

Fig. 2. Lookup of example resource

4 Related Work

We are aware of two Semantic Web search engines that index the Semantic Web by crawling RDF documents and then offer a search interface over these documents.

SWSE[6] crawls not only RDF documents but also "normal" HTML Web documents and RSS feeds and converts these to RDF (Harth *et al.*, 2007; Hogan *et al.*, 2007). SWSE stores the complete RDF found in the crawling phase and offers rich queries (expressiveness comparable to SPARQL) over this RDF data. Since SWSE also stores the provenance of all statements, it can also provide the source lookup functionality that we provide but with a cost: lookups are slower than in Sindice and the index is larger.

Similar to SWSE, Swoogle (Finin *et al.*, 2005) crawls and indexes the Semantic Web data found online. Again, the same differences apply: Swoogle offers richer functionality than we do but at a cost of index size and lookup times.

Finally, we compare our index to `http://pingthesemanticweb.com`. They maintain a list of recently updated documents, and are currently indexing over seven million RDF documents, but in contrast to our service, they do not index the statements or resources mentioned in these sources. Still, the service is very useful as companion to ours and indeed we use it to find recently updated RDF sources.

5 Conclusion

We have presented a simple lookup index for Semantic Web resources. Our index is small and scalable and allows for fast lookups. In contrast to other approaches,

[6] `http://swse.deri.org/`

our scope is purposely limited to keep the index simple and small. Future work and discussion in the workshop will be focused on improving performance and concurrency and evaluating and discussing the ranking approach.

Acknowledgements This material is based upon works supported by the Science Foundation Ireland under Grants No. SFI/02/CE1/I131 and SFI/04/BR/CS0694.

References

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

S. Brin and L. Page. Anatomy of a large-scale hypertextual web search engine. In *Proceedings of the International World-Wide Web Conference*. 1998.

T. W. Finin, L. Ding, R. Pan, A. Joshi, *et al.* Swoogle: Searching for knowledge on the semantic web. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*. 2005.

A. Harth, J. Umbrich, and S. Decker. Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2007.

A. Hogan, A. Harth, and S. Decker. Reconrank: A scalable ranking method for semantic web data with context. In *Second International Workshop on Scalable Semantic Web Knowledge Base Systems*. 2006.

A. Hogan, A. Harth, J. Umbrich, and S. Decker. Towards a scalable search and query engine for the web. In *Proceedings of the International World-Wide Web Conference*. 2007. Poster presentation.

Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46, 1999.

L. Sauermann, R. Cyganiak, and M. Völkel. Cool URIs for the semantic web. Tech. Rep. TM-07-01, DFKI, 2007.