# Distributed Web2.0 Crawling for Ontology Evolution

**Andreas Juffinger, Thomas Neidhart**

Institute for Knowledge Management
Graz University of Technology
E-mail: andreas.juffinger@tugraz.at, thomas.neidhart@tugraz.at

**Michael Granitzer, Roman Kern**

Knowledge Discovery Division
Knowcenter, Graz
E-mail: mgrani@know-center.at, rkern@know-center.at

**Arno Scharl**

Department of New Media Technology
MODUL University Vienna
E-mail: scharl@modul.ac.at

**Abstract:** The World Wide Web as a social network reflects changes of interest in certain domains. Furthermore it has been shown that free online content in Blogs, Wikis, News and Forums is a valuable source of information to identify trends in certain domains. Utilizing this data one can construct ontologies describing the information and getting an abstract, semantically correct overview of a domain. Tracked over time this also enables a user to identify trends and hypes. The decentralised structure of the Internet, the huge amount of data and upcoming Web2.0 technologies arise several challenges for a crawling system for ontology learning, evolution and trend analysis. In this paper we present a distributed crawling system with browser integration for Web2.0. The proposed crawler is a high performance, browser content equivalent, Web data retrieval system aimed to retrieve and prepare textual Web content for ontology learning.

## 1 Introduction

In the past different types of crawling systems have been developed to serve different kinds of information retrieval and text mining system demands. The ontology learning and evolution process is addicted to extremly high quality text

input. Web content on the other hand is usually of poor quality and contaminated with navigational elements, advertisements and other noise elements. New Web2.0 technologies additionally complicate the process of crawling high quality content because the content is often produced on the client side, with javascript sometimes even from different Web servers. A standard crawling system, which simply loads data from servers, is not sufficient for this task - for Web2.0 crawling one must interprete the scripts within an HTML document and compile a DOM tree, which is equivalent to the information a human Web surfer would experience and also one has to cut off navigational elements, advertisements and other noise. The reason, why the extracted text has to be of high quality in our ontology learning framework, becomes clear when one takes a closer look at our methodology.

The rest of the introduction explains in more detail ontology learning and our methodology, and therefore motivates the importancy of a crawling system with noise reduction. In the next Section we discuss then aspects of the history of Web crawling and the categorisation of crawlers, before we propose our system in Section 3. The evaluation of the noise reduction method is then discussed in Section 4 and finally we conclude in Section 5.

### *1.1 Ontology Learning and Evolution*

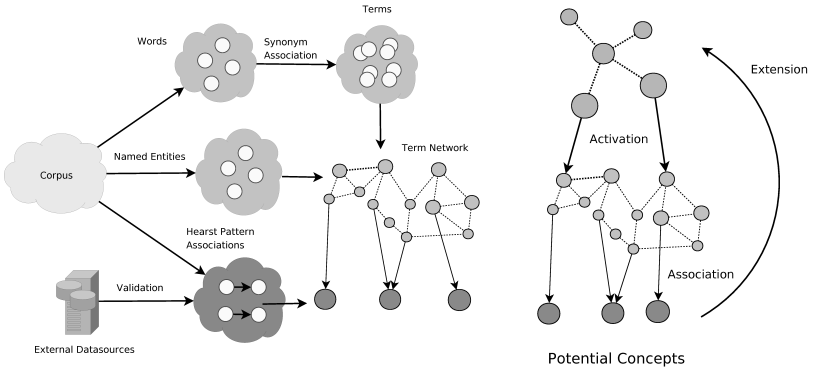
According to Buitelaar et al., 2005 and other authors Maedche, 2003; Staab and Studer, 2005, ontology learning is concerned with knowledge acquisition from data. Ontology learning from text is based on natural language processing, machine learning and artificial intelligence, and does overlap with knowledge acquisition due to the similar aim of extracting explicit knowledge implicitly contained in textual data. Ontology learning and population often follows the incremental growing paradigma, as outlined in Maedche, 2003. In our work we have to deal with the growing process on two different time scales. On one hand the initial domain ontology creation and on the other hand the evolution over time. For evolution one uses the ontology from the last evolution iteration and evaluates each concept and relation for presence in the actual data. New concepts are inserted and non present concepts and relations are deleted. Each operation is a single step towards the evolved ontology. Note that we deal exclusively with ontologies, where the changes are not reflected *in the ontology* with appropriate classes, concepts and relations. We consider ontology evolution in the sense of ontology versioning as in Klein and Fensel, 2001, where each single change triggers a new ontology variant or version.

### *1.2 Ontology Learning Methodology*

Our ontology learning and evolution methodology is sketched in Fig.1. Starting from tokenized text, we collapse tokens into terms and create from these terms a term network based on co-occurence analysis in plain text. Utilizing Hearst Patterns we detect associations between possible concepts and/or instances, Fig.1 (left). These detected associations are then attached to existing terms within the term network and labeled as possible concept candidates. With the use of spreading activation and injection of energy through the seed/earlier ontology into the term network, Fig.1 (right), we are able to calculate an activation field and rank the

concept candidates. The most relevant candidates are then proposed to the user as most significant extension to the existing ontology.

**Figure 1** Ontology Learning Methodology based on Term Network, Named Entities and Hearst Pattern




(a) Term Network Generation (b) Ontology Extension

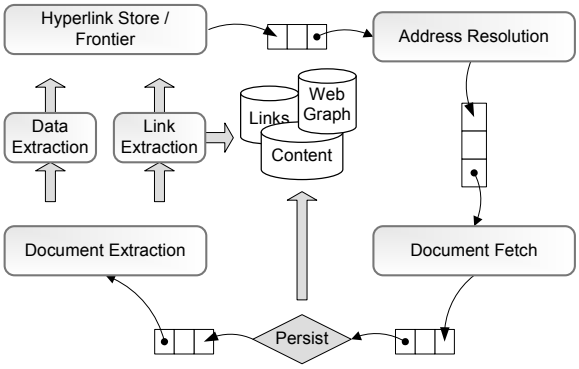
The term network, our basis for ontology extension, comes from the plain text analysis of the underlying corpus. In the context of this work this is the data crawled from the Web. Due to the fact that the term network is constructed from co-occurence it is important to minimize the impact of standard Web content to avoid relations rooting in a co-occurence of a person named in the page content and the navigation terms “user”, “admin”, or “home”. A similar pollution of the term network arises from advertisements, where wrong associations would appear and the system would associate Google with nearly every concept occuring in pages with Google AdSense elements.

## 2 Web Crawling History and Classification

A crawler, also known as wanderer, robot, or spider has the task of retrieving documents, extracting links and then fetching the newly discovered documents. The principal crawling loop is sketched in Fig.2: Starting from a collection $S_{t=0}$ of hyperlinks in the frontier, the crawler selects one link $s \in S_t$, resolves the hostname to an IP address, establishes a connection and fetches the document using a specific transfer protocol. The retrieved document is then preprocessed and parsed by a mime type specific parser. The discovered links $S'$ are then added to the frontier $S_t \cup S' \rightarrow S_{t+1}$. A crawler performs this loop until some kind of higher target is reached, or it runs out of links. Crawling the Web requires enormous amounts of hardware and network resources.

Web search engines gain their documents usually from some crawling systems. Because of this the first crawling systems have been invented by search engine developers. The first Web search engine “Wandex” used an index collected by the World Wide Web Wanderer a web crawler developed by Matthew Gray at MIT in 1993. Shortly after the RBSE Spider, WebCrawler and the Lycos WebBots have been developed Eichmann. (1994); Pinkerton. (1994); Zeinalipour-Yazti and Dikaiakos. (2001). Clearly, all major search engines on the web have highly optimised

**Figure 2** The Principal of a Crawling Loop: Address resolution, page download, link extraction, and URL selection for iteration




crawling systems, but they do not publish all details, therefore not much information from the big players is published. Furthermore, research in the field of crawling has focused on crawling strategies, which attempt to minimise the number of pages that need to be downloaded, thus not much research has been made in the area of highperformance crawling. Building a high-performance system that can download hundreds of millions of pages over several weeks presents a number of challenges in system design, I/O and network efficiency. Cho and Garcia-Molina, 2002 outlined the most important problems one is faced with within parallel crawling as:

- Overlap: Multiple parallel crawler processes have to be careful about the in-between overlap. Multiple downloads of individual documents should be minimised to save bandwidth, and increase the effectiveness. Therefore the problem is how to find a coordination strategy to prevent overlap.
- Quality: Usually a crawler wants to download "important" pages first, to improve the quality of the collection. The relevance is dependent on the overall collection, and thus a technique to predict importance throughout the crawling system is needed.
- Communication Bandwidth: In order to fulfil the first two points communication is needed within the crawling system, therefore, how to minimise this communication without impact on overlap and quality is an important factor, which cannot be overlooked.

In addition to the aforementioned, Shkapenyuk and Suel. (2001) outlined a number of requirements for high-performance crawling, whereby scalability, robustness and manageability are the major points.

A distributed crawler is formed by a group of *n local crawlers*, which are distributed over different geographical locations and connected together by the Internet. Each of the local crawlers may in itself be a multithreaded program, maintaining its own frontier and other data structures. A local crawler may even be a parallel program executing on a cluster of workstations, with centralised control and a global frontier. Distributed crawling systems follow to some degree the "divide and conquer" principle from parallel computing, by partitioning the Web and

assigning portions, either statically or dynamically, to different local crawlers. In this way each one of them focuses on a distinct subset of the Web URIs. Note that this is usually not enough, because different URIs might point to the same page, or a site might be mirrored to another site. Thus collaboration is also needed to identify and deal with duplicate content Heydon and Najork (1999); Broder et al. (1997); Chung and Clarke (2002) There are various techniques used to partition the Web for distributed crawling, and these have been listed by Cho and Garcia-Molina, 2002 as follows:

- Independent: Local crawlers act totally independent without any kind of coordination. Each crawler is initialised with a different seed set, and then no further coordination takes place. Thus the overlap is thereby significantly dependent on the seed set.
- Dynamic Partitioning: A central coordinator divides the Web into partitions according to a specific partitioning function. Pages are then dynamically assigned to the different local crawlers. Although this technique can guarantee no overlap it is very communication extensive, which might be a problem.
- Static Partitioning: The Web is partitioned statically before crawling takes place, so that each local crawler knows the portion of the Web it is responsible for. Further Cho and Garcia-Molina categorised the partitioning functions as: (a) URL-Hash: The URLs are partitioned by their URL fingerprint, ignoring site locality, and thus a huge number of inter-partition links occur. (b) Site-Hash: Instead of hashing the whole URL, only the site specific parts are hashed and the Web is partitioned according to this fingerprint, thereby minimising the inter-partition links. A drawback of this method is the fact that the site size follows a Power-Law. This leads to the problem of workload balancing in-between local crawlers. (c) Hierarchical: Utilising the hierarchical structure of hostnames, the Web can be partitioned by Top Level Domains, and also by countries. The hierarchical structure in the IP addresses can also be used to partition the sites.

Boldi et al., 2001 designed the UBICrawler, originally called Trovatore (2004), with the intent to create an efficient, distributed and highly scalable crawling system. The UBICrawler is self-stabilising in the classical sense, thus the system is able to react on adding/removing/crashing of local crawlers at runtime. Brin and Page, 1998 developed the initial version of the current GoogleBOT. They proposed a distributed crawling system with a single URL server, thus a dynamic partitioning system, and a number of crawlers (all implemented in Python). Mercator is a crawling system developed by Najork and Heydon, 2001 (Compaq, HP and CDLR Group) which now crawls the Web for the AltaVista search engine. The system can be classified as a static assignment distributed crawling system, whereby the partitioning technique is based on Site-Hash.

## 3 Web 2.0 Crawling

One of the major content types on the Web is of course HTML. A HTML document furthermore is often build up from multiple other sources which are referenced

or embedded. A browser is able to render an HTML document, and also to resolve and download external objects, such as style sheet definitions, images, javascripts and flash objects. According to O'Reilly, 2000, 2005, the Web 2.0 applications are web sites, and no longer desktop applications, or enterprise software systems. Furthermore he stated: *And once you start thinking of web sites as applications, you soon come to realize that they represent an entirely new breed, something you might call an "information application", or perhaps even "infoware".* Anderson, 2007 states that the key applications and services for Web2.0 are: Blogs, Wikis, Tagging and social bookmarking, file sharing, podcasting, RSS and syndication. Altough each of this application or service is of high interrest within IR, we focus in this paper on Blogs, Wikis and RSS. New Web2.0 technologies and frameworks like AJAX (Asynchronous Javascript + XML) and Flash based technologies are a big step towards the idea of *Infoware* and *The Web as Platform*. These technologies tackle the user demand on rich Web platforms with minimal download cost and delays. An AJAX application loads only small amount of content at once. After the initial page download only portions of the webpage are dynamically reloaded in real-time, what creates the impression of richer, more "natural" applications with the kind of responsive interfaces that are commonly found in desktop applications (Anderson, 2007).

Infoware sites as well as Web sites based on content management systems (CMS) contain many elements for navigational and auxiliary purposes. Often they are based on so called page templates. All pages, conform to a common template, share a number of content elements. These duplicate content parts do not carry as much information as other parts, altough some of the links, especially navigation links, are of primary interrest for a crawling system. Yossef and Rajagopalan, 2002 have been able to improve the precision of crawled text content for similarity based queries by removing template content by 5-10%. Yi et al., 2003 proposed a different technique, for template detection, based on a kind of DOM Tree, a so called Site Style Tree (SST). They have been able to improve the precision by another 5-10% in comparison to the method of Yossef and Rajagopalan, but at the cost of more computational complexity. Applying these methods to Web2.0 web sites allows one to pinpoint the two main demands for a modern Web2.0 crawling system:

- Noise reduction: (a) for content, so that text mining is not confused with advertisements and navigational elements and (b) for hyperlinks, to minimise the number of irrelevant page downloads.
- Executed Extraction: to ensure to process user visible data and not only source code which produces the visible content.

Additionally, for ontology learning and evolution, one has to fullfill another demand: Repeated high performance crawling to ensure consistency of the ontology in a changing world.

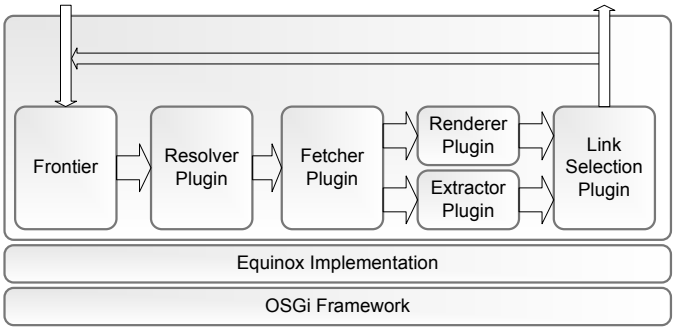
Web2.0 technologies, like AJAX, which enrich the user experience of a Web site in a browser, are a stumbling block for automatic Web content retrieval systems. As described above, these technologies are capable to reload and display fractions of Web pages, what makes it nearly impossible to crawl the page content without executing the embedded Javascript. Our approach allows us to overcome the Web2.0 complexity, and to ensure that the crawled content is the same as a user would experience it. We tackle the above three demands with two independent techniques.

Firstly, utilising a standard browser will ensure point two, together with heuristical DOM serialization we also can meet demand one for repeated crawling. Secondly, to achieve a high performance crawling system we are doing distributed crawling on a heterogenouse workstation cluster. The reminder of this section discusses our system architecture and those two approaches in more details.

### *3.1 System Architecture*

We have implemented each part of the principal crawling loop, as shown in Fig.3 as separate OSGi Plugin, a dynamic plugin system for Java. Furthermore our system is based Equinox, a implementation of the OSGi R4 core framework specification. The crawling system loads, dynamically at startup, available plugins and validates them. For each main step in the crawling loop, there must be at least one plugin which implements this step. Each plugin acts as a filter part within the overall architecture and has a input filter attached. The input filter decides then, based on metadata, if the plugin should process, skip or ignore the pipe element. For example the input filter for the renderer plugin accepts only elements tagged with "must render", whereby the extractor plugin ignores such elements to avoid duplicate processing.

**Figure 3** The Crawling Sytem Architecture based on OSGi Modul Framework




### *3.2 Browser Rendering and DOM Serialization*

Our local crawlers follow the principal of a crawling loop, as shown in Fig.2. We have implemented an asynchron crawler in Java, whereby every block works independent with a configurable number of threads. The communication inbetween the blocks is implemented according to the pipes and filters design pattern (Gamma et al., 1995). As main improvement to the standard crawling loop, we introduced a renderer pluging after the document fetching block. The document fetching plugin is, a multi threaded implementation of the HTTP protokoll to download as fast as possible the main content. The fetched documents are then classified into a "must render" and "dont render" based on the amount and type of javascript. Pages which fall into the first class are then tagged with "must render " and delivered to the later plugins. The input filter of the renderer plugin selects then the tagged elements and the filter for the extractor plugin allows only the other elements.

Our renderer plugin is based on SWT Embedded Mozilla and XPCOM, the Cross-Platform Component Object Model. XPCOM provides a framework which manages the creation, ownership, and deletion of objects and other data throughout Mozilla and the according Java interfaces. The Mozilla browser runs as a SWT widget and is fully controllable through the Mozilla Java API. This fully integrated renderer has the advantage, over the earlier used proxy approach based on selenium, that we can access and manipulate the displayed page, as well as we have now direct access to the document object model tree in the browser. The renderer processes documents as follows: The browser content is set to the already fetched data, whereby we ensure a HTML base tag to allow the browser to resolve relative, external links. The browser widget is then responsible to render the document and to load external style sheets and javascript when this is necessary. Due to the fact that the browser has an internal cache and that we ensure that pages from the same site are always delivered to the same browser instance, the browser loads only once the necessary external javascript libraries and stylesheets. When the browser has finished the page, an event is fired and we serialize the DOM tree from the browser. We have implemented a selective serializer which is able to serialize site specific subparts of the DOM tree, what allows us to cut off site specific advertisement sub trees and other noise elements.

The additional computational costs are manageable, and often negligible compared to the fetching time of an document. We experienced on a standard workstation that the time needed to render a document, and to execute all additional scripts, is about ten times smaller than the time it takes to load the page from the server. The two main drawbacks of this approach are the additional requirements on computational power, and the higher degree on memory complexity. In the following part we outline our approach to overcome these problems.

### *3.3 Distributed Crawling*

To be able to crawl thousands of sites within a short time frame, and to allow complex content processing, it is necessary to distribute the crawling process on a cluster. We distribute the crawling process at our institute utilising a perfect fitting programming model: MapReduce (Dean and Ghemawat, 2004). The MapReduce paradigm is two-phase, firstly a map phase, whereby each map task processes a single input split element, and generates intermediate results. Secondly, the reduce phase combines all the intermediate results into the combined result. Applied to the crawling process, an external master splits the input URL list, and assigns each split to a single crawling mapper. Each crawling mapper task processes then the assigned URLs, and stores the result in a central persistent storage. If all crawling mapper have finished, the reducer combines the extracted links into a single URL list. Exactly this crawling process is implemented by the open source project Nutch[a].

In a first step, we used exactly this approach and we figured out some drawbacks of this direct mapping approach: (1) Overlap according to Cho and Garcia-Molina, 2002 is not avoidable, because the independent mapping tasks are not informed about the redirection or mirroring of other mapping tasks. (2) Static non influen-

[a] http://lucene.apache.org/nutch/

cable partitioning technique which supports drawback one and splits the input list into parts of same size. Input splits of same size in a heterogenouse workstation cluster leads to different load factor. For each round, all machines have to wait till the slowest one has finished processing. Therefore faster machines are under loaded and slower ones over loaded. The whole cluster would be able to process the full data in less time. (3) Statefull site crawling is not supported, and (4) Unintended rapid fire requests and denial of service attacks to certain hosts.

To overcome these drawbacks we have implemented a different partitioning and crawling strategy. First of all we do use a PostgreSQL database system as central data and link storage. For partitioning we implemented dynamic partitioning, based on a combination of site hash and IP address. The mapping of IP address to domain names and vice versa is M:N, and therefore a combination is necessary to ensure, that sites on the same server always fall into the same partition. Initially each local crawler is responsible for exactly five sites, which are then crawled in parallel to ensure request distribution over different servers. An important part of each local crawler is the implementation of a guarded queue, as proposed by Najork and Heydon, 2001 to avoid rapid fire requests. The local crawler starts one browser instance per site, and loads pages in the browser instance according to the scheduling defined in the guarded queue. From the resulting DOM tree in the browser, we extract on one hand the content elements and on the other hand the navigational parts. All links are then extracted from both parts, and stored in the central link storage. Site local links are automatically added to the guarded queue. Due to the fact that one browser instance is used to crawl all pages of one site, we have been able to implicitly do statefull crawling. As soon as the site queue runs out of links the local crawler queries the central link storage for site links found by other local crawlers. Whenever a site is completely crawled the according thread retrieves another site from the central store. This simple approach eliminates the drawback of different computational power in a heterogenouse cluster, and allows furthermore dynamic partitioning.

## 4 Evaluation

Due to the fact that the ontology learning and ontology evolution process is semi automatic, and therefore the outcome is dependent on the user and other non deterministic factors, we are not able to evaluate the overall system performance. For the Web2.0 crawling system, which is also applicable for other applications we evaluated the noise reduction performance. The outcome of the crawling system is a huge amount on plain text what makes manual evaluation not applicable. To evaluate our noise reduction method we used variable ranking as outlined in Guyon and Elisseeff 2003. The scoring function is the TFIDF (term frequency - inverse document frequency) weighting value from Information Retrieval. For the evaluation we used a crawl with about 2000 documents, for which we had filtered and unfiltered content available. The corpus contained the filtered and unfiltered version, and had therefore a overall size of about 4000 documents. We have then calculated the TFIDF weight for each term in the corpus. Assuming each document pair as a cluster we then calculated the cosine distance of each filtered document to each unfiltered document. The assumption was that we only cutted irrelevant

noise from the documents and therefore the distance between unfiltered and filtered document should be smaller as the distance to each other document. In other words, would it be possible to identify the original document when we only have the filtered document. In our experiment this was true for 94% of all filtered documents, what leads to the interpretation that our noise reduction method works quite well. Of course, this would be perfectly true for simply not filtering the document so we also evaluated the amount of filtering. The current system cutted off about 37% of the terms leading to an recognition error of 6%.

## 5 Conclusion

Acquiring knowledge is essential in any knowledge management system, especially for knowledge management system, which aim to track trends or changes, such as an ontology evolution system. In this paper, we described a Web crawling system, which is able to deliver high quality, noise reduced accurate data found on the internet. The system is further able to take a snapshot of the relevant fraction within a short time to ensure time consistency across a single crawl.

## Acknowledgment

The project results have been developed in the AVALON (Acquisition and Validation of Ontologies) project. AVALON is financed by the Austrian Research Promotion Agency (http://www.ffg.at) within the strategic objective FIT-IT under the project contract number 810803 (http://kmi.tugraz.at/avalon).

## References

Anderson, P. (2007). What is web 2.0? ideas, technologies and implications for education. Technical report, JISC Technology and Standards Watch.

Boldi, P., Cadenotti, B., Santini, M., and Vigna., S. (2004). Ubicrawler: A scalable fully distributed web crawler. In *Software: Practice and Experience*.

Boldi, P., Codenotti, B., Santini, M., and Vigna, S. (2001). Trovatore: Towards a highly scalable distributed web crawler. In *Proc. of 10th World Wide Web Conference*.

Brin, S. and Page, L. (1998). The anatomy of a large-scale web search engine. *Computer Networks*, 30:107–117.

Broder, Z., Glassman, S., and Manasse, M. (1997). Syntactic clustering of the web. In *Proc. of 6th WWW Conference*.

Buitelaar, P., Cimiano, P., and Magnini, B. (2005). *Ontology Learning from Text: Methods, Evaluation and Applications*. IOS Press.

Cho, J. and Garcia-Molina, H. (2002). Parallel crawlers. In *Proc. of the 11th WWW Conference,*.

Chung, C. and Clarke, C. (2002). Topic-oriented collaborative crawling. In *Proc. of 11th Conference on Information and Knowledge Management.*

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation,.*

Eichmann., D. (1994). The rbse spider - balancing effective search against web load. In *Proc. of the 1st Int. WWW Conference.*

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research.*

Heydon, A. and Najork, M. (1999). A scalable extensible web crawler. In *Proc. of WWW Conference.*

Klein, M. and Fensel, D. (2001). Ontology Versioning on the Semantic Web. In *Proc. of the Int. Semantic Web Working Symposium (SWWS), Stanford University.*

Maedche, A. (2003). *Ontology Learning for the Semantic Web.* Kluwer Academic Publishers.

Najork, M. and Heydon, A. (2001). High-performance web crawling. Technical report, Compaq System Research Center.

O'Reilly, T. (2000). *Open Sources: Voices from the Open Source Revolution.* O'Reilly.

O'Reilly, T. (2005). What is web 2.0. O'Reilly Media, Inc.

Pinkerton., B. (1994). Webcrawler. In *Proc. of the 2nd WWW Conference.*

Shkapenyuk, V. and Suel., T. (2001). Desing and implementation of a high-performance distributed web crawler. Technical report, Polytechnic University Brooklyn, NY.

Staab, S. and Studer, R. (2005). *Handbook on Ontologies.* Springer.

Yi, L., Liu, B., and Li, X. (2003). Eliminating noisy information in web pages for data mining. In *Proc. of the IEEE Intl. Conf. on Data Engineering.*

Yossef, Z. and Rajagopalan, S. (2002). Template detection via data mining and its applications. In *Proc. of the 11th international conference on World Wide Web.*

Zeinalipour-Yazti, D. and Dikaiakos., M. (2001). High-performance crawling and filtering in java. Technical report, Department of Computer Science, University Cyprus.