Five Years on in SWSE: Experiences from the Semantic Web Search Engine Project

Aidan Hogan\textsuperscript{a}, Andreas Harth\textsuperscript{b}, Jürgen Umbrich\textsuperscript{a}, Sheila Kinsella\textsuperscript{a}, Axel Polleres\textsuperscript{a}, Stefan Decker\textsuperscript{a}

\textsuperscript{a}Digital Enterprise Research Institute, National University of Ireland, Galway
\textsuperscript{b}AIFB, Karlsruhe Institute of Technology, Germany

Abstract

In this paper, we discuss the architecture and implementation of a Semantic Web search engine, appropriately called the Semantic Web Search Engine (SWSE). Following traditional search engine architecture, SWSE consists of crawling, data enhancing, indexing and a user interface for search, browsing and retrieval of information; unlike traditional search engines, SWSE operates over RDF Web data—loosely also known as Linked Data—which implies unique challenges for the system design, architecture, algorithms, implementation and user interface. In particular, many challenges exist in adopting Semantic Web technologies for Web data: the unique challenges of the Web—in terms of scale, unreliability, inconsistency and noise—are largely overlooked by the current Semantic Web standards. In this paper, we detail the current SWSE system, initially detailing the architecture and later elaborating upon the function, design, implementation and performance of each individual component. In so doing, we also give an insight into how current Semantic Web standards can be tailored, in a best-effort manner, for use on Web data. Throughout, we offer evaluation and complementary argumentation to support our design choices, and also offer discussion on future directions and open research questions. Later, we also provide candid discussion relating to the difficulties currently faced in bringing such a search engine into the mainstream, and lessons learnt from roughly five years working on the Semantic Web Search Engine project.

Key words: web search, semantic search, RDF, semantic web, linked data

\textsuperscript{1} The work presented in this paper has been funded in part by Science Foundation Ireland.

Preprint submitted to Elsevier 4 February 2010
1 Introduction

Graduate students have enjoyed mixed success in the area of Web search engines. In 1998 – and armed with a couple of scientific publications and a novel ranking algorithm – graduate students Larry Page and Sergey Brin launched the renowned search engine “Google”, which was to be the beginning of an infamous and largely unprecedented success story. Entering a market full of established competitors, Google succeeded by offering users a minimalistic and uncluttered user interface, a simple keyword-based user-interaction model, fast response times, and better prioritisation of results.

Google has since become the yard-stick for web-search, servicing $\sim 64.6\%$ of the traditional web search queries and offering a deceptively simple and enviably reliable and fast service – even resourceful competitors such as Microsoft and Yahoo have thus far failed to make much impression upon Google’s market share (Yahoo [16\%], MSN/Bing [10.7\%], AOL [3.1\%] and Ask [1.7\%]).

Arguably, Google reaches the imminent limit of providing the best possible search over the HTML data it indexes. However, from the user perspective, the core Google engine (here serving as the archetype for traditional HTML search engines) is far from the consummate Web search solution: Google does not typically produce direct answers to queries, but instead typically recommends a selection of related documents from the Web. Thus, Google is not suitable for complex information gathering tasks requiring aggregation from multiple indexed documents: for such tasks, users must manually aggregate tidbits of pertinent information from various recommended sites, each site presenting information in its own formatting and using its own navigation system.

Google’s limitations are predicated on the lack of structure in HTML documents, whose machine interpretability is limited to the use of generic markup tags mainly concerned with document rendering and linking – the real content is contained in prose text which is inherently difficult for machines to interpret. Addressing this inherent problem with HTML Web data, the Semantic Web movement provides a stack of technologies for publishing machine-readable data on the Web, the core of the stack being the Resource Description Framework (RDF).

Using URIs to name things – and not just documents – RDF offers a standardised and flexible framework for publishing structured data on the Web such that data can link, incorporate, extend and re-use other RDF across the

---

1 Ireland under Grant No. SFI/08/CE/I1380 (Lion-2) and by an IRCSET postgraduate scholarship.
Web, such that heterogenous data from independent sources can be automatically integrated by software-agents, and such that the meaning of data can be well-defined using lightweight ontologies described in RDF using the RDF Schema (RDFS) and Web Ontology Language (OWL) standards.

Thanks largely to the “Linked Open Data” project – which has emphasised more pragmatic aspects of Semantic Web publishing – a rich lode of open RDF data now resides on the Web: this “Web of Data” includes content exported from, for example: Wikipedia, the BBC, the New York Times, Flickr, LastFM, scientific publishing indexes, biomedical information and governmental agencies.

Even in the formative days of this Web of Data, certain pockets of graduate students have been wide-eyed with the possible impact on Web search: assuming large-scale adoption of high-quality RDF publishing on the Web, a search engine indexing RDF could feasibly improve upon Google’s current HTML engine by offering advanced querying and browsing of structured data with search results automatically aggregated from multiple documents and rendered directly in a clean and consistent user-interface, thus reducing the manual effort required of its users. Indeed, there has been much research devoted to this topic, with various incarnations of RDF-centric web search engines emerging – Swoogle, FalconS, Watson, Sindice – and in this paper, we present the culmination of over five years research on yet another such engine: the “Semantic Web Search Engine” (SWSE)³.

Indeed, the realisation of SWSE has implied two major research challenges: the system must scale to large amounts of data, and must be tolerant to heterogenous, noisy, and possibly conflicting data collected from a large number of sources. Semantic Web standards and methodologies are not naturally applicable in such an environment; in presenting the design and implementation of SWSE, we show how standard Semantic Web approaches can be tailored to meet these two challenging requirements, often taking cues from traditional information retrieval techniques.

In this paper, we collect together the previous five years of research relating to the SWSE project, representing the current state-of-the-art in the SWSE project. As such, we present the core of a system which we demonstrate to provide scale, and which is distributed over a cluster of commodity hardware. Throughout, we focus on the unique challenges of applying standard Semantic Web techniques and methodologies, and show why the consideration of the source of data is an integral part of creating a system which must be tolerant to Web data. Also, there are many research questions still very much open with respect to the direction of the overall system, as well as improvements to

³ http://swse.deri.org/
be made in the individual components; we discuss these as they arise, giving a general road-map of past and possible future research in the area of Web search over RDF data.

More specifically, in this paper we:

- present the architecture and modus-operandi of a system for offering search and browsing over RDF web data (Section 2);
- detail the design and implementation of crawling, ranking, consolidation, reasoning, indexing, query processing and user interface components, offering pertinent evaluation and related work throughout (Section 4-10);
- present related work in RDF search engines (Section 11);
- conclude with discussion of open research challenges and current limitations of web search over RDF data (Section 12).

2 Architecture and System Overview

2.1 Application Overview

SWSE is a search engine for data: users enter a keyword query and receive a list with matches. In contrast to web search engines which used to return links to HTML documents containing the specified keywords [12], SWSE returns objects representing real-world entities. While current search engines such as Google, Bing and Yahoo return search results in different categories (“Web”, “Images”, “Videos”, “Shopping”, etc), data on the semantic web is flexibly typed and does not need to follow pre-defined categories. Returned objects can be people, companies, cities, proteins, or any other thing people care to publish data about. The rich knowledge representation enabled by standards such as RDF and OWL allow for specifying much more precise queries and returning potentially directly answers – the sought after data item itself – rather than documents which contain the piece of information couched in natural language text.

The system allows users to specify keyword queries in an input box, similar to web search engines. The results returned, however, are objects rather than documents. In subsequent steps, users can refine the result set based on facets which are taken from the current result set. For example, users start a query with the keyword “apple” to which the system replies with a set of objects matching the specified keyword: Apple (of type Company), New York (the city nicknamed “big apple”), the Macintosh computer (of type “brand of computer”), Fiona Apple (of type Artist), and so on. Users then narrow down that initial result set by type and arrive at the set of results matching their
information need.

Figure 1 shows the screenshot containing the list of objects returned as result to a keyword search.

![Screenshot of result for keyword query “tim berners lee”](image)

Fig. 1. Screenshot of result for keyword query “tim berners lee”

2.2 System Architecture

The high-level system architecture of SWSE loosely follows that of traditional HTML search engines [12]; Figure 2, details the pre-runtime architecture of our system, showing the components involved in achieving a local-index of RDF Web data amenable for search. Like traditional search engines, SWSE contains components for crawling, ranking and indexing data; however, there are also components specifically designed for handling RDF data, viz.: the consolidation component and the reasoning component. The high-level process is as follows:

- the crawler accepts a set of seed URIs and retrieves a large set of RDF data from the Web;
- the consolidation component tries to find synonymous (i.e., equivalent) identifiers in the data, and canonicalises the data according to the equivalences found;
- the ranking component performs links-based analysis over the crawled data and derives scores indicating the importance of individual elements in the data (the ranking component also considers URI redirections encountered by the crawler when performing the links-based analysis);
- the reasoning component materialises new data which is implied by the inherent semantics of the input data (the reasoning component also requires URI redirection information to evaluate the trustworthiness of sources of data);
the indexing component prepares an index which supports the information retrieval tasks required by the user interface.

We will detail the design and operation of each of the components in the following sections, but beforehand, we present the distribution framework upon which all of our components are implemented.

2.3 Distribution Abstraction

In order to scale, we deploy each of our components over a distributed framework which we now briefly describe; Figure 3 illustrates the distributed operations possible in our framework. The framework consists of one master machine which orchestrates the given tasks, and several slave machines which perform parts of the task in parallel.

The master machine can instigate the following distributed operations:

- **scatter**: partition a file into chunks given some local split function, and send the chunks to individual machines – usually only used to initialise a task and seed the slave machines with an initial set of data for processing;
- **run**: request the parallel execution of a task by the slave machines – such a task either involves processing of some local data (embarrassingly parallel), or execution of the coordinate method by the slave swarm;
- **gather**: gathers chunks of output data from the slave swarm and performs some local merge function over the data – this is usually performed to create
The master machine is intended to disseminate input data to the slave swarm, to provide the control logic required by the distributed task (commencing tasks, co-ordinating timing, ending tasks), to gather and locally perform tasks on global knowledge which the slave machines would otherwise have to replicate in parallel, and to transmit globally required knowledge. The master machine can also be used to compute the final result for a given distributed task; however, the end goal of our distributed framework is to produce a distributed index over the slave machines, thus this task is never required in our system.

The slave machines, as well as performing tasks in parallel, can perform the following distributed operation (on the behest of the master machine):

- **coordinate**: local data on each machine is partitioned according to some split function, with the chunks sent to individual machines in parallel; each machine also gathers the incoming chunks in parallel using some merge function.

The above operation allows slave machines to partition and disseminate intermediary data directly to other slave machines; the coordinate operation could be replaced by a pair of gather/scatter operations performed by the master machine, but we wish to avoid the channeling of all such intermediary data through one machine. In fact, without the coordinate operation, our framework closely resembles the Map/Reduce framework, with scatter corresponding to the Map operation, and gather corresponding to the Reduce operation.

In order to evaluate the benefit of the coordinate operation over a pair of gather/scatter operations, we performed an experiment over ∼31m statements evenly and arbitrarily distributed over eight machines. We then attempted to redistribute the data according to the hash of the subject position using both the co-ordinate and the gather/scatter approaches: the coordinate approach took 245 seconds, whilst the gather/scatter approach took 1544 seconds (669 seconds gathering, 728 seconds split, 147 seconds scatter). Given such statistics, and given the need for such an operation in many of our components (especially the indexing), we reasonably justify use of the coordinate operation in addition to the operations supported in the more traditional Map/Reduce type architecture.

We instantiate this architecture using the standard Java Remote Method Invocation libraries as a convenient means of development given our Java code-
base. Please note that much of the evaluation presented in this paper assumes that the slave machines have roughly equal specifications in order to ensure that tasks finish in roughly the same time, assuming even data distribution. We do not currently consider more advanced topics in our architecture; such as load-balancing, replication, uptime and counteracting hardware failure; and discussion of these are outside of the current scope.

3 Preliminaries

Before we continue, we briefly introduce some standard core notation used throughout the paper, relating to RDF terms, triples and quadruples.

**RDF Term** Given a set of URI references $\mathcal{U}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, the set of *RDF terms* is denoted by $\mathcal{RDFTerm} = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. The set of blank nodes $\mathcal{B}$ is a set of existentially quantified variables. The set of literals is given as $\mathcal{L} = \mathcal{L}_p \cup \mathcal{L}_t$, where $\mathcal{L}_p$ is the set of *plain literals* and $\mathcal{L}_t$ is the set of *typed literals*. A typed literal is the pair $l = (s, t)$, where $s$ is the lexical form of the literal and $t \in \mathcal{U}$ is a datatype URI. The sets $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}_p$ and $\mathcal{L}_t$ are pairwise disjoint.

Please note that in this paper, we treat blank nodes as their skolem versions: i.e., not as existential variables, but as denoting their own syntactic form.
**RDF Triple** A triple \( t = (s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L) \) is called an RDF *triple*. In a triple \((s, p, o)\), \(s\) is called subject, \(p\) predicate, and \(o\) object.

**RDF Triple in Context/RDF Quadruple** A pair \((t, c)\) with a triple \(t = (s, p, o)\) and \(c \in U\) is called a *triple in context* \(c\) [32,33]. We may also refer to \((s, p, o, c)\) as the RDF *quadruple* or quad \(q\) with context \(c\).

Note that we commonly use the context element to denote the RDF Web document from which a triple originated in our web crawl.

Finally, we use namespace prefixes in this paper as common in the literature, the full URIs for which can be retrieved from the convenient http://prefix.cc service. We often denote owl: as the default namespace.

### 4 Crawling

The first component required for the building of our index is the crawler: the goal of our crawler is to retrieve a large set of RDF documents from the Web. Our crawler starts with a set of seed URIs, retrieves the content of URIs, parses and writes content to disk and recursively extracts new URIs for crawling; we consider all http: protocol URIs extracted from an RDF document as candidates for crawling.

Like traditional HTML crawlers, we identify the following requirements for crawling:

- **Politeness**: The crawler must implement politeness restrictions to avoid hammering remote servers with dense HTTP GET requests and to abide by policies identified in the provided robots.txt files.
- **Throughput**: The crawler should crawl as many URIs as possible in as little time as is possible within the bounds of the politeness policies.
- **Scale**: The crawler should employ scalable techniques, and on-disk indexing as required.
- **Quality**: The crawler should prioritise crawling URIs it considers to be “high quality”.

Thus, the design of our crawler is inspired by related work from traditional HTML crawlers. Additionally – and specific to crawling structured data – we identify the following requirement:

- **Structured Data**: The crawler should retrieve a high percentage of RDF/XML documents and avoid wasted lookups on unwanted formats: e.g., HTML documents.
Currently, we crawl for RDF/XML syntax documents – RDF/XML is still the most commonly used syntax for publishing RDF on the Web, and we plan in future to extend the crawler to support other formats such as RDFa, N-Triples and Turtle.

The following algorithm details the operation of the crawler, and will be explained in detail throughout this section.

Algorithm 1 Algorithm for crawling

```
Require: SEEDS, ROUNDS, PLD_LIMIT, MAX_DELAY, MIN_DELAY
1: frontier ← SEEDS
2: pld_0...n : queue
3: stats ← new stats
4: while rounds + 1 < ROUNDS do
5:   put frontier into pld_0...n
6:   while depth + 1 < PLD_LIMIT do
7:     rank(pld, stats)
8:   start_time ← current_time;
9:   for i = 0 to n do
10:      get uri from pld_i
11:      fetch uri
12:      add outlinks to frontier
13:      update stats
14:   end for
15:   if (current_time − start_time) < MIN_DELAY then
16:     wait for MIN_DELAY − (current_time − start_time)
17:   end if
18: end while
```

4.1 Breadth-first Crawling

Traditional web crawlers (cf. [9] [39]) typically use a breadth-first crawling strategy: the crawl is conducted in rounds, with each round crawling a frontier. The frontier comprises of seed URIs for round 0, and thereafter with novel URIs extracted from documents crawled in the previous round. Thus, the crawl emulates a breadth-first traversal of inter-linked Web documents (avoiding cycles). On a high-level, Algorithm 1 represents this round-based approach, applying ROUNDS number of rounds, adding newly extracted URIs to the frontier. However, the algorithm is further tailored according to requirements we will describe as the section progresses.

As we will see later in the chapter, the round-based approach fits well with in-
stantiation of our crawler on our distributed framework, allowing for crawlers to work independently for each round, and co-ordinating new frontier URIs at the end of each round. Additionally, [54] show that a breadth-first traversal strategy tends to discover high-quality pages early on in the crawl, with the justification that well-linked documents (representing higher quality documents) are more likely to be encountered in earlier breadth-first rounds; similarly, breadth first crawling leads to a more diverse dataset earlier on, rather than a depth-first approach which may end up traversing deep paths within a given site. In [50], the authors justify a rounds-based approach to crawling according to observations that writing/reading concurrently and dynamically to a single queue can become the bottleneck in large-scale crawler.

4.2 Incorporating Politeness

The crawler must be careful not to bite the hands that feed it by hammering the servers of data providers or breaching policies outlined in the provided robots.txt file. We use pay-level-domains (PLDs; e.g. deri.org) to identify individual data-providers, and implement politeness on a per-PLD basis. Firstly, when we first encounter a URI for a PLD, we cross-check the robots.txt file to ensure that we are permitted to crawl that site; secondly, we implement a “minimum PLD delay” to avoid hammering servers, viz.: a minimum time-period between subsequent requests to a given PLD. This is given by MIN_DELAY in Algorithm 1.

In order to accommodate the min-delay policy with minimal effect on performance, we must refine our crawling algorithm: large sites with a large internal branching factor (large numbers of unique intra-PLD outlinks per document) can result in the frontier of each round being dominated by URIs from a small selection of PLDs. Thus, naive breadth-first crawling can lead to crawlers hammering such sites; conversely, given a politeness policy, a crawler may spend a lot of time idle waiting for the min-delay to pass.

One solution is to reasonably restrict the branching factor [50] – the maximum number of URIs crawled per PLD per round – which ensures that individual PLDs with large internal fan-out are not hammered; thus, in each round of the crawl, we implement a cut-off for URIs per PLD, given by PLD_LIMIT in Algorithm 1.

Secondly, to ensure the maximum gap between crawling successive URIs for the same PLD, we implement a per-PLD queue (given by pld_0...n in Algorithm 1) each PLD is given a dedicated queue of URIs filled from the frontier, and during the crawl, a URI is polled from each PLD queue in a round-robin fashion. If all of the PLD queues have been polled before the min-delay is
satisfied, then the crawler must wait: this is given by Lines 15-17 in Algorithm 1. Thus, the optimal crawl time for a round becomes $MIN_{DELAY} \times PLD_{LIMIT}$.

### 4.3 On-disk Queue

As the crawl continues, the in-memory capacity of the machine will eventually be exceeded by the capacity required for storing URIs [50]. Performing a stress-test, we observed that with 2GB of JAVA heap-space, the crawler could crawl approx. 199k URIs (not including frontier URIs) before throwing an out-of-memory exception. In order to scale beyond the implied main-memory limitations of the crawler, we implement on-disk storage for URIs – with the additional benefit of maintaining a persistent state for the crawl.

We implement the on-disk storage of URIs using Berkeley DB which comprises of two indexes – the first provides lookups for URI strings against their status (polled/unpolled); the second offers a key-sorted map which can iterate over unpollled URIs in decreasing order of inlink count. The inlink count reflects the total number of documents from which the URI has been extracted thus far; we deem a higher count to roughly equate to a higher priority URI.

The crawler utilises both the on-disk index and the in-memory queue to offer similar functionality as above. The on-disk index and in-memory queue are synchronised at the start of each round:

1. links and respective inlink counts extracted from the previous round (or seed URIs if the first round) are added to the on-disk index;
2. URIs polled from the previous round have their status updated on-disk;
3. an in-memory PLD queue is filled using an iterator of on-disk URIs sorted by descending inlink count.

Most importantly, the above process ensures that only the URIs active (current PLD queue and frontier URIs) for the current round must be stored in memory. Also, the process ensures that the on-disk index stores the persistent state of the crawler up to the start of the last round; if the crawler unexpectedly dies, the crawl can be resumed from the start of the last round. Finally, the in-memory PLD queue is filled with URIs sorted in order of inlink count, offering a cheap form of intra-PLD URI prioritisation.
4.4 Multi-threading

The bottle-neck for a single-threaded crawler will be the response times of servers; the CPU load, I/O throughput and network bandwidth of a crawling machine will not be efficiently exploited by sequential HTTP GET requests over the Web. Thus, crawlers are commonly multi-threaded to mitigate this bottleneck and perform concurrent HTTP lookups. At a certain point of increasing the number of lookup threads operating, the CPU load, I/O load, or network bandwidth becomes an immutable bottleneck; this becomes the optimal number of threads.

In order to find a suitable thread count for our particular setup (with respect to processor/network bandwidth), we conducted some illustrative small-scale experiments comparing a machine crawling with the same setup and input parameters, but with an exponentially increasing number of threads: in particular, we measure the time taken for crawling 1,000 URIs given a seed URL for 1, 2, 4, 8, 16, 32, 64, and 128 threads. Also, to alleviate the possible effects of remote caching on our comparison of increasing thread counts, we pre-crawled all of the URIs before running the benchmark.

Figures 4 displays the time taken in minutes for the crawler to run, whilst Figure 5 shows the average percentage CPU usage (the averages are over readings extracted from the UNIX command `ps` taken every three seconds during the crawl). Time and CPU% noticeably have an inverse correlation. As the number of threads increases up until 64, the time taken for the crawl decreases – the reduction in time is particularly pronounced in earlier thread increments; similarly, and as expected, the CPU usage increases as a higher density of documents are retrieved and processed. Beyond 64 threads, the effect of increasing threads becomes minimal as the machine reaches the limits of CPU and disk I/O throughput; in fact, the total time taken starts to increase – we suspect that contention between threads for shared resources affects performance. Thus, we settle upon 64 threads as an approximately optimal figure for our setup.

4.5 Crawling RDF/XML

Since our architecture is currently implemented to index RDF/XML, we would feasibly like to maximise the ratio of HTTP lookups which result in RDF/XML content; i.e., given the total HTTP lookups as \( L \), and the total number of downloaded RDF/XML pages as \( R \), we would like to maximise the ratio \( R/L \).

\[ \text{http://sw.deri.org/~aidanh/foaf/foaf.rdf} \]
In order to reduce the amount of HTTP lookups wasted on non-RDF/XML content, we implement the following heuristics:

1. firstly, we blacklist non-\texttt{http} protocol URIs;
2. secondly, we blacklist URIs with common file-extensions that are highly unlikely to return RDF/XML (e.g., html, jpg, pdf, etc.) following arguments we previously laid out in [70];
3. thirdly, we check the returned HTTP header and only retrieve the content of URIs reporting \texttt{Content-type: application/rdf+xml};
4. finally, we use a \textit{useful ratio score} when polling PLDs to indicate the probability that a URI from that PLD will yield RDF/XML based on past observations.

Although the first two heuristics are quite trivial and should still offer a high theoretical recall of RDF/XML, the third is arguable in that previous observations [44] indicate that 17% of RDF/XML documents are returned with a \texttt{Content-type} other than \texttt{application/rdf+xml}. Thus, we automatically exclude such documents from our crawl; however, here we put the onus on publishers to ensure correct reporting of \texttt{Content-type}.

Finally, we implement an algorithm for selectively polling PLDs based on their observed useful ratio score; since our crawler only requires RDF/XML, we use this score to access PLDs which offer a higher percentage of RDF/XML more often. Thus, we can reduce the amount of time wasted on lookups of HTML documents and save the resources of servers for non-RDF/XML data providers.

The useful ratio score for PLD $i$ is derived from the following credibility formula:

\textsuperscript{5}Indeed, one advantage RDF/XML has over RDFa is an unambiguous MIME-type useful in such situations
usr_i = \frac{rdf_i + \mu}{total_i + \mu}

where rdf_i is the total number of RDF documents returned thus far by PLD i, total_i is the total number of lookups performed for PLD i excluding redirects, and \mu is a “credibility factor”, a rational constant which we set to 10. The purpose of the credibility formula is to dampen scores derived from few readings (where total_i is small) towards the value 1 (offering the benefit-of-the-doubt), with the justification that the credibility of a score with few readings is less than that with a greater number of readings: with a low number of readings (total_i \ll \mu), the usr_i score is affected more by \mu than actual readings for PLD i; as the number of readings increases (total_i \gg \mu), the score is affected more by the observed readings than the \mu factor. Thus, for example, if we observe that PLD a has returned 1/5 RDF/XML documents and PLD b has returned 1/50 RDF/XML documents, then usra = (1 + \mu)/(5 + \mu) = 0.73 and usrb = (1 + \mu)/(50 + \mu) = 0.183 – we thus ensure that PLDs are not unreasonably punished for returning non-RDF/XML documents early on.

To implement selective polling of PLDs according to their useful ratio, we simply use the usr score as a probability of polling a URI from that PLD queue in that round. Thus, PLDs which return a high percentage of RDF/XML documents – or indeed PLDs for which very few URIs have been encountered – will have a higher probability of being polled, guiding the crawler away from PLDs which return a high percentage of non RDF/XML documents.

We evaluated the useful ratio scoring mechanism on a crawl of 100k URIs, with the scoring enable and disable. In the first run, with scoring disabled, 22,504 of the lookups resulted in RDF/XML (22.5%), whilst in the second run with scoring enabled, 30,713 lookups resulted in RDF/XML (30.7%). Table 1 enumerates the top 5 PLDs which were polled and the top 5 PLDs which were skipped for the crawl with scoring enabled, including the useful ratio (usr) and the weighted useful ratio score (usr). Clearly, those in the top 5 were observed to return a high percentage of RDF/XML, and those in the bottom five were observed to return a low percentage of RDF.

4.6 Distributed Crawling Framework

We have seen that given a sufficient number of threads, the bottleneck for multi-threaded crawling becomes the CPU and/or I/O capabilities of one machine; thus, by implementing a distributed crawling framework balancing the CPU workload over multiple machines, we expect to increase the throughput of the crawl. We apply the crawling to our framework as follows:
<table>
<thead>
<tr>
<th>PLD</th>
<th>polled</th>
<th>skipped</th>
<th>ur</th>
<th>usr</th>
<th>% polled</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkedmdb.org</td>
<td>1980</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>geonames.org</td>
<td>1971</td>
<td>26</td>
<td>0.996</td>
<td>0.996</td>
<td>98.7</td>
</tr>
<tr>
<td>rdfabout.com</td>
<td>1944</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>fu-berlin.de</td>
<td>1925</td>
<td>74</td>
<td>0.947</td>
<td>0.949</td>
<td>96.3</td>
</tr>
<tr>
<td>bbc.co.uk</td>
<td>1917</td>
<td>33</td>
<td>0.982</td>
<td>0.982</td>
<td>98.3</td>
</tr>
<tr>
<td>deri.ie</td>
<td>233</td>
<td>1814</td>
<td>0.013</td>
<td>0.054</td>
<td>11.4</td>
</tr>
<tr>
<td>megginson.com</td>
<td>196</td>
<td>1795</td>
<td>0</td>
<td>0.049</td>
<td>9.8</td>
</tr>
<tr>
<td>xbrl.us</td>
<td>206</td>
<td>1785</td>
<td>0</td>
<td>0.046</td>
<td>10.3</td>
</tr>
<tr>
<td>wikipedia.org</td>
<td>203</td>
<td>1784</td>
<td>0</td>
<td>0.051</td>
<td>10.2</td>
</tr>
<tr>
<td>uklug.co.uk</td>
<td>215</td>
<td>1776</td>
<td>0</td>
<td>0.044</td>
<td>10.7</td>
</tr>
</tbody>
</table>

Table 1
PLD ratings

(1) scatter: the master machine scatters a seed list of URIs to the slave machines, using a hash-based injective split function;
(2) run: each slave machine adds the new URIs to it’s frontier, and performs a round of the crawl, slaves write the retrieved and parsed content to the local hard-disk;
(3) coordinate: each slave machine then scatters the new URIs found using the split function to it’s peers;

Steps 1 & 2 are recursively applied until ROUNDS has been fulfilled.

In order to evaluate the effect of increasing the number of crawling machines within the framework, we performed a crawl performing lookups on 100k URIs on 1, 2, 4 and 8 machines using 64 threads. The results are presented in Table 2, showing number of machines, number of minutes taken for the crawl, and also the percentage of times that the in-memory queue had to be delayed in order to abide by our politeness policies. There is a clear increase in the performance of the crawling with respect to increasing number of machines. However, in moving from four machines to eight, the decrease in time is only ~11%. With 8 machines (and indeed, starting with 4 machines), there are not enough active PLDs in the queue to fill the adjusted min-delay of 4 seconds (8*500ms), and so the queue has a delay hit-rate of 94.6%.

We term this state PLD starvation: the slave machines do not have enough unique PLDs to keep them occupied until the MIN_DELAY has been reached. Thus, we must modify somewhat the end-of-round criteria to reasonably improve performance in the distributed case:
Table 2
Time taken for a crawl performing lookups on 100k URLs, and average percentage of time each queue had to enforce a politeness wait, for differing numbers of machines

<table>
<thead>
<tr>
<th>#</th>
<th>mins</th>
<th>% delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>360</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>156</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>71</td>
<td>81.1</td>
</tr>
<tr>
<td>8</td>
<td>63</td>
<td>94.6</td>
</tr>
</tbody>
</table>

- firstly, a crawler can return from a round if the $MIN\_DELAY$ is not being filled by the active PLDs in the queue – the intuition here being that new PLDs can be discovered in the frontier of the next round;
- secondly, to ensure that the slave machines don’t immediately return in the case that new PLDs are not found in the frontier, we implement a $MIN\_PLD\_LIMIT$ which ensures that slave machines don’t immediately return from the round;
- finally, in the case that one crawler returns from a round due to some stopping criteria, if one slave machine returns from a round, the master machine will request that all other slave machines also end their round such that machines do not remain idle waiting for their peers to return.

The above conditions help to somewhat mitigate the effect of PLD starvation on our distributed crawl; however, given the politeness restriction of 500ms per PLD, this becomes a hard-limit for performance independent of system architecture and crawling hardware, instead imposed by the nature of the Semantic Web itself. Also, as a crawl progresses, active PLDs (PLDs with unique content still to crawl) will become less and less, and the performance of the distributed crawler will approach that of a single-machine crawl. As the Semantic Web expands and diversifies, and as the number of servers offering RDF content increases, better performance would be observed for distributed crawling on larger numbers of machines: for the moment, we observe that 8 machines currently approaches the limit of performance given our setup and policies.

4.7 Evaluation

To perform scale-up experiments for the crawler – and indeed to achieve a large dataset for evaluation of later components – we ran the crawler continuously for 48 hours on 8 machines from a seed list of $\sim 8m$ URIs extracted from an old dataset$^6$; in that time, we gathered a total of $1.192b$ quads and we observed a

$^6$ The crawl was conducted in late January, 2010
mean of 149m quads per machine and an average absolute deviation of 1.62m across machines: considering that the average absolute deviation is ~1% of the mean, this indicates near optimal balancing of output data on the machines.

An overview of the total number of URIs crawled per each hour is given in Figure 6; in particular, we observe and attribute the notable decrease (~50%) as the effect of the PLD starvation, which is verifiable by cross-reference to Figure 7.

We show the distribution of the response codes and the content-types for the 200:Okay lookups in Figure 8. The large percentage (77.69%) of application/rdf+xml documents could be attributable to our useful score ratio scheme, which offers an increasing percentage of RDF/XML documents.

Similarly, we can further see from Figure 9 that the percentage of HTTP lookups yielding RDF/XML documents is increasing over time and even more important the fraction of “wasted” lookups – for which PLDs are punished by our useful ratio score – decreases continuously; again, wasted lookups are those HTTP lookups with response codes different from 200 and 30x, as well as 200 lookups with content-types different from application/rdf+xml.

![Fig. 6. Number of HTTP lookups per crawl hour.](image)

### 4.8 Related Work

Parts of our architecture and some of our design decisions are influenced by work on traditional web crawlers; e.g., the IRLBot system of Lee et al. [50] and the distributed crawler of Boldi et. al. [9].

The research field of focused RDF crawling is still quite a young field, with most of the current work based on the lessons learnt from the more mature
Fig. 7. Number of total and new PLDs per crawl hour.

Fig. 8. Response code distribution and content-type distribution for successful lookups.

Fig. 9. Fraction of HTTP lookups per crawl hour.

area of traditional Web crawling. Related work in the area of focused crawling can be categorised [6] roughly as follows:
• classic focused crawling: e.g., [16] uses primary link structure and anchor texts to identify pages about a topic using various text similarity of link analysis algorithms;

• semantic focused crawling: is a variation of classical focused crawling but uses conceptual similarity between terms found in ontologies [24,23]

• learning focused crawling: [20,61] uses classification algorithms to guide crawlers to relevant web paths and pages.

However, a major difference between these approaches and ours is that our definition of high quality pages is not based on topics or ontologies, but instead on the content-type of documents.

4.8.1 Future Direction and Open Research Questions

From a pragmatic perspective, we would prioritise extension of our crawler to handle arbitrary RDF formats – especially the RDFa format which is growing in popularity. Such an extension may mandate modification of the current mechanisms for ensuring a high percentage of RDF/XML documents: for example, we could no longer blacklist URIs with a .html file extension, nor could we rely on the Content-type returned by the HTTP header (unlike RDF/XML, RDFa does not have a specific MIME-type).

The other main challenge posed in this section is that of PLD starvation; although we would expect this to become less of an issue as the Semantic Web matures, it perhaps bears further investigation. For example, we have yet to fully evaluate the trade-off between small rounds with frequent updates of URIs from fresh PLDs, and large rounds which persist with a high delay-rate but require less co-ordination. Also, given the inevitability of idle time during the crawl, it may be practical from a performance perspective to give the crawler more tasks to do inorder to maximise the amount of processing done on the data, and minimise idle time.

Finally, we have not discussed the possibility of incremental crawls: choosing URIs to recrawl may lead to interesting research avenues. Besides obvious solutions such as HTTP caching, URIs could be re-crawled based on, e.g., detected change frequency of the document over time, some quality metric for the document, or how many times data from that document was requested in the UI. More practically, a incremental crawler could use PLD statistics derived from previous crawls, and the HTTP headers for URLs – including redirections – to achieve a much higher ratio of lookups to RDF documents returned. Such considerations would larger countermand the effects of PLD starvation, by reducing the amount of lookups the crawler needs in each run.
5 Entity Consolidation

Different RDF documents on the Web published by independent parties may speak about the same entities using different URIs; to make matters worse, RDF allows for the definition of anonymous entities – entities identified by a blank node – without a prescribed URI. Thus, although in theory RDF offers great potential for automatic data aggregation, such potential is premised on the widespread sharing and re-use, across all sources, of URIs for specific entities.

In fact, using different URIs – or indeed no URIs – to identify entities is common practice on the Web [43]. Offering search and querying over a raw RDF dataset collected from the Web would thus entail many duplicate results referring to the same entity, emulating the current situation on the HTML Web where information about different resources is fragmented across source documents. Given a means of identifying equivalent entities in RDF data – entities representing the same real-world individual but identified incongruously – would enable the merging of information contributions on an entity given by heterogeneous sources without the need for consistent URI naming of entities.

In fact, OWL [65] provides some standard solutions to such problems. Firstly, OWL defines the owl:sameAs property which is intended to relate two equivalent entities; the property has symmetric, transitive and reflexive semantics as one would expect. Many sources on the Web offer owl:sameAs links between entities described locally and equivalent entities described remotely.

Further, OWL provides some other mechanisms for discovering implicit owl:sameAs relations in the absence of explicit relations: the most prominent such example is provision of the class owl:InverseFunctionalProperty, which defines a class of properties whose value uniquely identifies an entity. One example of an inverse-functional property would be an ISBN property, where ISBN values uniquely identify books. If two entities share the same ISBN value, a same-as relation can be inferred between them. Using OWL, same-as relations can also be detected using owl:FunctionalProperty, owl:maxCardinality, and owl:cardinality (and, now in OWL2RL using owl:maxQualifiedCardinality and owl:qualifiedCardinality): however, the recall of inferences involving the latter OWL constructs are relatively small [45] and thus considered out of scope here.

In [43], we provided a simple batch-processing based approach for deriving owl:sameAs relations between individuals using inverse-functional properties defined in the data. However, in our experience, the precision of such inferences can be quite poor. As an example, in [43] we
found 85,803 equivalent individuals to be inferable from a Web dataset through the incongruous values 08445a31a78661b5c746feff39a9db6e4e2cc5cf and da39a3ee5e6b4b0d3255bfef95601890afd80709 for the prominent inverse-functional property foaf:mbox_sha1sum – the former value is the sha1-sum of an empty string and the latter is the sha1-sum of the `mailto:` string, both of which are erroneously published by online FOAF exporters.\(^7\) Aside from such known incidences of false positives being generated through such an approach, publishers commonly do not respect the semantics of inverse-functional properties [44].

In fact, the performance of satisfactory entity consolidation over large-scale RDF Web data is still an open research question. At the moment, we rely on owl:sameAs relations which are directly asserted in the data to perform consolidation; and this section briefly outlines our distributed approach, provides performance evaluation for the algorithm, and provides some insights into the fecundity of such an approach – with respect to finding equivalence – over RDF Web data.

5.1 High-level Approach

The overall approach involves two scans of the main body of data, with the following high-level steps:

1. owl:sameAs statements are extracted from the data: the main body of data is scanned once, identifying owl:sameAs triples and buffering them to a separate location;
2. the transitive/symmetric closure of the owl:sameAs statements are computed, inferring new owl:sameAs relations;
3. for each set of equivalent entities found (each equivalence class), a canonical identifier is chosen to represent the set in the consolidated output;
4. the main body of data is again scanned and consolidated: identifiers are rewritten to their canonical form – we do not rewrite identifiers in the predicate position, objects of rdf:type triples, or literal objects.

In previous work, we have presented two approaches for performing such consolidation; in [43], we stored owl:sameAs in memory, computing the transitive/symmetric closure in memory, and performing in-memory lookups for canonical identifiers in the second scan. In [45], we presented a batch-processing technique which uses on-disk sorts and scans to execute the owl:sameAs transitive/symmetric closure, and the canonicalisation of identifiers in the main body of data. The former approach is in fact much faster in that it reduces the amount of time consumed by hard-disk I/O operations;

\(^7\) See, for example http://blog.livedoor.jp/nkgw/foaf.rdf
however, the latter batch-processing approach is not limited by the main-memory capacity of the system, and will have similar performance for larger sets of \texttt{owl:sameAs} relations.

Either approach is applicable with our consolidation component (even in the distributed case); however, since for the moment we only operate on asserted \texttt{owl:sameAs} statements (we found 3.96m \texttt{owl:sameAs} in our full-scale crawl, which is comfortably within our 4GB in-memory capacity), for now we apply the faster in-memory approach.

The first step in performing our consolidation is the extraction of \texttt{owl:sameAs} statements from the main body of data. Thus, we perform a sequential scan of the data and buffer the \texttt{owl:sameAs} statements to a separate file. The \texttt{owl:sameAs} transitive/symmetric closure of the extracted \texttt{owl:sameAs} statements are then computed. The process can also be distributed over our framework, as we will describe in the following section.

5.2 Distributed Approach

Assuming that the target-data of the consolidation is arbitrarily (and preferably evenly) split over multiple machines – as should be the result of our crawling component – we can apply the consolidation process in a distributed manner as follows:

(1) \texttt{run}: \texttt{owl:sameAs} statements are extracted from the data in parallel on each slave machine;
(2) \texttt{gather}: the \texttt{owl:sameAs} statements are gathered onto the master machine, which computes the transitive/symmetric closure over them, and chooses the canonical identifiers;
(3) \texttt{flood}: the closed \texttt{owl:sameAs} statements and chosen canonical identifiers are sent (in their entirety) to the slave machines;
(4) \texttt{run}: in parallel, the slave machines scan the main body of data, rewriting identifiers to their canonical form and outputting to a new file.

In the above process, only the \texttt{owl:sameAs} statements need be transferred between machines. The more expensive on-disk scans can be conducted in parallel, and thus we would reasonably expect near-linear scale with respect to the number of machines for consolidation over a fixed dataset – the assumptions being that the data has been pre-distributed, that the proportion of \texttt{owl:sameAs} statements is relatively small compared to the main body of data, and that the dataset is relatively large compared with the number of machines (all of which apply in our setting).
In order to evaluate the above claim, we ran a small scale experiment over 1, 2, 4 and 8 machines using a dataset of 31.3M statements extracted from one of the 100k URI crawls from the previous section. The dataset contained 24.9k `owl:sameAs` statements. Table 3 presents the total time taken for each experiment, where in particular, performance appears to be a near-linear function on the number of machines.

### Table 3

<table>
<thead>
<tr>
<th>#</th>
<th>mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>3.2</td>
</tr>
<tr>
<td>8</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Time taken for consolidation of ~31.3M statements for differing numbers of machines

5.3 Full-scale Evaluation

Using the above setup, we ran consolidation over our full-scale RDF dataset with one master and 8 slave machines. The first scan extracting `owl:sameAs` statements took ~15 minutes, with an average idle time for the servers of ~13.5 seconds (i.e., the average time waiting for all servers to finish). Aggregating the `owl:sameAs` statements on the master machine took 88 seconds. The second scan rewriting the data according to the canonical identifiers took in total ~43.6 minutes, with an average idle time of 87 seconds for each machine at the end of the round. The slower time for the second round is attributable to the extra overhead of re-writing the data to disk, as opposed to just reading. The entire consolidation process took ~61.3 minutes. From a performance perspective, we conclude that the methods presented are suitably fast, and that the machines execute their tasks in comparable time, resulting in little idle time waiting for other servers to finish.

With respect to the results of applying the process, identifiers in 70.4m positions were rewritten – roughly 3.6% of the total possible considering all subject positions and object positions which are not literals or are not type triples. Figure 10 presents the distribution of sizes of the equivalence classes; 2,147,024 equivalence classes were found, with the largest equivalence class containing 4,930 equivalent entities and 736k equivalence classes containing two equivalent identifiers.
Fig. 10. Distribution of sizes of equivalence classes (log/log scale)

5.4 Related Work

Entity consolidation has an older related stream of research relating largely to databases, with work under the names of record linkage, instance fusion, and duplicate identification; cf. [56,53,17] and a survey at [25]. Due to the lack of formal specification for determining equivalences, these older approaches are mostly concerned with probabilistic methods.

Bouquet et al. [10] motivate the problem of (re)using common identifiers as one of the pillars of the Semantic Web, and provide a framework and fuzzy matching algorithms to fuse identifiers.

Online systems such as Sig.Ma\(^8\), rkbexplorer\(^9\), and ObjectCoref offer on-demand querying for `owl:sameAs` relations found for a given input URI, which they internally compute and store.

5.5 Future Work and Open Research Questions

We are currently investigating consolidation methods, with particular emphasis on extracting some notion of quality or trustworthiness of derived equivalences: we could extract a larger set of “candidate equivalences” – with higher recall but lower precision – and then apply some trustworthiness threshold to maintain high precision. Candidate equivalences could be derived from the OWL constructs previously mentioned, from simple textual matching, or from statistical approaches analysing the data – such approaches would need to have

\(^8\) http://sig.ma

\(^9\) http://www.rkbexplorer.com/sameAs/
been proven scalable. They could also be derived from the online systems men-
tioned, although performing the large-volume access required to such external
services may not be feasible.

A further avenue for research in the same vein is applying “disambiguation”, or
attempting to assert that two entities cannot (or are likely not) to be equivalent
using statistical approaches: disambiguation would allow for increasing the
precision of the consolidation component by quickly removing “obvious” false
positives. Such approaches would likely have a significant impact on the quality
of data integration possible in an engine such as SWSE operating over RDF
Web data.

6 Ranking

Ranking is an important mechanism in the search process with the function
of prioritising data elements. In traditional Web search engines, the main
purpose of ranking is to determine in which order to display document results.
When dealing with RDF data, there are additional motivations for ranking, for
example which predicate/value pairs to show in a faceted browser, and indeed,
a more complex data model to consider in the ranking procedure. In previous
work [35] we proposed a scalable algorithm for ranking structured data from an
open, distributed environment, based on a concept we term naming authority.
We re-introduce select important discussion from [35] and extend here by
implementing the method in a distributed way and re-evaluating with respect
to performance.

6.1 Requirements

Search on the Web is an extensively studied field. A ranking method for Web
data must help to mitigate the complications inherent in datasets which have
been aggregated from multiple sources, including:

- **Domain variety**: The Web contains information about a multitude of top-
  ics – from social networks to protein pathways; from entertainment to fi-
  nancial data.
- **Structural data**: Aggregating data from many autonomous sources results
in overlapping, redundant, and possibly contradictory information.
- **Noise**: A Web dataset will typically contain a significant number of syntax
  errors, spelling mistakes, wrong identifiers, and other inconsistencies.
- **Spam**: With the lack of controls and the low cost of publishing on the Web,
malicious activity may emerge.
• **Scale:** Identifiers and documents on the Web number in the trillions.

There is a significant body of related work on link-based algorithms for the Web, however these are generally targeted towards hypertext documents. Established methods for ranking web documents are not directly applicable to RDF data for two main reasons: the basic result units of search moves from documents to entities (which may be associated with several sources), and users are able – in addition to keyword searches – to more accurately state their information needs via precise queries.

Existing ranking algorithms for structured data, on the other hand, are typically designed for manually curated data from a small number of sources. These methods generally require the manual input of a domain expert and do not consider the authority of data sources. Search on the Semantic Web requires ranking in datasets which have been integrated from a multitude of disparate sources and exhibit enormous variance in vocabularies used. Thus, neither traditional approaches for ranking Web documents, nor existing methods operating on structured data are applicable.

We require a ranking method for structured Web data, which takes the source of information into account. Therefore the algorithm must be applicable to datasets with the following general properties: i) global, reused identifiers, ii) tracking of provenance, and iii) correspondence between object identifiers and source identifiers. Specifically, we assume the following model:

- a set of identifiers $I$, encompassing a set of global identifiers $U$, a set of local identifiers $B$, and a set of strings $L$ (we omit datatypes such as integer or date for brevity)
- a set of data sources $S \subseteq U$
- a function $ids$ which maps sets of global and local identifiers and literals $i \in I$ to the sources $s \in S$ in which they occur

This generic model can be applied to a wide variety of data models, such as hypertext, graph-structured data, and relational data. We identify the following properties which should be exhibited by a ranking procedure operating on collaboratively-edited datasets which fit such a model:

- The use of an identifier owned by source $s_a$ by a source $s_b$ indicates an acknowledgement of the authority of $s_a$ and should benefit the ranking of $s_a$
- Data providers who reuse identifiers from other sources should not be penalised, i.e. their data sources should not lose any rank value.
- A data provider who simply links to important external identifiers (requiring no input from the external data source) should not gain any rank from doing so. Using only the node-link graph without taking into account the source (e.g. [5]) makes the ranking method receptive for spam: by adding a triple
pointing from a popular URI to a spam URI, the spam URI gains rank from the popular URI.

- We cannot make any assumptions of directionality of links between objects, since link direction is arbitrary (is $u_a$ related to $u_b$ or $u_b$ related to $u_a$?). Thus we cannot use links occurring in the data graph as a vote.

### 6.2 Naming Authority

A principal objective when ranking on the web is rating popular pages higher than unpopular ones. Indeed, Google’s PageRank [58] algorithm interprets hyperlinks to other pages as votes. A possible adaptation for structured data sources would rank popular data sources higher than unpopular ones. However data models such as RDF do not specify explicit links to other web sites or data sources. Therefore a direct adaptation of PageRank for structured data is not possible, since the concept of a hyperlink (interpreted as a vote for a particular page) is missing.

However, a closer examination of the data model leads to the following observation: a crucial feature of structured data sources is the use of global identifiers. Typically, these global identifiers – URIs in case of the web – have a specified syntax, and exploit naming mechanisms such as the domain name system. These practices for labeling resources mean that given an identifier, we can associate it with a particular source.

For example, consider the identifier `http://www.danbri.org/foaf.rdf#-danbri`. Clearly the owner of the `danbri.org` domain can claim authority for creating this URI. Thus, the usage of the URI on other sites can be seen as a vote for the authority of the data source `danbri.org`.

To generalise this idea, one needs to define the notion of “naming authority” for identifiers: a naming authority is a data source with the power to define identifiers of a certain structure. Naming authority is an abstract term which could be applied to the provenance of a piece of information, be that a document, host, person, organisation or other entity. Data items which are denoted by unique identifiers may be reused by sources other than the naming authority.

Hence, the **naming authority** of a global identifier $u \in U$ is the data source $s \in S$ which has the authority to mint the globally unique identifier $u$.

We apply the notion of naming authority at the level of PLDs, motivated by existing work on Web ranking which takes a similar approach. Najork et al. [55] compared results of the HITS [49] ranking approach when performed on the level of document, host and domain granularity and found that domain
granularity returned the best results. In some cases PLD-level granularity may be preferable to domain or host-level granularity because some sites like LiveJournal (which export vast amounts of user profile data in the Friend Of A Friend [FOAF] vocabulary) assign subdomains to each user, which would result in large tightly-knit communities if domains were used as naming authorities. Previous work has performed PageRank on levels other than the page level, for example at the more coarse granularity of directories, hosts and domains [47], and at a finer granularity such as logical blocks of text [14] within a page.

An additional advantage to ranking naming authorities at the PLD level is that the size of the input graph to the PageRank calculation is significantly reduced, which implies faster execution of the ranking procedure – especially, as we will see later, over our distributed framework.

6.3 Approach

The method consists of the following steps:

1. Based on the occurrence of identifiers \( u \in S \), construct the naming authority graph \( S \times S \) that serves as input to a fixpoint calculation.
2. From the naming authority graph, derive PageRank scores for the data sources \( S \).
3. Using the source ranks, derive a rank value for both global identifiers \( u \in U \) and data elements with local scope \( b \in B, l \in L \).

6.3.1 Deriving the Naming Authority Matrix

As a first step, we derive the naming authority graph from the input dataset. That is, we construct a graph which encodes links between data sources, based on the implicit connections created via identifier reuse.

Given a data source \( s_i \in S \) we specify the naming authority matrix \( A \) as a square matrix defined as:

\[
    a_{i,j} = \begin{cases} 
        1 & \text{if } s_i \text{ uses identifiers for which } s_j \text{ has naming authority} \\
        0 & \text{otherwise}
    \end{cases} \tag{1}
\]

The algorithm can be implemented using a single scan over the dataset which derives both the naming authority matrix and a data structure that records
the use of terms in data sources; as such, the method can be applied to streaming data. Subsequent calculations can then be carried out on the intermediate data structures without the use of the original data. Note that we (optionally, and in the case of later evaluation) do not consider URIs found in the predicate position of a triple, or the object position of an `rdf:type` triple, in the derivation of the naming authority graph, such that we do not want to overly inflate scores for vocabularies, with the justification that users will generally not be interested in results describing the domain of knowledge itself [35].

6.3.2 Calculating Source Ranks

Having constructed the naming authority matrix, we now can compute scores for data sources. For computing ranking scores we calculate PageRank over the naming authority graph: we calculate the dominant eigenvector of the naming authority graph using the Power iteration while taking into account a damping factor (see [58] for more details).

In the input graph there may be sources which have no outlinks, referred to by Page et. al. as dangling nodes [58]. The rank of these dangling nodes is split and distributed evenly across all remaining nodes. Conversely, there might be sources which have no inlinks, in the case where nobody uses the source’s identifier, or identifiers the source speaks authoritatively for; these sources only receive the damping factor plus the rank of the dangling nodes.

6.3.3 Calculating Identifier Ranks

We can now propagate data source scores to both global identifiers and identifiers and literals with a local scope which cannot be re-used in other sources.

Based on the rank values for the data sources, we now calculate the ranks for individual identifiers. The rank value of the individual identifier \( u \in U \) depends on the rank values of the data sources \( s \in S \) where the identifier occurs. The identifier rank of a global identifier \( u \in U \) is defined as the sum of the ranks of the sources \( s \in S \) in which \( u \) occurs.

\[
\text{identiﬁerrank}(u) = \sum_{s \in \{ s | u \in s; s \in S \}} \text{ sourcerank}(s) \tag{2}
\]

The identifier rank of local identiﬁers \( b \in B \) and \( l \in L \) are deﬁned as the source rank of the source in which \( b \) or \( l \) occurs: i.e., the score of literals is not aggregated across sources. Note again – and with similar justiﬁcation as for deriving the named authority graph – we do not include URIs found in the
predicate position of a triple, or the object position of an \texttt{rdf:type} triple in the above summation for our evaluation.

### 6.4 Quality Evaluation

Full details of our quality evaluation are available in [35]; to summarise our findings, we found that our method performed significantly better than the baseline method of implementing PageRank on the node-link graph (an approach which is similar to existing work such as ObjectRank [5]). Also, we found that use of the PLD level graph and URI level graph as input for our PageRank calculations yielded roughly equivalent results for identifier ranks in our user evaluation. Thus, given optimisations possible on the smaller PLD-based graph, we choose this method for implementation over our distributed framework: firstly, the PLD graph is smaller than the URI level graph and thus scores are faster to compute; secondly, the PLD scores are globally required knowledge, and so the smaller data will be cheaper to gather, aggregate and flood to the slave machines.

### 6.5 Distributed Approach

We now discuss our approach for applying the ranking analysis over our distributed framework – we again assume that the input data are evenly distributed across the slave machines:

1. **run**: each slave machine scans its segment of the data in parallel, and extracts PLD level links;
2. **gather**: the master machine gathers the PLD graph from the slave machines, aggregates the links, executes the PageRank algorithm, and derives the scores for each PLD;
3. **flood**: the master machine sends the PLD scores to all machines;
4. **run**: the slave machines calculate and summate the identifier-ranks given the PLD scores and their local view on the segment of data;
5. **gather**: the master machine must now gather the identifier-ranks from the slave machines, and aggregate the scores – importantly, rank contributions for a given identifier from a given PLD must be uniqued across machines.

We again performed some smaller-scale experiments to illustrate the performance advantages of distribution for our ranking methods, demonstrating again over \(\sim 31m\) statements and 1, 2, 4, and 8 machines. Table 4 presents the results. The distribution exhibits near-linear scale with respect to the number
Table 4
Time taken for ranking of ~31.3M statements for differing numbers of machines

<table>
<thead>
<tr>
<th>#</th>
<th>mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42.7</td>
</tr>
<tr>
<td>2</td>
<td>22.5</td>
</tr>
<tr>
<td>4</td>
<td>12.6</td>
</tr>
<tr>
<td>8</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Table 5
Top 5 ranked PLDs (values are normalised)
of machines, with the most expensive tasks being run in an embarrassingly parallel fashion: we will see more in our full-scale experiments to follow.

6.6 Full-Scale Evaluation

We now discuss the results of applying the ranking procedure over our full-scale crawl (1.192b statements) over 8 machines. The entire process took ~212.2 minutes. The PLD-level graph was extracted in parallel in ~62.5 minutes, with an average idle time of ~4.5 minutes (7%) for machines waiting for their peers to finish. The PageRank calculation – performed in memory – took only 13 seconds. Calculating the identifier ranks in parallel took ~146.4 minutes, with an average idle time of ~7 minutes (4.8%). Locally aggregating the ranks took ~2 minutes.

Table 5 presents the top 5 ranked PLDs. Interestingly, e.g., the domain w3.org

\footnote{In fact, we decided to extract the PLD graph from the unconsolidated data: to derive said graph as it was natively found on the Web, unaffected by the consolidation process; and to apply identifier ranking over the consolidated data: to ensure that the identifier ranks were aggregated correctly for canonicalised identifiers in the consolidated data going forward. This, however, would have minimal effect on the performance evaluation presented.}
There have been several methods proposed to handle the task of ranking Semantic Web data. The system which uses the most similar approach to SWSE is Swoogle [22]. Swoogle ranks documents using the OntoRank method, a variation on PageRank which iteratively calculates ranks for documents based on references to terms (classes and properties) defined in other documents. We extend the method described in [22] in several important ways: we generalise the notion of term use to naming authority which establishes a connection between identifier and source; we include the PLD abstraction layer which has been found to be advantageous for ranking in the web environment; and we extend our ranking scheme to not only cover vocabulary terms but instance identifiers as well, which is important in our Linked Data browser use-case scenario.

ObjectRank [5] is an approach to rank a directed labelled graph using PageRank. The work includes a concept called authority transfer schema graphs, which defines weightings for the transfer of propagation through different types of links. ObjectRank relies on user input to weight the connections between nodes to describe their semantic weight, so that the three-way representation can be collapsed into a two-way matrix, on which a PageRank-style algorithm is applied. Our approach does not require any manual input, which is not feasible given the scale and heterogeneity of the input. In addition, omitting the provenance of data as in ObjectRank opens up the method to abuse - anyone could maliciously link to their own identifiers from well-known, highly ranked identifiers and therefore gain reputation by association. Using our notion of naming authority, reusing popular identifiers only results in a propagation of reputation from the containing sources to the popular source.

ReConRank [42] applies a PageRank-type algorithm to a graph which unifies the documents and resources in a dataset. The method generates scores for the documents and entities in a collection, but not for the properties. ReConRank does take data provenance into account, however because it simultaneously operates on the object graph, it is still susceptible to spamming.

SemRank [3] ranks relations and paths on Semantic Web data using information-theoretic measures. In contrast, we assign a rank value to all identifiers occurring in the data sources, based on a fixpoint calculation on the naming authority graph.

AKTiveRank [1] is a system for ranking ontologies based on how well they cover specified search terms. AKTiveRank combines the results of multiple analytic methods to rank each ontology. Individual instances and vocabulary terms are not ranked. Ontocopi [2] provides a way of locating instances in a
knowledge base which are most closely related to a target instance. The Ontocopi tool uses a spreading activation algorithm and allows both manual and automatic tuning. However the source of data is not taken into consideration. Similarly, the SemSearch system [51] ranks entities according to how well they match the user query but does not consider the source of data.

6.8 Future Work and Open Research Questions

Ranking in web search engines depends on a multitude of factors, ranging from globally computed ranks to query-dependent ranks to location, preferences, and history of the searcher. Factoring additional signals into the ranking procedure is an area for further research, especially in the face of complex database-like queries and results beyond the simple list of objects. For example, we have already seen that we exclude predicate and class identifiers from the ranking procedure, in order not to adversely affect our goal of ranking entities (individuals) in the data; specific modes and display criteria of the UI may require different models of ranks, providing multiple contextual ranks for identifiers in different roles – e.g., creating a distinctive ranking metric for identifiers in the role of predicates, reflecting the expectations of users given various modes of browsing.

Another area for research is the question of finding appropriate mathematical representations of directed labelled graphs, and appropriate operations on them [34], [29]. Most of the current research in ranking RDF graphs is based around the directed graph models borrowed from hypertext ranking procedures. A bespoke mathematical model for RDF (directed, labelled, and named) graphs may lead to a different view on possible ranking algorithms.

Finally, the evaluation of link-based ranking as an indicator of trustworthiness would also be a valid contribution; thus far, we have evaluated the approach according to user evaluation reflecting preference for the prioritisation of entity results in the UI. However, given that we also consider the source of information in our ranking, we could see if there was a co-occurrence, for example, of poorly-ranked PLDs and noisy and inconsistent data. Such a result would have particular impact for the reasoning component, presented next, and some discussion is provided in the respective future work and directions section to follow.
7 Reasoning

Using the Web Ontology Language (OWL) and the RDF Schema language (RDFS), instance data (i.e., assertional data) describing individuals can be supplemented with structural data (i.e., terminological data) describing classes and properties, allowing to well-define the domain of discourse and ultimately provide machines a more sapient understanding of the RDF data. We have already seen that OWL semantics can be used to automatically aggregate heterogeneous data – using owl:sameAs relations and, e.g., the owl:Inverse-FunctionalProperty to derive said – where the knowledge is fractured by use of discordant identifiers. However, RDFS and OWL descriptions in the data can be further exploited to infer new statements based on the terminological knowledge and provide a more complete dataset for query answering, and to automatically translate data from one conceptual model to another (where appropriate mappings exist in the data).

In previous work [45], we introduced the Scalable Authoritative OWL Reasoner (SAOR) system for performing large-scale materialisation using a rule-based approach over a fragment of OWL. We now re-introduce important discussion from that paper, and provide novel contribution in the form of extension of the reasoning to a fragment of OWL2RL [31], and performance evaluation from applying the approach over our distributed framework. For a more thorough treatment and formalisations relating to the following discussions, we refer the interested reader to [45].

7.1 Requirements

Applying RDFS and OWL reasoning at large scale has only recently become a more mature area of research [71,73,45], with most legacy works focussing on more expressive logics and theoretical considerations such as computational complexity and completeness, demonstrating evaluation typically over clean and domain-specific datasets, and relying largely on in-memory computation.

In SAOR – and indeed motivated by the Semantic Web Search Project – we identified the following requirements for large-scale RDFS and OWL reasoning over Web data:

• **pre computation**: the system should pre-compute inferences to avoid the

---

11 Note that in this paper, we deliberately decouple consolidation and reasoning, since in future work, we hope to view the unique challenges of finding equivalent identifiers as separate from those of inferencing according to terminological data presented here.
runtime expense of backward-chaining, such that could negatively impact upon response times;

- **reduced output**: the system should not produce so many inferences that it over-burdens the consumer application;
- **scalability**: the system should scale near-linearly with respect to the size of RDF Web dataset;
- **web-tolerant**: the system should be tolerant to noisy and possibly inconsistent data on the Web;
- **domain-agnostic**: the system should be applicable over data from arbitrary domains, and consider non-core Web ontologies (ontologies other than RDF(S)/OWL) as equals.

We now continue by introducing our approach, and showing how reasoning over web data can be conducted in a manner sympathetic with the above requirements.

### 7.2 High-level Approach

Firstly, we choose a rule-based approach since it offers greater tolerance in the inevitable event of inconsistency than Description Logics based approaches – indeed, consistency cannot be expected on the Web (cf. [44] for our discussion on reasoning issues in Web data). Secondly, rule-based approaches offer greater potential for scale following arguments made in [28]. Finally, many web-ontologies – although relatively lightweight and inexpressive – are not valid DL ontologies: for example, FOAF defines the data-type property `foaf:mbox_sha1sum` as inverse-functional, which is disallowed in OWL DL – in [7] and [72], the authors provided surveys of Web ontologies and showed that most are in OWL Full, albeit for largely syntactic reasons.

However, there does not exist a standard ruleset suitable for application over arbitrary Web data – we must compromise and deliberately abandon completeness, instead striving for a more pragmatic form of reasoning tailored for the unique challenges of Web reasoning: firstly, OWL Full reasoning is well known to be undecidable [62]; secondly – and as a demonstration of applying complete reasoning over arbitrary RDF data – with the following triples in a knowledge base, naively applying standard RDFS/OWL rules would lead to the inference of all possible triples:

```
rdfs:subClassOf rdfs:subPropertyOf rdfs:Resource.
rdfs:subClassOf rdfs:subPropertyOf rdfs:subPropertyOf.
rdf:type rdfs:subPropertyOf rdfs:subClassOf.
rdfs:subClassOf rdf:type :SymmetricProperty.
```
In [45] we discussed the tailoring of a non-standard OWL ruleset – pD* given by ter Horst in [67] – for application over Web data. More recently, OWL 2 has become a W3C Recommendation, and interestingly from our perspective, includes a standard rule-expressible fragment of OWL, viz.: OWL 2 RL [31]. In [41], we presented discussion on the new ruleset from the perspective of application over Web data, and showed that the ruleset is not immediately amenable to the requirements outlined, and still needs amendment for our purposes. In this paper, we follow on from discussion in [41] and implement a fragment of OWL 2 RL: we present our ruleset in Appendix A and now discuss how we tailored the standard ruleset [31] for our purposes.\textsuperscript{12}

Firstly, we do not apply rules which infer what we term as “extended axiomatic statements”, which refer to syntactic RDFS and OWL statements such as \texttt{rdf:type rdfs:Resource} statements, and reflexive \texttt{owl:sameAs} statements – statements which apply to every term in the graph. Given \(n\) such rules, and \(t\) unique terms in the dataset, such rules would burden the consumer application with \(t \times n\) largely jejune statements.

Secondly, we identified that separating terminological data (our so called T-Box) that describes classes and properties from assertional data (our A-Box) that describes individuals could lead to certain optimisations in rule execution, leveraging the observation that only \(\sim 1\%\) of RDF Web data is terminological, and that the terminological data is the most frequently accessed segment for OWL reasoning. We used such observations to justify the identification, separation, and provision of optimised access to our T-Box, storing it in memory.

Thirdly, after initial evaluation of the system at scale encountered a puzzling deluge of inferences, we discovered that incorporating the source of data into the reasoning algorithm is of utmost importance; naïvely applying reasoning over the merge of arbitrary RDF graphs can lead to unwanted inferences whereby third parties redefine classes and properties provided by popular ontologies – what we call “ontology hijacking”. We counter-act such behavior by incorporating the analysis of authoritative sources for classes and properties in the data.

We will now discuss the latter two issues in more detail, but beforehand let us treat some preliminaries used in this section.\textsuperscript{13}

\textsuperscript{12} Please note that we do not consider rules which infer an inconsistent (have a \texttt{false} consequent), and consider equality reasoning separately in the consolidation component: thus we do not support any of the rules in rule group \(\mathcal{R2}\) as defined in [41] – also of note, we co-incidentally do not supported any rules which use the new OWL 2 constructs, since they require A-Box joins which – as we will justify herein – our system currently does not support.

\textsuperscript{13} Please note that we largely re-use the definitions provided in [45], which are required here for further discussion of our reasoning approach.
### 7.3 Preliminaries

**Web Knowledge-base** Given a set $S_W$ of RDF web graphs, our view of a web knowledge-base $KB$ is taken as a set of pairs $(W', c)$ for each $W \in S_W$, where $W'$ contains a unique set of blank nodes for $S_W$ and $c$ denotes the URL location of $W$.

Informally, $KB$ is a set of quadruples retrieved from the Web wherein the set of blank nodes are unique for a given document and triples are enhanced by means of context which tracks the web location from which each triple is retrieved. The notion of a Web knowledge-base formalises the output of our crawl. We use the abbreviated notation $W \in KB$ or $W' \in KB$ where we mean $W \in S_W$ for $S_W$ from which $KB$ is derived or $(W', c) \in KB$ for some $c$.

**Meta-class** Informally, a *meta-class* is a class of classes or properties; i.e., the members of a meta-class are either classes or properties; we restrict our notion of meta-classes to the set defined in RDF(S) and OWL specifications used in our OWL 2 RL rules, viz.: \{owl:AnnotationProperty, owl:Class, owl:DatatypeProperty, :ObjectProperty, :SymmetricProperty\}.

**Meta-property** A *meta-property* is one which has a meta-class as it’s domain; again, we restrict our notion of meta-properties to the set defined in RDF(S) and OWL specifications and used in our rules, viz.: \{rdfs:domain, rdfs:range, rdfs:subClassOf, rdfs:subPropertyOf, owl:equivalentClass, owl:equivalentProperty, owl:hasValue, owl:intersectionOf, owl:inverseOf, owl:minCardinality, owl:oneOf, owl:onProperty, owl:someValuesFrom, owl:unionOf\}.

#### 7.3.0.1 Terminological Triple

We define a *terminological triple* as one of the following:

1. an triple with a meta-class as the object and rdf:type as the predicate;
2. a triple with a meta-property as predicate;
3. a triple in a non-branching, non-cyclic path $t_{0}^{r}, ..., t_n^{r}$ where $t_0^{r} = (s_0, p_0, o_0)$ for $p_0 \in \{:intersectionOf, :oneOf, :unionOf\}$; $t_k^{l} = (o_{k-1}, \text{rdf:rest}, o_k)$ for $1 \leq k \leq n$, $o_{k-1} \in B$ and $o_n = \text{rdf:nil}$; or a triple $t_k^{l} = (o_k, \text{rdf:first}, e_k)$ with $o_k$ for $0 \leq k < n$ as before.

We refer to triples $t_{1}^{l}, ..., t_n^{l}$ and all triples $t_k^{l}$ as *terminological collection triples*, whereby RDF collections are used in a union, intersection or enumeration class.
description. Informally, we identify terminological statements as all of those matching the patterns headlined “terminological” in Appendix A.

7.3.0.2 Inference Rule We define an inference rule \( r \) as the pair \((\text{Ante}, \text{Con})\), where the antecedent \( \text{Ante} \) and the consequent \( \text{Con} \) are basic graph patterns [63], and at least one variable co-exists in \( \text{Ante} \) and \( \text{Con} \). In this paper, we use SPARQL-like syntax to represent graph-patterns, and will typically formally write inference rules as:

\[
\text{Ante} \Rightarrow \text{Con}
\] (3)

Rule Application and Closure We define a rule application in terms of the immediate consequences of a rule \( r \) or a set of rules \( \mathcal{R} \) on an RDF graph \( \mathcal{G} \). That is, if \( r \) is a rule of the form (3), and \( \mathcal{G} \) is a set of RDF triples, then:

\[
T_r(\mathcal{G}) = \{ \mu(\text{Con}) \mid \exists \mu \text{ such that } \mu(\text{Ante}) \subseteq G \}
\]

such that \( \mu \) is a mapping \( \mu : \mathcal{V} \cup \mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \to \mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \), which maps constant RDF terms reflexively, and maps variables to constants injectively. Accordingly \( T_\mathcal{R}(\mathcal{G}) = \bigcup_{r \in \mathcal{R}} T_r(\mathcal{G}) \). Also, let \( \mathcal{G}_{i+1} = \mathcal{G}_i \cup T_\mathcal{R}(\mathcal{G}_i) \) and \( \mathcal{G}_0 = \mathcal{G} \); we now define the exhaustive application of the \( T_\mathcal{R} \) operator on a graph \( \mathcal{G} \) as being up to the least fixpoint (the smallest value for \( n \)) such that \( \mathcal{G}_n = T_\mathcal{R}(\mathcal{G}_n) \). We call \( \mathcal{G}_n \) the closure of \( \mathcal{G} \) with respect to ruleset \( \mathcal{R} \), denoted as \( Cl_\mathcal{R}(\mathcal{G}) \). Note that we may also use the intuitive notation \( Cl_\mathcal{R}(\mathcal{KB}) \), \( T_\mathcal{R}(\mathcal{KB}) \) as shorthand for the more cumbersome \( Cl_\mathcal{R}(\bigcup_{W \in \mathcal{KB}} W) \), \( T_\mathcal{R}(\bigcup_{W \in \mathcal{KB}} W) \).

7.4 Separating Terminological Data

Given the above notation and discussion, we can progress by more formally defining our notion of a T-Box:

**Definition 1 (T-Box)** Let \( T_\mathcal{G} \) be the union of all graph pattern instances from a graph \( \mathcal{G} \) for one of the terminological graph patterns enumerated in Appendix A; i.e., \( T_\mathcal{G} \) is itself a graph. We call \( T_\mathcal{G} \) the T-Box of \( \mathcal{G} \).

For our knowledge-base \( \mathcal{KB} \), we define our T-Box \( \mathcal{T} \) as the set of all pairs \( (T_W, c) \) where \( (W, c) \in \mathcal{KB} \) and \( T_W \neq \emptyset \). Again, we may use the intuitive notation \( T_W \in \mathcal{T} \). We define our A-Box \( \mathcal{A} \) as containing all of the statements in \( \mathcal{KB} \), including \( \mathcal{T} \); i.e., unlike description logics, our \( \mathcal{A} \) is synonymous with our \( \mathcal{KB} \). We use the term A-Box to distinguish data that are stored on-disk (which includes T-Box data also stored in memory).
Intuitively, our T-Box represents the descriptions of classes and properties required in our rules. Thus, terminological graph patterns requiring information about classes and properties are sourced from the T-Box, for which we optimise access. To more formally introduce this notion, we must introduce our notion of a $T$-split inference rule, whereby part of the antecedent is a basic graph pattern strictly instantiated by a static T-Box $T$.

**Definition 2 ($T$-split inference rule)** We define a $T$-split inference rule $r$ as the triple $(\text{Ante}_T, \text{Ante}_G, \text{Con})$, where $\text{Ante}_T$ is a basic graph pattern matched by a static T-Box $T$ and $\text{Ante}_G$ is matched by data in the graph $G$; if $\text{Ante}_T$ and $\text{Ante}_G$ are non-empty, then they must share at least one common variable.

We generally write $(\text{Ante}_T, \text{Ante}_G, \text{Con})$ as $\text{Ante}_T \Rightarrow \text{Ante}_G \Rightarrow \text{Con}$. We call $\text{Ante}_T$ the terminological or T-Box antecedent pattern and $\text{Ante}_G$ the assertional or A-Box antecedent pattern.

**Definition 3 ($T$-split inference rule application)** We define a $T$-split rule application to be $T_r(T, G)$ for $r = (\text{Ante}_T, \text{Ante}_G, \text{Con})$ as follows:

$$T_r(T, G) = \{ \mu(\text{Con}) \mid \exists \mu \text{ such that } \mu(\text{Ante}_T) \subseteq T \text{ and } \mu(\text{Ante}_G) \subseteq G \}$$

Thus, in our reasoning system, we have a well defined distinction between T-Box and A-Box information, reflected in the definition of our rules, and the application of rules over the T-Box split data. Again, by introducing this decoupling of T-Box and A-Box, we can incorporate the following optimisations:

1. knowing that the T-Box is relatively small and is the most frequently accessed segment of crawling – e.g., all of the rules in Appendix A require terminological knowledge – we can store the T-Box in an optimised index;
2. we can identify optimised $T$-split rules as those with low assertional-arity – namely, rules which do not require joins over a large A-Box can be performed in an optimal and scalable manner;
3. we will later use the separation of the T-Box as an integral part of our distributed approach.

With respect to the first possible optimisation, at the moment we store the entire T-Box in memory, but on-disk indices can be employed as necessary.\(^{14}\) We will refer in Section 7.6 to the third optimisation avenue.

With respect to the second optimisation, in [45], we showed that rules involving more than one assertional pattern (i.e., involved a query pattern join over

\(^{14}\) We would expect a caching approach to work well, since many class/property descriptions will be accessed more frequently; this follows from statistics presented in Tables B.1 & \fig{statsprops}.
the large A-Box) were in practice difficult to compute. Thus, we categorised rules according to the **assertional arity** of their antecedent; i.e., the number of assertional patterns in the antecedent. In [41], we performed similar categorisation of OWL 2 RL rules. The rules in Appendix A follows this categorisation, showing rules with no antecedent (axiomatic triple rules) in Table A.1, rules with only terminological patterns answerable entirely from our T-Box in Table A.2, and rules with one assertional pattern and at least one terminological pattern in Table A.3. Not shown are rules with multiple assertional patterns; we currently only apply reasoning over rules with less than one assertional pattern using an optimised approach as follows.

Our approach consists mainly of two scans:

1. to commence, we apply rules with no antecedent (Table A.1), inferring axiomatic statements;
2. we then run the first scan of the data, identifying terminological knowledge found in the data, and separating and storing the data in our in-memory T-Box representation;
3. using this T-Box, we apply rules which only require T-Box knowledge (Table A.2) – we now have our static T-Box;
4. the second scan sequentially joins individual A-Box statements with the static in-memory T-Box (Table A.3).

At all points, the inferred statements are written to disk. Inferred statements are also recursively scanned and joined with the T-Box as they appear. Thus – and as follows from our categorisation and selection of rules – we need not index the A-Box and need not perform expensive intra-A-Box joins, performing reasoning at the cost of two sequential scans of the input data, and the cost of writing the inferred statements to disk.\(^\text{15}\)

Further, in [45], we showed that in the ruleset we applied over Web data, 99.7% of inferences occurred through rules with zero or one assertional patterns; we thus proposed that such rules cover the inferencing required by the lightweight web vocabularies whose terms are commonly used on the Web (see Table B.3 for an indication of the most popular vocabularies currently used in RDF Web data).

\(^\text{15}\)Discussion of completeness is outside the scope of this paper, but informally, we posit that we are complete with respect to the exhaustive application of our ruleset excluding inferences which involve terminological statements being inferred by rules with assertional antecedent patterns. This would usually occur through non-standard usage of the core vocabulary terms (meta-classes and -properties) [45].
7.5 Authoritative Reasoning

In initial evaluation of our SAOR system, we encountered a puzzling deluge of unexpected inferences; for example, we found that inferencing on single membership assertion of \texttt{owl:Thing} caused 4,251 inferences.\(^{16}\) For reasoning on a single membership assertion of the popular \texttt{foaf:Person} class wrt. to our T-Box, we found 4,631 inferences: an additional 380 inferences on top of \texttt{owl:Thing}, whereas the FOAF specification mandates six inferences. As another example, another document\(^ {17}\) defines nine of its \textit{properties} as being the domain of \texttt{rdf:type}. We generally term such third-party interference with remote class/property definitions as “ontology-hijacking”. For this, we re-use a similar notion to that of named authority for URIs introduced in Section 6.2, here instantiating the more generic concept for RDF:

\textbf{Definition 4 (Authoritative Source)} A web-graph \(W\) from source (context) \(c\) speaks authoritatively about an RDF term \(n\) iff:

\begin{enumerate}
\item \(n \in B\); or
\item \(n \in U\) and \(c\) coincides with the document dereferenced by \(n\).
\end{enumerate}

For example, the FOAF document is authoritative for all terms in the \texttt{foaf:} namespace. Intuitively, all documents are authoritative for their blank nodes, since blank node labels identifiers are locally scoped. We can now define ontology hijacking:

\textbf{Definition 5 (Ontology Hijacking)} Let \(T_W\) be the T-Box extracted from a web-graph \(W\) and let \(\widehat{\text{sig}}(W)\) be the set of classes and properties for which \(W\) speaks authoritatively; then if \(Ct_{\mathcal{R}_G}(T_W, G) \neq G\) for any \(G\) not mentioning any element of \(\widehat{\text{sig}}(W)\), we say that web-graph \(W\) is performing ontology hijacking.

In simpler terms, ontology hijacking is the contribution of terminological statements about classes and/or properties in a non-authoritative source such that reasoning on those classes and/or properties is somehow affected. One particular method of ontology hijacking is defining new super-classes or properties of third-party classes or properties. Note again that this does not include extension of classes and properties; for example, the triple \texttt{foaf:name rdfs:subPropertyOf hijack:name} , is hijacking the \texttt{foaf:name} property and effecting the translation of all \texttt{foaf:name} statements in the web knowledge-base into \texttt{hijack:name} statements as well. However, the statement \texttt{hijack:name rdfs:subPropertyOf foaf:name} is a valid example of extension of an exter-

\(^{16}\)For example, the document \texttt{http://lsdis.cs.uga.edu/~oldham/ontology/ wsag/wsag.owl} accounts for 55 such inferences where \texttt{owl:Thing} is a member of 55 union class descriptions.

\(^{17}\)\texttt{http://www.eiao.net/rdf/1.0}
nal class or property, assuming publication in a location authoritative for the hijack: namespace.

Although not all such cases are necessarily malicious by any means, they at the very least cause an increase in inferred triples using properties and classes in obscure namespaces: for the moment, we are mainly concerned with ontology hijacking with respect to it’s infringement on the requirement for a reasonable result size so as not to over-burden later consumers of the data – in particular, the indexing component; however, as the Semantic Web gathers more attention and spamming becomes a possible issue, ontology hijacking would also cover such malicious behavior.

Clearly, such third-party contributions should not affect reasoning on triples using those terms. In order to negate such contributions in our reasoning – and thereby shield the definitions of classes and properties as defined in their original specifications – we identified the need to consider the source of terminological data when performing reasoning, and proposed our notion of authoritative reasoning, involving the following modification to our rule application procedure:

**Definition 6 (Authoritative Rule Application)** Again let \( \tilde{\text{sig}}(W) \) be the set of classes and properties for which \( W \) speaks authoritatively and let \( T_W \) be the T-Box of \( W \). We define an authoritative rule application for a graph \( G \) w.r.t. the T-Box \( T_W \) to be a \( T \)-split rule application \( T_r(T_W, G) \) where additionally, if both \( \text{Ante}_T \) and \( \text{Ante}_G \) are non-empty, then for the mapping \( \mu \) of \( T_r(T_W, G) \) there must exist a variable \( v \in (V(\text{Ante}_T) \cap V(\text{Ante}_G)) \) such that \( \mu(v) \in \tilde{\text{sig}}(W) \). We denote an authoritative rule application by \( \text{T}^\#(T_W, G) \).

In essence, the authoritative application of our rules requires that the terminological data satisfying the terminological patterns in our antecedent to speak authoritatively for at least one term in the assertional statement matching the assertional pattern. In Table A.3, we identify authoritative variables – those variables which must be authoritatively spoken for in the T-Box – using boldface. Take the following rule as an example, where the terminological pattern is underlined and the authoritative variable is in bold:

\[
\text{?p rdfs:domain ?c . ?x ?p ?y . } \Rightarrow \text{?x a ?c .}
\]

Here, the term matched by \( \text{?p} \) must be authoritatively spoken for by the document serving the \text{rdfs:domain} triple. Therefore, taking the previous example where nine domains for \text{rdf:type} are non-authoritatively defined, the document \text{http://www.eiao.net/rdf/1.0} does not speak authoritatively for \text{rdf:type}, which is bound by \( \text{?p} \): thus, no inference takes place.
7.6 Distributed Approach

We now show how the above techniques can be applied to perform authoritative reasoning wrt. our ruleset over our distributed framework. Again, assuming that the data is distributed (preferably evenly, and possibly in an arbitrary fashion) over the slave machines, we can apply the following distributed approach:

1. run: each slave machine scans it’s segment of the knowledge-base in parallel, extracting terminological statements;
2. gather: the master machine firstly executes all axiomatic rules locally; the master machine then gathers all terminological statements found by the slave machines in the previous step; the terminological statements are indexed in memory, with authoritative analysis applied to each of the statements; finally, rules requiring only terminological knowledge are executed locally;
3. flood: the master machine sends the authoritative in-memory T-Box to all machines;
4. run: the slave machines performs authoritative application of the rules in Table A.3, joining statements in their segment of the knowledge-base (A-Box) with the T-Box, and outputting inferences locally.

Thus, our notion of a separate T-Box, and restriction of our rules to those with zero or one assertional patterns allows us to flood the small T-Box to all machines, and avoids the need to compute potentially expensive A-Box joins across machines; that is to say, given the T-Box as global knowledge, the slave machines can perform reasoning over the large A-Box in an embarrassingly parallel fashion.

As before, in order to evaluate the benefit of distributing the reasoning process over multiple machines, in Table 6 we present the time taken for reasoning over ~31.3m distributed across 1, 2, 4 and 8 machines. The demonstrated scale is near-linear, with the common aggregation of T-Box information causing the non-linear factor.

<table>
<thead>
<tr>
<th>#</th>
<th>mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.9</td>
</tr>
<tr>
<td>2</td>
<td>11.3</td>
</tr>
<tr>
<td>4</td>
<td>6.5</td>
</tr>
<tr>
<td>8</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 6
Time taken for reasoning of ~31.3M statements for differing numbers of machines
Continuing the thread of the paper, we applied the above reasoning approach over the data (1.192b statements) output from the consolidation step as described in Section 5. The entire process took \( \sim 113 \) minutes: extraction of the T-Box took \( \sim 16 \) minutes; local aggregation of the T-Box – including application of T-Box rules, loading the T-Box into memory, and performing authoritative analysis – took \( \sim 14.2 \) minutes; application of the A-Box rules took roughly \( \sim 82.75 \) minutes – again the extra cost is attributable to writing inferred statements to disk. The reasoning produced 569m inferences – although this figure possibly includes duplicates.

The T-Box consisted of 2.039M statements (0.2% of the data), which represents a marked reduction from previous observations \([45]\).

In Figure 11, we give some indication as to the fecundity of each rule, giving an absolute count of triples inferred through that rule (where the rule was the last applied; that is, we do not consider recursion in the count), and also an indication of how triples inferred by certain rules overlap. Note that, for illustrative purposes and space limitations, we present only the rules which require A-Box information (Table A.3). To take an example, we see that the prp-rng rules produce 36.7% duplicate triples with respect to those already asserted in the data. We also see a correlation in overlap between rules which produce \texttt{rdf:type} triples, and non-\texttt{rdf:type} triples.

Further, in Table 7, we enumerate the top 5 T-Box only rules with respect to inferred statements produced.

### Table 7
The top 5 T-Box only rules wrt. number of inferred statements

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
<th>inferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>scm-cls</td>
<td>905,122</td>
</tr>
<tr>
<td>2</td>
<td>scm-op</td>
<td>301,546</td>
</tr>
<tr>
<td>3</td>
<td>scm-sco</td>
<td>217,106</td>
</tr>
<tr>
<td>4</td>
<td>scm-rng</td>
<td>174,435</td>
</tr>
<tr>
<td>5</td>
<td>scm-dp</td>
<td>108,964</td>
</tr>
</tbody>
</table>

### 7.8 Related Work

In this section we presented the extension of our earlier presented work on SAOR\([45]\) towards larger coverage of OWL2RL and parallel distribution of
inference. Similarly, other works have been presented that tackle large-scale reasoning by massive parallelisation: Urbani et. al. [71,57] presented a MapReduce approach to RDFS reasoning in a cluster of commodity hardware similar to ours. In fact, and following on from SAOR, they identified the distribution properties which the separation of terminological data offers.

Also following SAOR, and in a similar approach to that of [71], Weaver and Hendler [73] use a separation of terminological data and identify the parallel executability of rules with a low (zero or one) number of assertional patterns in the antecedent, identifying the entire RDFS ruleset as showing such characteristics.

Although both have demonstrated scale in the order of a billion triples, their experiments were based on synthetic datasets derived from the LUBM benchmark: they have not demonstrated the feasibility of reasoning over Web data. For example, both of these works do not consider our notion of authoritative, which we found to be a significant element in reducing the inferences to a reasonable volume on Web data, and avoiding a lot of noise otherwise inferred on real world, unclean data.

A viable alternative approach to Web reasoning – the relation to which is discussed in depth in [45] – is to consider a small “per-document” closure per Web graph only when doing inference over Web data, which is followed e.g., in Sindice [19] which is also parallelised over MapReduce.

For giving answers to search queries, we focus on rule-based inference in the
spirit of the novel OWL2RL fragment so far. Nonetheless, there are other interesting approaches tackling reasoning over the new tractable fragments of OWL DL (OWL QL, OWL EL) for which approximative, scalable techniques (e.g. [59,60]) are being devised – such could potentially be used on OWL Full Web knowledge bases for incomplete/approximate inferencing.

7.9 Future Directions and Open Research Questions

In order to make reasoning on Web data feasible – both in terms of scale and usefulness of the inferred data – we currently renounce a lot of inferences theoretically warranted by the OWL semantics. One of these renouncements is to treat authoritativeness as an absolute, thus only allowing inferences about TBox statements where they are published; another renouncement being that we do not further consider deriveable inconsistencies, by leaving out rules that produce so-called “clashes” (inference of falsity) [67,31].

In future works we aim to extend our approach to deal with those issues in a more fine-grained manner, combining ranks of objects, triples and their contexts (discussed in detail in Section 6) in inference, and investigating their applicability for consideration of the trustworthiness of inferences. We aim to explore how far we can define similar tailored rulesets that deploy fuzzy rule application [66] where a degree of trust is measured, and paraconsistent reasoning to get a more complete picture in inference. Naturally, this will push the boundaries for scalability even further than our current, cautious approach.

8 Indexing

To allow for speedy access to the RDF data we employ a set of indices. We employ a inverted index for lookups on RDF literals (text), and a sparse index for lookups on the RDF structure. Inverted indices are standard for keyword searches in information retrieval; we employ a sparse index because it represents a good trade-off between lookup performance and simplicity [36]. In principle, we could employ any sufficiently optimised implementation of an index structure that offers prefix lookup capabilities on keys.
8.1 Inverted Index

We first present performance results for the inverted index based on Lucene\footnote{http://lucene.apache.org/java/}. The Lucene index over the RDF data is constructed the following way:

- for each subject in the RDF graph, construct a Lucene document with the union of all string literals attached to the RDF subject;
- to each subject, add fields containing the identifier (URI or blank node) of the subject, labels (rdfs:label), descriptions (rdfs:comment and dc:description), classes (objects of rdf:type triples, and latitude and longitude values if available;
- in addition, globally computed ranks are added for each identifier.

For lookups, we specify a set of keyword terms for which matching identifiers should be returned, and in addition the desired slice of the result set (e.g. result 1 to 10). Lucene combines the globally computed ranks with query-dependent TF*IDF ranks and selects the slice of results to be returned. From the contents of the additional fields we generate an RDF graph and return the results to higher layers.

8.2 Structured Index

The structured index is implemented using “sparse indices” [36], where a blocked and sorted ISAM file contains the RDF quads. Lookups are supported by a small in-memory index which holds the first entry of each block. The in-memory index is search using binary search to locate the blocks which can potentially contribute to the answer. As a last step, the blocks are fetched and matching quads filtered out and returned.

There are two tuning parameters for such an index: the block size, which determines i) the size of the chunks of data fetched from disk and ii) indirectly the size of the in-memory portion of the index. The second parameter for tuning is compression: minimising the amount of data transferred from disk to memory should speed up lookups, provided that the time saved by smaller data transfers outweights the time required for uncompressing data. We tested a number of configurations for compression: no compression, gzip, zlib, and a combination of these with a simple encoding (labeled rle) where repeating URIs inside a block are encoded with an integer.
Returning top-k results for keyword searches

Fig. 12. Lookup time vs result size on log/log scale for top-k results (k = 10, 100, 1000) of popular keyword searches

8.3 Evaluation

For the inverted index, we selected the top 100 keyword searches users posed to the online SWSE system. Figure 12 plots the time elapsed versus result size in bytes on a log/log scale. Lucene returns results roughly in time linear to their size.

For the structure index, we tested lookups in various index configurations; from experiments we determined 8k blocks as competitive block size. We compared our index using different compression techniques against MySQL in version 5.0.51a. We created graphs consisting of 5m to 125m RDF quads using the random graph model proposed by Erdos-Renyi [26], and randomly created lookup keys which were evaluated against the index. Figure 13 shows the results; the figure includes results for the relational database to up to 8m statements - larger data would trigger OutOfMemory exceptions when performing lookups. Even for data sizes where MySQL returns results, our index organisation outperforms the relational database significantly.

8.5 Full-Scale Evaluation

In the final step of our pre-runtime evaluation, we must build the index over the 1.76b consolidated and reasoned statements. The entire process took 640 minutes (the most expensive component in our architecture to run bar the
crawler): the coordinate function hashing the data by subject and redistributing took 202 minutes (with an average idle time of 5.3 minutes); the index building – including the merge-sorting of the data – took 409 minutes (with an average idle time of 4 minutes); finally, the Lucene build took 38.8 minutes (with a relatively large average idle time of 5.7 minutes, attributable to the slow operation of one machine). The expensive methods here largely consists of the cost of re-organising all of the data over the network, and performing the merge-sort over the data.

8.6 Future Work and Open Research Questions

The future work of the indexing section is inextricably linked to that of the future direction of the query processing, and the user interface components. At the moment, our index support simple lookups for subjects matching a given keyword, and the quads for which that subject appears.

In previous work [36], we presented YARS2, which provided six indices (roughly equivalent to those presented) and an inverted-keyword index to support arbitrary SPARQL queries, and in earlier versions of SWSE, YARS2 served as the back-end. However, given the limited range of query types required to be serviced by the UI, the cost of building the full SPARQL index is difficult to justify – other than of course, e.g., to offer programmatic access to the underlying data.

Given a relatively static query model, a custom built index can be tailored to offer optimised service to the UI, as opposed to, e.g., a generic SPARQL engine. The main directions for future work in indexing would be to identify
8.4 Distributed Index Building

With respect to building the index in a distributed manner, we identify the following operations:

(1) **coordinate**: the slave machines hash triples from the consolidated data and the reasoned data according to the subject position – the data fragments are then sent to the appropriate peer and incoming data fragments are received from all peers;

(2) **run**: the slave machines perform a merge-sort of the data, producing a sorted quad index as described;

(3) **run**: the slave machines then build the inverted-keyword index over the sorted data.

Table 8 presents the results of applying the indexing procedure on 1, 2, 4, and 8 machines. Again, we observe a largely linear trend with respect to the number of machines: indeed, here our coordinate distributed function proves it’s worth, by avoiding the bottleneck of gathering/scattering on the master machine.

<table>
<thead>
<tr>
<th># of machines</th>
<th>Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>104.4</td>
</tr>
<tr>
<td>2</td>
<td>53.3</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>15.4</td>
</tr>
</tbody>
</table>

Table 8
Time taken for indexing of ~31.3m input statements and ~16.7m reasoned statements for differing numbers of machines

an intersection of queries for which optimised indexes can be built in a scalable manner, and queries which offer greater potential to the UI.

In summary: future research on indexing includes in-memory index structures to satisfy the requirements for interactive browsing and querying large amount of RDF data, appropriate compression techniques, and indices that support sophisticated join processing, (including top-k processing over arbitrary joins).

9 Query Processing

The goal of the query processor is to split the query and perform lookups in parallel on index machines. We thus use the master-machine to accept input queries from the user interface, which are then flooded to the slave machines. For a top-k keyword query, the co-ordinating machines requests result identifiers and ranks from each of the slave machines. The co-ordinating
Table 9
Total time taken for answering 100 popular keyword queries for differing numbers of machines

<table>
<thead>
<tr>
<th>#</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.16</td>
</tr>
<tr>
<td>2</td>
<td>20.99</td>
</tr>
<tr>
<td>4</td>
<td>18.79</td>
</tr>
<tr>
<td>8</td>
<td>18.01</td>
</tr>
</tbody>
</table>

machine then computes the aggregated top-k hits from the results returned; the co-ordinating machine then requests the full result data for each of the hits from the respective originating machines, and streams data to the initiating agent.

9.1 Evaluation

We have tested the performance of the query processing on a varying number of machines. Table 9 shows the aggregated query time for retrieving the top-100 results of 100 popular keyword searches. Figure 14 shows the query time for 1, 2, 4, and 8 machines as a function of result size. For small results sets, adding machines adds overhead and thus increases query time. That effect is reversed as result sizes get larger and multiple machines can work in parallel, thus decreasing overall query time despite the overhead for distribution.

Fig. 14. Query performance for 100 popular searches on 1, 2, 4, and 8 machines.
9.2 Future Directions and Open Research Question

Extending the query processing to handle more complex queries is a topic of importance when considering extension and improvement of the current Spartan UI. In order to fully realise the potential benefits of querying over structured data, we need to be able to perform optimised query processing. An open research question here is how to optimise for top-k querying in queries involving joins; joins at large-scale can potentially lead to the access of large-volumes of intermediary data, used to compute a final small results size; thus, the question is how top-k query processing can be used to immediately retrieve the best results for joins, allowing – e.g. – path queries in the UI (joins on objects and subject) such that large intermediate data volumes need not be accessed, and rather than the approach of joining several attribute restrictions (e.g. facets) as done in the threshold algorithm [27].

10 User and Application Programming Interface

In the following section we discuss a proof-of-concept user interface over the large corpus of enhanced and integrated data. In addition, we outline functionality that an application programming interface can provide for software programs accessing the corpus.

10.1 Searches and Results

End users require means to interrogate the underlying data and interact with the system, which can be separated into two phases: i) pose queries and ii) receive answers to these queries.

Queries posed to search engines can be classified into three groups [13]:

- informational: locate static information about a topic
- navigational: locate a particular web site
- transactional: shopping at, downloading from, or otherwise interacting with the result

A large fraction of searches are informational [64], and given that we do search over data which supports informational searches, we focus on that: locating and displaying static information about a topic.

We discuss several modalities which could be used for posing the query:
natural language: this is the most desirable way for posing queries (as shown in a user study [48]). However, there are inherent problems with this type of interaction [68].

keyword search: this is the commonly used way for posing queries; users type in a few keywords (typically less than three terms [38]) which describe their information need.

browsing: this is typically used for domain-specific collections of documents or data, where the individual data items are classified along specified dimensions and users point and click with a mouse to select categories or items.

The second step in a searching episode, receiving answers, can be also implemented in a number of ways:

- direct answer: this is the most desirable way, and would work well in conjunction with natural language query input.
- set of matching items: this is the currently employed way of returning results; search engines either present a list of ten links in case documents are returned, or a grid of images
- visualisation of matching items: show the things that come back in some sort of visual display, for example as a map or a calendar.

Given the variety of domains and ontologies on the Semantic Web, a user interface can provide some basic, domain-independent means of specifying queries and returning results; we chose to provide basic keyword lookups for objects to support informational searches which has been traditionally the largest class of queries for web search engines. The current prototype of the user interface for SWSE implements keyword searches over the enhanced and integrated dataset, allowing for users to search the unified contents of the Semantic Web; resulting objects are displayed in a paged list view. Both keyword searches for specifying the information need and results display in a list are generic and domain-independent.

Having available structured data rather than text documents, however, potentially allows for deploying a large class of user interaction models and visualisations on top of the data, provided the data is of sufficient quality. An application programming interface (API) can give third-party developers access to SWSE’s rich corpus who in turn provide tools operating on the data and utilising the corpus.
10.2 Application Programming Interface Outline

SWSE’s corpus consists of structured data which could power a large variety of visualisations, for example serve data to SIMILE’s timeline view\(^{19}\) or a Google map view\(^{20}\). Countless other visualisations are possible, for a history and examples of visualisations see e.g. [30].

For external applications accessing SWSE data the systems communicate via an API. There main requirement for an API for SWSE is performance: the API has to return results fast enough to enable interactive applications. The API should represent a solid foundation which guaranteed response times which enables application developers to build stable and scalable applications on top. Full SPARQL is likely too powerful (and hence too expensive to evaluate) for providing stable and fast responses, but a suitable subset (DESCRIBE queries and conjunctive queries with a limited number of joins) could serve as a foundation for visualisations and other applications leveraging integrated web data in SWSE.

10.3 Evaluation

To evaluate the performance of the user interface we compare the query response times of the API against the response times of the UI for the same set of queries. Figure 15 shows that the additional processing for rendering the user interface is minimal, and most of the queries return in less than 0.4 seconds.

10.4 Future Directions and Open Research Questions

Having collected and enhanced the data and made the integrated dataset available for query is common to all user interface models. The data quality of the underlying dataset directly affects the workings of any interface on top of the data; in general, data quality has to improve by either fixing and improving the original data sources, or find algorithms which detect the data publisher’s true intent in case of broken data.

Indexing and query processing heavily depends on the operations enabled in the user interface.

\(^{19}\)http://www.simile-widgets.org/timeline/

\(^{20}\)http://maps.google.com/
For searching and querying data, there is a trade-off between scalability of the approach and expressivity of the query language used. In general, joins are expensive operations, and when attempting to perform those on very large datasets either the system consumes a large amount of resources per query or becomes slow. Some systems (such as [46]) solve the scalability issue by partitioning the data sets into smaller units and have the user select a sub-dataset before further browsing or querying.

We have made first experiments with more expressive user interfaces for interacting with data (as opposed to just searching) with VisiNav 21, which allows for faceted browsing [74] and path traversal [4] on top of keyword searches and object focus operations. A major open question is how to balance expressive (database) query operations with fast (search engine) response times. Optimised indexing for data warehousing seems applicable at first, however, the structure of web data is much less uniform than relational data; as data from the web is very diverse, traditional database methods developed for data adhering to a regular fixed structure have to be adapted. Similarly, just providing keyword search functionality over rich data does not fully exploit the potential offered by richly structured data. Efforts to match keyword searches to structured queries (e.g. [69]) could bring together the ease of use of web search engines and the precise answers of structure data query, however, performance and ambiguity of poorly specified information needs remain challenges yet to be addressed.

21 http://visinav.deri.org/
11 Related Work

The Harvest [11] system is an infrastructure to provide decentralised caching facilities and to provide keyword search over a set of “object brokers” that go beyond the ability of looking up static resources by dereferencing a URI.

The approaches related to the term “Deep Web” heavily rely on manually constructed wrappers to extract data from HTML pages [15]. The bottleneck in these systems is that each web page has to be wrapped and translated into a structured representation, which requires a large amount of manual labour and thus does not scale well to cover a large number of sources. Google Base\textsuperscript{22} is an example of a site that allows people to post metadata content about data items. The site then provides basic query answering capabilities over the submitted dataset. While the site recently has added guided navigation features depending on the result set, Google Base still does not allow to navigate from item to item and thus exploit relationships between objects. In contrast, SWSE allows to traverse the data graph actively collected from a large number of sources, and thus exploits richly interlinked Web data.

We adapted the distributed architecture similar in spirit to what has been sketched in [12], [52], [40]. Scaling up these semantic systems represents the current challenge.

A more structured data model is the main driver for differences in the traditional search engine architecture to an engine working over structured data. The need to deal with structured data rather than documents affects each component in the architecture.

Early prototypes using the concepts of ontologies and semantics on the Web include Ontobroker [18] and SHOE [37], which can be seen as predecessors to standardisation efforts such as RDF and OWL. Swoogle [21] utilises an inverted keyword index on RDF documents, stored in a custom schema in a relational database. Swoogle calculates metrics that allow ontology designers to check the popularity of certain properties and classes. In contrast to Swoogle which is mainly concerned with ontologies, SWSE focuses on obtaining large amounts of instance data, which requires scalable distributed architecture SWSE is able to provide query functionality over the collected data graph, where Swoogle only allows to match RDF documents with keywords.

Moreover, Swoogle views the Semantic Web as a collection of documents or files and enables keyword search facilities over the files. The main distinction between web search engines (and Swoogle) and semantic search engines is the ability to perform queries over structured data involving joins. Therefore, we

\textsuperscript{22}http://base.google.com/
utilise methods from databases in the system, especially indexing and query processing, but also object consolidation: techniques known from data warehousing and data integration systems.

The recently developed Falcon Search\(^{23}\) offers entity-centric searching over RDF instances and classes based on Lucene and MySQL. Sindice\(^{24}\) is a registry and lookup service for RDF files based on Lucene. Watson\(^{25}\) is an effort to provide natural language search facilities over Semantic Web data.

12 Conclusion

In this paper, we have presented the results of research carried out as part of the SWSE project over the past five years. In particular, we have adapted the architecture of large-scale web search engines to the case of structured data. Research on how to integrate and interact with large amounts of data from a very diverse set of independent sources is fairly recent, as many characteristics of the research questions in the field became visible after the deployment of significant amounts of data by a host of data publishers. The traditional application development cycle for data-intensive applications is to model the data schema and build application on top - data modeling and application development are tightly coupled. That process is separated on the Semantic Web: data publisher just model and publish data, often with no particular application in mind. That approach leads to a chicken-egg problem: people do not have an incentive to publish data because there are no applications that would make use of the data; and developers do not create useful applications because of the lack of quality data. However recently there has been some success with Linked Data where an active community publishes datasets in a broad range of topics, and maintains and interlinks these datasets. We hope that SWSE makes a contribution – perhaps research oriented – towards fostering applications making use of the increasingly interlinked data. Based on the experiences collected we have identified open research questions which we believe should be solved in order to get closer to the vision of search over Semantic Web data discussed in the introduction.

Making web data available for querying and navigation has significant scientific and commercial potential. Firstly, the web, the largest artifact of human knowledge, becomes subject to scientific analysis\(^{8}\). Understanding the implicit connections and structure in the web data graph can help to reveal new understandings of collaboration patterns and the processes by which networks

\(^{23}\)http://iws.seu.edu.cn/services/falcons/
\(^{24}\)http://sindice.com/
\(^{25}\)http://watson.kmi.open.ac.uk/WatsonWUI/
form and evolve. Secondly, making sense out of scientific data published on the web can help scientists to gain insights for their research. The ability to navigate and search effectively through a store of knowledge integrated from thousands of sources can broaden the pool of information and ideas available to a scientific community. Thirdly, making the web data graph available for interactive querying, browsing, and navigation has applications in areas such as e-commerce and e-health, and has the potential to drastically improve the quality and utility of web search in general. Commercial success – and bringing a system such as the Semantic Web Search Engine into the mainstream – is perhaps a longer term goal, relying on the increased growth in RDF data becoming available: data which is of mainstream interest, and has a broad coverage of topics.

We have presented the architecture and implementation of a system that syntactically and semantically integrates information from a large number of data sources on the Web, leveraging RDF and Semantic Web methodologies to demonstrate the feasibility of bringing together data and corroborating information from multiple independent sources. We hope to continue researching in this field, extending upon the work presented herein, and tackling the discussed open research questions for the next few years to come.

References


## A Rule Tables

### \( R_0 : \text{no antecedent} \)

<table>
<thead>
<tr>
<th>OWL2RL</th>
<th>Consequent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>prp-ap</td>
<td>?ap a :AnnotationProperty .</td>
<td>For each built-in annotation property</td>
</tr>
<tr>
<td>cls-thing</td>
<td>:Thing a :Class .</td>
<td></td>
</tr>
<tr>
<td>cls-nothing</td>
<td>:Nothing a :Class .</td>
<td></td>
</tr>
<tr>
<td>dt-type1</td>
<td>?dt a rdfs:Datatype .</td>
<td>For each built-in datatype</td>
</tr>
<tr>
<td>dt-type2</td>
<td>?l a ?dt .</td>
<td>For all ?l in the value space of datatype ?dt</td>
</tr>
<tr>
<td>dt-eq</td>
<td>?l1 :sameAs ?l2 .</td>
<td>For all ?l1 and ?l2 with the same data value</td>
</tr>
<tr>
<td>dt-diff</td>
<td>?l1 :differentFrom ?l2 .</td>
<td>For all ?l1 and ?l2 with different data values</td>
</tr>
</tbody>
</table>

Table A.1
Rules with no antecedent
<table>
<thead>
<tr>
<th>OWL2RL</th>
<th>Antecedent (terminological)</th>
<th>Consequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls-00</td>
<td><code>?c : oneOf (?x_1 ... ?x_n)</code></td>
<td><code>?x_1 ... ?x_n a ?c</code></td>
</tr>
<tr>
<td>scm-cla</td>
<td><code>?c a :Class</code></td>
<td><code>?c rdfs:subClassOf ?c ; :Thing ; :equivalentClass ?c ; Nothing rdfs:subClassOf ?c</code></td>
</tr>
<tr>
<td>scm-sco</td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
<td><code>?c1 rdfs:subClassOf ?c_3</code></td>
</tr>
<tr>
<td>scm-eqc1</td>
<td><code>?c1 : equivalentClass ?c_2</code></td>
<td><code>?c2 rdfs:subClassOf ?c_1</code></td>
</tr>
<tr>
<td>scm-eqc2</td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
<td><code>?c2 rdfs:subClassOf ?c_1</code></td>
</tr>
<tr>
<td>scm-eqp1</td>
<td><code>?p1 : equivalentProperty ?p_2</code></td>
<td><code>?p2 rdfs:subPropertyOf ?p_1</code></td>
</tr>
<tr>
<td>scm-rng1</td>
<td><code>?p rdfs:range ?c_1</code></td>
<td><code>?p rdfs:range ?c_2</code></td>
</tr>
<tr>
<td>scm-hv</td>
<td><code>?c1 : hasValue ?i ; :onProperty ?p_1</code></td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
</tr>
<tr>
<td>scm-svf1</td>
<td><code>?c1 : someValuesFrom ?y_1 ; :onProperty ?p</code></td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
</tr>
<tr>
<td>scm-svf2</td>
<td><code>?c1 : someValuesFrom ?y ; :onProperty ?p_1</code></td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
</tr>
<tr>
<td>scm-avf1</td>
<td><code>?c1 : allValuesFrom ?y_1 ; :onProperty ?p</code></td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
</tr>
<tr>
<td>scm-avf2</td>
<td><code>?c1 : allValuesFrom ?y ; :onProperty ?p_1</code></td>
<td><code>?c1 rdfs:subClassOf ?c_2</code></td>
</tr>
<tr>
<td>scm-int</td>
<td><code>?c : intersectionOf (?c_1 ... ?c_n)</code></td>
<td><code>?c rdfs:subClassOf ?c_1 ... ?c_n</code></td>
</tr>
<tr>
<td>scm-uni</td>
<td><code>?c : unionOf (?c_1 ... ?c_n)</code></td>
<td><code>?c rdfs:subClassOf ?c</code></td>
</tr>
</tbody>
</table>

Table A.2
Only terminological antecedent patterns
### $\mathcal{R}3$ : at least one terminological/only one assertional pattern in antecedent

<table>
<thead>
<tr>
<th>OWL2RL</th>
<th>Antecedent</th>
<th>Consequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls-int2</td>
<td>?c :intersectionOf (?c1 ... ?cₙ) .</td>
<td>?x a ?c , ?x a ?c1...?cₙ .</td>
</tr>
</tbody>
</table>

**Table A.3**

At least one terminological and exactly one assertional pattern
B  Selected Dataset Statistics

Fig. B.1. Class distribution, and top 5 classes.

Fig. B.2. Property distribution, and top 5 properties.

Fig. B.3. Distribution of class/property namespaces, and top 5 namespaces.