MultiCrawler: A Pipelined Architecture for Crawling and Indexing Semantic Web Data

Andreas Harth, Jürgen Umbrich, and Stefan Decker

National University of Ireland, Galway
Digital Enterprise Research Institute
firstname.lastname@deri.org

Abstract. The goal of the work presented in this paper is to obtain large amounts of semistructured data from the web. Harvesting semistructured data is a prerequisite to enabling large-scale query answering over web sources. We contrast our approach to conventional web crawlers, and describe and evaluate a five-step pipelined architecture to crawl and index data from both the traditional and the Semantic Web.

1 Introduction

The enormous success of Google and similar search engines for the HTML web has demonstrated the value of both crawling and indexing HTML documents.

However, recently more and more information in structured formats such as XHTML, microformats, DC, RSS, Podcast, Atom, WSDL, FOAF, RDF/A etc. has become available – and we expect this trend to continue. In conjunction with Semantic Web based RDF data, these data formats are poorly handled by current search engines: for instance, query answering based on keywords does not allow to exploit the semantics inherent to structured content. Consequently, current well developed and understood web crawling and indexing techniques are not directly applicable, since they focus almost exclusively on text indexing.

In other words, to be able to answer queries which exploit the semantics of Semantic Web sources, different crawling and indexing techniques compared to conventional search engines are necessary. The differences between conventional crawling/indexing approaches and crawling/indexing heterogeneous semantic data sources can be summarized as follows:

1. *URI extraction.* HTML crawlers extract links from HTML pages in order to find additional sources to crawl. This mechanism usually does not work as straightforwardly for structured sources, since very often there exists no direct concept of a hyperlink. Therefore different methods for extracting URIs must be found.
2. *Indexing.* Conventional text indexes for the HTML web are well understood. However, these text indexes perform poorly at capturing the structure and semantics of heterogeneous sources, e.g., a FOAF file or an RSS source. A different way for indexing and integrating the various data formats is needed.

These two key differences illustrate the need for new approaches compared to traditional web crawling. A pipelined document indexing infrastructure has already been defined and analyzed (see [8]). However, the same approach is not applicable for Semantic Web data due to the variety of stages and different time and space behavior.

The main contributions of this paper are:

- Following the general approach of [8] we define a pipelined approach for the Semantic Web with respect to structured data crawling and indexing. The pipeline can be adapted to arbitrary content.
- We define a general URI extraction method from structured sources that helps to find more sources for indexing.
- We describe a general representation format for heterogeneous data on the web which allows indexing and answering of expressive queries.
- We describe an implementation of our pipelined architecture and determine the optimal configuration of the entire pipeline to maximize parallel processing and crawling.
- We evaluate the pipeline by conducting experiments on a cluster.

The remainder of this paper is organized as follows: In Section 2 we give an overview of the architecture. Section 3 describes the processing pipeline in detail, including complexity analysis and experimental results derived from each individual phase. In Section 4, we analyze the results, discuss tradeoffs for distributing the pipeline to multiple machines and running multiple pipelines in parallel. Section 5 covers related work and Section 6 concludes the paper.

2 Crawler and Indexer Architecture

When designing a crawler and indexer architecture a number of requirements need to be taken into account:

- *Performance and scalability.* The architecture needs to be as performance oriented as possible in order to handle data on a web-scale and keep up with the increase in structured data sources. The system should scale up by adding new hardware – without a fundamental redesign.
- *Utilizing data from different formats and disparate sources.* The system has to syntactically transform and index data from different web sources to arrive at an integrated dataset.

Text indexing software pipelines have been investigated by [8] as a means to optimize and decouple the crawling and indexing process. The pipelined architecture in [8] has lead to considerable performance improvements. We have adopted the pipelined architecture and defined a *software pipeline* for Semantic Web data crawling and indexing. The idea behind a software pipeline is to improve performance by executing different steps concurrently.

Fig. 1. Five phases for crawling and indexing Semantic Web data.

Our crawling algorithm is an adaption of the standard breadth-first search algorithm. Najork and Wiener [10] argue that breadth-first crawling yields high-quality pages early on in the crawling process.

The process of crawling and indexing Semantic Web data can be logically split into 5 phases, as illustrated in Figure 1. We refer to these phases as *fetch*, *detect*, *transform*, *index*, and *extract*. During the fetch phase, the information is fetched from the web. The detect phase detects the type of the content, eg. RDF, WSDL, GIF etc. The transform phase is a key difference compared to conventional text indexing and translates the data into the common data format. The index phase builds an index, which is used during the extract phase to query for URIs to more information sources.

We provide a rationale for some of the different phases in more detail.

Detect A challenge in dealing with multiple data formats is to be able to accurately detect the content type and format of documents. Most of the data formats can be detected by using the file extension or the content-type returned with the header part of an HTTP request. In the case of XML files, the MIME type and the file extension give indication for XML content, but do not give any information about whether the content is well-formed, or which schema is used. Sometimes this information is important, therefore the content itself has to be investigated.

Transform Since we are aiming at a general indexing and querying infrastructure we need mechanisms to extract information from the files and transform them to a structured representation. Ideally, we would like to use a declarative transformation language so that users can define transformations without the need to write code in a procedural language. However, the system should be also able to use procedural language code to extract data from binary data or natural language text, ultimately arriving at a representation of the metadata.

To describe transformations in a declarative way, we decided to use XSL Transformations (XSLT)[1]. With XSLT we are able to translate arbitrary XML content to RDF. Even though XSLT is Turing complete [5] and therefore might be too expressive, using XSLT has the benefit of permitting the reuse of already available stylesheets. Besides, it is possible to integrate GRDDL[2], a recent effort which aims at standardizing the mechanism of using XSLT to extract information from web pages.

Index An index over the data can be used to extract links and finally perform searches and answer queries. The index should enable keyword-based searches because that is a good method to explore a dataset with unknown structure.

[1] `http://www.w3.org/TR/xslt`
[2] `http://www.w3.org/TeamSubmission/grddl/`

Equally important we require an index on the graph structure for the ability to pose structured queries.

Extract For extracting URIs, we decided to use an RDF query against the final cleaned and structured dataset. We perform URI extraction at the end of the pipeline, since at that stage the indexes over a uniform representation of the data have been built already and we are able to extract URIs cheaply. Depending on the crawling strategy (only crawl one site, perform shallow crawling and only take external links into account, etc), we can adapt queries to extract URIs. We need to extract links also from HTML pages, otherwise we will not discover the URI of structured pages, since files with structured data are currently not well interlinked. URIs to structured sources appear mainly in `a href` links within HTML documents.

To be able to scale, we need to parallelize and distribute the system. Fetching the data takes much less time than processing. Thus, we want to perform steps in parallel, which means we have to use multiple threads that fetch data and multiple threads that process data etc. Communication between the steps is done via queues. If we want to scale up the process even further, we replace threads with multiple computers, queues with remote/persistent queues, and pipes with network data transfer. As a result, we are able to speed-up the entire process even more. Besides, in the distributed setup it is easy to identify bottlenecks – and resolve them by adding new machines to a phase. Another benefit of a distributed architecture is that it facilitates the integration of external components (i.e., web services) into the process.

Our goal is to analyze the complexity of the single tasks and to find the right balance in server ratios to keep the average utilization of the servers as high as possible. In the next section we describe each processing step, investigate the complexity and present experimental measurements.

3 Processing Pipeline

In this section, we describe each step in the processing pipeline in detail. The processing pipeline is composed of five different modules, each of which is capable of running the task in a multi-threaded fashion. First, the fetching module downloads the content and header information of a web page. Second, the detecting module determines the file type of the web page. Third, based on the file type, the transformation module converts the original content into RDF. Fourth, the indexing module constructs an index over the RDF data to enable URI extraction. Fifth, the extracting module poses a query over the index to extract URIs and feeds the resulting URIs back into the pipeline.

To be able to pass parameters between different phases, the system needs to store information associated with the URIs. We put the metadata associated with a URI as RDF triples in a metadata store which runs on a separate machine.

Each phase has an associated queue which contains URIs of pages to be processed. Each phase takes a URI from the queue, retrieves the content from

a previous phase if necessary, processes the content, stores the content on disk, and puts the URI into the queue corresponding to the next step in the pipeline. Content is passed to successive steps via an Apache HTTP server.

In the following sections, we include complexity analysis and experimental results for each step. We carried out the experiments using a random sample of 100k URIs obtained from the Open Directory Project[3]. We performed all experiments on nodes with a single Opteron 2.2 GHz CPU, 4 GB of main memory and two SATA 160GB disks. The machines were interconnected via a 1GBbp network adapter on a switched Ethernet network.

3.1 Fetching Data

The functionality of the fetching module includes obtaining a new URI from the queue, checking for a `robots.txt` file to adhere to the Robots Exclusion Protocol[4], and fetching and storing header information and content.

After obtaining the next URI from the queue, we retrieve the `robots.txt` information for the host either from the metadata store or directly from the host. Then we determine if the fetcher is allowed to crawl the page or not. If the URI passes the check, we look at the content length provided by the header information. To avoid downloading very large files we compare the content-length from the header-field with a given file size threshold.

If the URI passes all these checks, we connect to the web server and download the content of the page. Then we store or update the header information on the metadata store. Finally, we send the URI to the next module in the pipeline and return to the beginning, to poll the next URI from the queue.

To provide an estimate of the complexity of the step, let N be the size of the documents fetched, including header information. The fetch step needs to transfer N bytes from the Internet, which takes linear time in the size of the content, O(N).

We verified the complexity analysis experimentally. We chose randomly 100k URIs from ODP's collection of over 5M sites. Figure 2 shows the experimental results for the crawling component resulting in 78038 downloaded pages (1.4 GBytes of data). The fetching component achieved an average download rate of around 600 KBytes/sec.

3.2 Detecting File Types

The detecting module tries to determine the exact content type of the data, which is used in the transformation phase to execute the right transformation module. The type detection is based on the information we are able to derive from the URI, the header fields and the content of the page itself.

In the first step of the file type detection process we try to detect the content type based on the file extension of the URI. The second step retrieves the content-type header field from the metadata store and compares the header field to a list

[3] `http://dmoz.org/`

[4] `http://www.robotstxt.org/wc/exclusion.html`

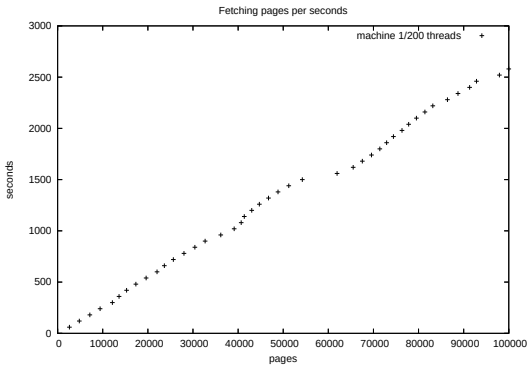

Fig. 2. Experimental results derived from crawling 100k randomly selected URIs.

of content types. Table 1 lists all supported content types and the information the system needs to detect them. If one of these checks successfully detects a type, we can stop the process and store the type on the metadata repository. In case of XML content, we perform another check to figure out the schema of this XML file. In this case we must parse the content itself.

TypeID	RFC	MIME media type	File extension	Root element
HTML	2854	text/html	.html .htm	html
XHMTL	3236	application/xhtml+xml	.xhtml	`xhtml:html`
XML	3023	text/xml application/xml	.xml	-
RSS2.0	-	application/rss+xml	.rss	rss
Atom	4287	application/atom+xml	.atom	`atom:feed`
RDF	3870	application/rdf+xml	.rdf	`rdf:RDF`

Table 1. File types the system is currently able to handle.

If we detect XML content, we try to find out the special type of the XML content, that is, we retrieve the content data from the file system and parse it with a SAX XML parser. We try to extract namespaces and root element of the XML file and compare the values to the known content types. If all checks fail, we assume an unknown or unsupported content type. Finally, we store the type on the metadata store and forward the URI to the next pipeline module.

During the complexity analysis, we do not consider the simplest case where we can detect the file type based on file extension or header information. Let N be the size of the XML document which content type we want to detect. Parsing the XML content utilizing SAX to retrieve the root element has a time complexity of O(N).

Figure 3 shows the experimental results for the file type detection phase.

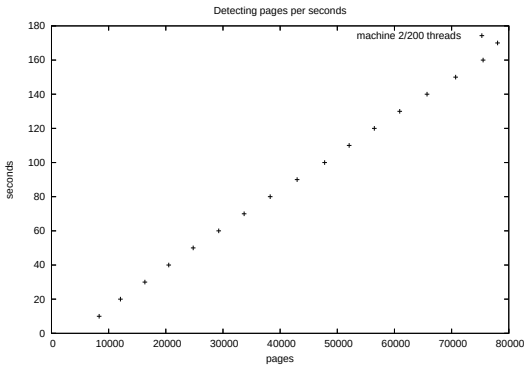

Fig. 3. Experimental results for detecting file types.

3.3 Transforming to RDF

For transforming the content into the common data format RDF, the system applies different transformation modules depending on the type of the content. The transformation phase can be split into two steps: (i) conversion from non-XML data, such as HTML, into XML by using user specified transformation tools and (ii) transformation of XML data to RDF via XSLTs and xsltproc[5]. For a given URI we retrieve the content type, which has been added in the detect phase, from the metadata store. Depending on the result of the query, we execute different transformation modules on the content. Naturally, if the data format is already RDF, we can skip the transforming step.

To transform non-XML data, we can call out to external services which convert the data directly into XML or RDF. At the moment our support for non-XML data consists only of cleaning up HTML using the tool Tidy[6] running as a cgi-bin on a HTTP server, but various external services for extracting metadata from e.g. image or video files can be easily plugged in.

To transform XML data, we use xsltproc with an XSLT from the file system, depending on the type identifier of the page. We use an XSLT that transforms RSS 2.0 and Atom to RDF[7]. We also developed an XSLT[8] which transforms XHTML pages into an RDF representation based on RDFS, DC, and FOAF vocabularies. In this stylesheet we extract from a HTML document the following information: title, email addresses, images, and relative and absolute links and their anchor labels.

After the URI passes successfully all transformation steps, we pass it to next step of the pipeline.

[5] `http://xmlsoft.org/XSLT/`
[6] `http://tidy.sourceforge.net/`
[7] `http://www.guha.com/rss2rdf/`
[8] `http://sw.deri.org/2006/05/transform/xhtml2rdf.xsl`

The worst case scenario when performing the transforming step is in dealing with HTML documents, because we must first pass the content to Tidy and then perform the XSLT transformation. Imagine a document of size N and a XSLT stylesheet of size M. We assume Tidy takes time linear to the size of the content O(N). The worst-case complexity for XPATH has shown to be $O(N^4 * M^2)$ [5], however, for a large fragment called Core XPath the complexity is O(N*M). Our XHTML XSLT uses only simple Core XPath queries, therefore the worst-case complexity for the step is O(N*M).

Figure 4 shows the experimental results for the transformation component utilizing the `xhtml2rdf.xsl` and `rss2rdf.xsl` stylesheets. Using 200 threads as in all other tests, the transformation performance decreased rapidly after around 13k pages because the machine was assigned with too many transformation tasks and had to swap. Therefore we plotted only the first 60 minutes of running time. We repeated the tests with only 50 threads to not overload the machine. In the end, the transformation step yields 907Mbytes of XHTML resulting in 385Mbytes RDF/XML.

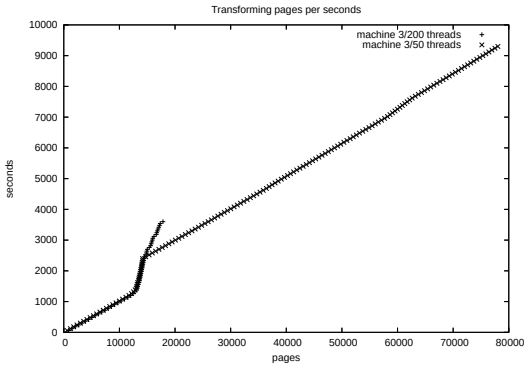

Fig. 4. Experimental transformation performance using 50 and 200 threads.

3.4 Building Indexes

We summarize the index organization and building process here. For a more detailed description of the index organization we refer the interested reader to a previous paper [6]. Please observe that we operate on an extension of the RDF data model which includes the notion of context to store the provenance of RDF triples. Tracking provenance is achieved by adding a fourth field and therefore using quadruples.

The goal of the index structure is to support efficient evaluation of *select-project-join* queries. The selection operation enables the retrieval of quads, given any combination of subject, predicate, object, and context. To be able to perform

the quad retrieval with just one index lookup, the index organization uses a complete index on quads which covers all 16 possible access patterns on quadruples. Conceptually, we have (key, value) pairs stored in a B+ tree, which allows to perform lookups – especially prefix and range lookups – on keys. We also use an inverted index on string literals to allow to search the index via keyword-based searches.

The index structure contains two sets of indexes: the *Lexicon* covers the string representation of the graph, and the *Quad Index* covers the quads. The Lexicon maps values of resources and literals to objects identifiers (OIDs) using two B+ tree indexes for node/OID mapping. In addition we employ an inverted index for string literals. The quad index covers the triples of the graph plus context. We use concatenated keys on all combinations of subject, predicate, object, and context and therefore are able to retrieve any combination with a single index lookup without performing joins.

When the indexer receives a quad for indexing, it first performs lookups for each element of the quad in the Lexicon to either retrieve its OID or assign a new OID. New OIDs are assigned monotonically for each new quad element. In case the element is a string literal, we include the string literal in the inverted index. Next, the keys for the quad are constructed based on the OIDs of the individual elements of a quad. Given our index organization with concatenated keys and prefix lookups, we only need six indexes to cover all 16 quad patterns [6]. In total, given our index organization, there are 6 keys for insertion into the 6 indexes.

The two indexes mapping from quad element values and back are implemented in Berkeley DB JE[9]. Additionally, we store string literals in Apache Lucene[10] for textual search. The quad indexes are maintained in Berkeley DB as well, with one index acting as the primary index and five secondary indexes, to implement a complete index on quadruples.

Since index construction is technically involved, we will describe the time complexity in more detail. Let N be the size of the input in RDF/NTRIPLES, N_L the number of Lexicon entries, N_K the number of words per Lexicon entry, M_L the order of the Lexicon B trees, N_T the number of quadruples and M_T the order of the B+ tree with respect to the quads. First, the system performs OID lookups/assignments in the Lexicon which is largely determined by the input size of the data $O(N * 4 * 2 * log_{M_L} N_L)$, next creates a text index in Lucene over the newly added string literals which takes $O(N_L * N_K)$ time, and finally adds the quads into the respective B+ trees $O(N * 6 * log_{M_T} N_T)$.

Figure 5 shows the experimental results for constructing the index on the 26906 pages that were transformed without errors resulting in 76.3MBytes of data in RDF/NTRIPLES format (and a total of 571915 triples).

[9] `http://www.sleepycat.com/products/bdbje.html`
[10] `http://lucene.apache.org/`

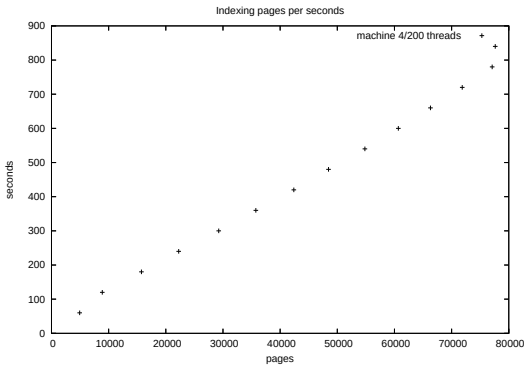

Fig. 5. Performance for indexing the syntactically integrated dataset.

3.5 Extracting URIs

To feed the processing pipeline with new URIs we have to extract URIs from the indexed content, which is done in the extracting module. The process can be divided into two steps. The first step is to extract URIs from the data and the second step is to filter the URIs to make sure only URIs matching specified criteria get processed.

To extract new URIs we execute a query on the index for typical link predicates such as `rdfs:seeAlso` and `rss:link`. We are able to perform conjunctive queries, which are evaluated by translating a N3QL[11] query expression to a relational algebra expression to an executable query plan.

If an extracted URI is to be added to the queue, we pass this URI through the installed filter. In this filter we can restrict which URI should be sent to the fetching module. If we want to crawl only a domain or a set of domains, we can filter the addition of URIs using regular expressions. These expressions are stored in memory. It is also possible to add new expressions during the runtime to the filter.

The main functionality for the link extraction phase is the processing of (conjunctive) queries utilizing the index. Let N_L the number of Lexicon entries, M_L the order of the Lexicon B+ trees, N_T the number of quadruples, M_T the order of the B tree with the quads, M the number of conjuncts in the query, and R the result size. We first sort the conjuncts starting with the conjunct which contains the least number of variables taking $O(MlogM)$ time, then detect the join conditions (similar to union-find) $O(MlgM/2)$, translate the elements of the quads to OIDs which can be done in $O(4 * M * log_{M_L} N_L)$, perform the selections on the index and index nested loops joins, $O(log_{M_T} {N_T}^M)$, and finally translate the resulting OIDs to element values, which takes $O(R * log_{M_L} N_L)$.

Figure 6 shows the experimental results for the extraction component. We discuss the results of all phases in the next section.

[11] `http://www.w3.org/DesignIssues/N3QL.html`

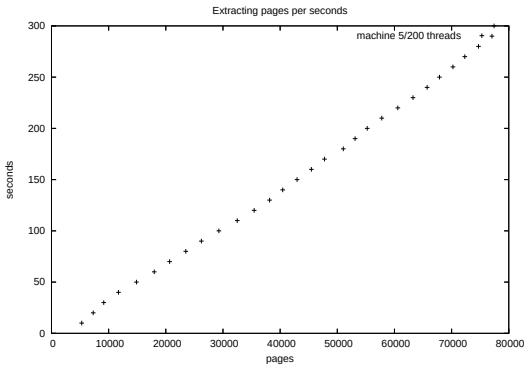

Fig. 6. Experimental link extraction performance.

4 Analysis and Tradeoffs

In the following we analyze the performance results for the five phases of the pipeline process and discuss two questions: i) how to distribute the individual phases to remove potential bottlenecks and fully utilize the processing power of each machine and ii) how to run multiple pipelines in parallel to achieve a throughput of the total system which can be calculated by: *number of pipelines * pipeline throughput*.

Currently, the transform phase represents the bottleneck in the pipeline and can only process a fraction of the pages delivered by the fetch and detect phase. The random sample of URIs are biased towards HTML data, which means that during the transform phase almost every page has to be processed. If we are able to reduce the amount of HTML and XML sources and increase the amount of RDF sources, the transform phase has to process less pages and as a result the throughput (in terms of time per page) increases. However, given the fact that the majority of content on the web is in HTML format, we have to distribute the transform component to achieve acceptable performance.

Assuming an architecture as described in this paper, we can distribute phases by just adding more machines. Pages are assigned to nodes using a hash function. In initial experiments we observed that we can scale up the fetch step by a constant factor if we add more fetcher machines and all fetcher nodes take URIs concurrently from the queue. The case is a bit different for the transform step; here, we employ one thread pool with individual threads which retrieve a URI from the previous step in the pipeline and invoke Tidy and XSLT operations on cgi-bins running on a web server. In other words, while the other phases employ a pull model, inside the transform component tasks are pushed to external processors. We chose the push model because the ability to include external transformation services was a requirement.

Figure 7 shows the performance results where all steps and external processors run on one node, where one node was used for the steps and two nodes

for external processors (1+2), and the case where four external processors (1+4) were used. Why was the scale-up not constant in the number of machines added? The reason is that the hash function assigns the pages equally to the external processors. In case a single page takes a very long time to process, the external processor node cannot keep up with the assigned operations and at some point in time needs to swap, which leads to a decrease in performance.

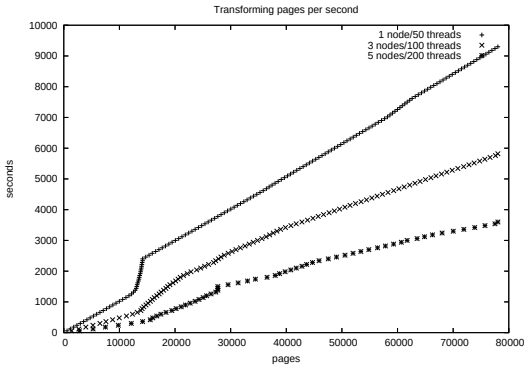

Fig. 7. Performance measurement for the transform phase with 1, 3 (1+2), and 5 (1+4) transform nodes.

Apart from the issues described for the transform steps, we claim all other steps can be scaled by a constant factor (number of machines added) using a hash function to distribute URIs to nodes since URIs in those phases can be processed independently. Table 2 shows the throughput in pages per second for each phase, and a ratio that determines which fraction of the stream (assuming that the fastest component determines the throughput) one node can process.

Phase	Servers	Pages/sec	Ratio.
fetch	1	38	0.082
detect	1	460	1.0
transform	1	5	0.011
transform	1+2	13	0.028
transform	1+4	21	0.045
index	1	92	0.2
extract	1	260	0.565

Table 2. The number of servers and the achieved performance. Ratio is calculated based on the fasted phase (1 = 460 pages/sec).

To be able to scale up the system even further, we can just employ more pipelines and achieve a total throughput which can be calculated by multiplying the number of pipelines with the throughput achieved on one pipeline. The limit is then only determined by how many resources (Internet bandwidth and number of machines) are available.

5 Related Work

There are two types of related work to our framework: the first consist of large scale web crawling and indexing systems, and the second are systems extracting information from semistructured sources.

Crawler frameworks such as UbiCrawler [2] or Mercator [7] are focused on the performance of the crawling step only. Google [3] handles HTML and some link structure. We focus less on crawling but on detecting Semantic Web data, the transformation of XHTML and XML to RDF and the indexing.

A few efforts have been undertaken to extract structured content from web pages, but these efforts differ considerably in scale. Fetch Technologies' wrapper generation framework[12] and Lixto [1] are examples of commercially available information extraction tools. Lixto defines a full-fledged visual editor for creating transformation programs in their own transformation language, whereas we use XSLT as transformation language and focus on large-scale processing of data. Fetch (similarly [9]) combine wrapper generation and a virtual integration approach, whereas we use a data warehousing approach and therefore need scalable index structures.

SemTag (Semantic Annotations with TAP) [4] perform mostly text analysis on documents, albeit on a very large scale. In contrast, we extract structured information from documents and XML sources, and combine the information with RDF files available on the web.

6 Conclusion

We have presented a distributed system for syntactically integrating a large amount of web content. The steps involved are crawling the web pages, transforming the content into a directed labelled graph, constructing an index over the resulting graph, and extracting URIs that are fed back into the pipeline. We have shown both theoretical complexity and experimental performance of the five-step pipeline. We are currently working on performing a long-term continuous crawl and testing the system on larger datasets.

Acknowledgements

We thank Hak Lae Kim for discussing various requirements related to RSS crawling and Brian Davis for commenting on an earlier draft of this paper. This work

[12] `http://www.fetch.com/`

is supported by Science Foundation Ireland (SFI) under the DERI-Lion project (SFI/02/CE1/l131). We gratefully acknowledge an SFI Equipment Supplement Award.

References

1. R. Baumgartner, S. Flesca, and G. Gottlob. Visual Web Information Extraction with Lixto. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 119–128, September 2001.
2. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: a Scalable Fully Distributed Web Crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.
3. S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7):107–117, 1998.
4. S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. SemTag and Seeker: Bootstrapping the Semantic Web via Automated Semantic Annotation. In *Proceedings of the Twelfth International World Wide Web Conference*, pages 178–186, May 2003.
5. G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *Journal of the ACM*, 52(2):284–335, 2005.
6. A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the 3rd Latin American Web Congress*, pages 71–80. IEEE, 2005.
7. A. Heydon and M. Najork. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4):219–229, 1999.
8. S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-Text Index for the Web. In *Proceedings of the 10th International World Wide Web Conference*, pages 396–406, 2001.
9. M. Michalowski, J. L. Ambite, S. Thakkar, R. Tuchinda, C. A. Knoblock, and S. Minton. Retrieving and Semantically Integrating Heterogeneous Data from the Web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.
10. M. Najork and J. L. Wiener. Breadth-First Crawling Yields High-Quality Pages. In *Proceedings of the Tenth International World Wide Web Conference*, pages 114–118, May 2001.