# Distributed Web2.0 Crawling for Ontology Evolution

Andreas Juffinger
and Thomas Neidhart
Graz University of Technology
8010 Graz, Austria
{ajuffing,tneidhart}@tugraz.at

Michael Granitzer
and Roman Kern
Know-Center
8010 Graz, Austria
{mgrani,rkern}@know-center.at

Albert Weichselbraun
and Gerhard Wohlgenannt
Vienna University of Economics
1090 Vienna, Austria
{aweichse,wohlg}@wu-wien.ac.at

## Abstract

*Semantic Web technologies in general and ontology-based approaches in particular are considered the foundation for the next generation of information services. While ontologies enable software agents to exchange knowledge and information in a standardised, intelligent manner, describing todays vast amount of information in terms of ontological knowledge and to track the evolution of such ontologies remains a challenge.*

*In this paper we describe Web2.0 crawling for ontology evolution. The World Wide Web, or Web for short, is due its evolutionary properties and social network characteristics a perfect fitting data source to evolve an ontology. The decentralised structure of the Internet, the huge amount of data and upcoming Web2.0 technologies arise several challenges for a crawling system. In this paper we present a distributed crawling system with standard browser integration. The proposed system is a high performance, sitescript based noise reducing crawler which loads standard browser equivalent content from Web2.0 resources. Furthermore we describe the integration of this spider into our ontology evolution framework.*

## 1. Introduction

The Web as a huge, freely available source of multimedial content, gatheres increased momentum as data source for ontology research. Different attempts to structure the Web data such as directory services like Yahoo! [25] and DMOZ OpenDirectory [11] also provide a human edited categorisation of fractions of the Web. These taxonomies are to some degree upper level ontologies on its own what makes them a good starting point to retrieve an adequate amount on human categorised data. Due to the fact that text mining techniques and machine learning approaches depend on sufficient data these methods depend on automatic document retrieval. Different types of crawling systems have been developed to serve different kinds of information retrieval and text mining system demands. In the context of this paper we differ the following two:(1) crawling for initial content and (2) repeated crawling for ontology extension and evolution. The decentralised structure of the Internet, the huge amount of data and upcoming Web2.0 technologies arise several challenges for a crawling system. The ontology evolution framework can only then produce interpretable and meaningfull results as long as the underlying data is of high quality. Nowadays Web pages are often stucked with advertisement and many navigational elements. New Web2.0 technologies additionally complicate the process of crawling 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 longer sufficient - for Web2.0 crawling one must execute the scripts within an HTML document to compile an DOM tree which is similar to the one a human Web surfer would experience.

This paper is structered as follows: First we give an introduction to general crawling techniques and distributed crawling followed by a part discussing arising Web2.0 technologies and their impact on crawling. The last part in the introduction describes the specific demands for a crawling system in the context of ontology evolution. The second section concentrates on our approach for Web2.0 crawling and then we outline our ontology evolution framework.

### 1.1. Web Crawling Basics

A crawler, also known as wanderer, robot, spider has the task of retrieving documents, extracting links and then fetching the newly discovered documents. The principal crawling loop is sketched in Fig.1: 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.

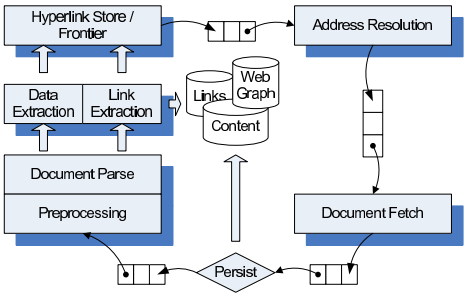


**Figure 1. The Principal of a Crawling Loop**

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 [12, 22, 28]. Clearly, all major search engines on the web have highly optimised 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 [8] outlined the most important problems one is faced with within parallel crawling as: (A) 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. (B) 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. and (C) 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 [23] outlined a number of requirements for high-performance crawling, whereby scalability, robustness and manageability are the major points. Scalability, usually attained by using distributed systems [8, 23], as a single crawler cannot achieve the required download rate through asynchronous I/O or multithreading. Robustness is another basic point, due to the fact that the system has to deal with a vast number of bad HTML pages, spider traps and congestion problems. Manageability is also necessary to identify problems during a crawl, as well as to stop and restart a crawl, and adjust the crawl speed.

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 [7, 9, 14]

There are various techniques used to partition the Web for distributed crawling and these have been listed by Cho and Garcia-Molina [8] as follows: (1) 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. (2) 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. (3) 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. [5] designed the UBICrawler, originally called Trovatore [4], 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. Brien and Page [6] 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 Phyton). Each crawler typically held 300 HTTP connections open, and they implemented a local DNS cache for each crawler to maximise throughput. Mercator is a crawling system developed by Najork and Heydon [17] (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.

### 1.2. Web 2.0 Applications and Content

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 Tim O'Reilly [19, 20] the Web 2.0 applications are no longer desktop productivity applications or even a back-office enterprise software systems, but individual web sites. 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".*

Infoware Web sites contain many elements for navigational and auxiliary purposes. Often they are based on so called page templates, for instance any kind of content management system (CMS) is based on templates or style sheets. Since all pages, conform to a common template, share a number of contents, it is clear that these duplicate content parts do not carry as much information as other parts, altough some of the links, especially menu links, are of primary interrest for an crawling system. Yossef and Rajagopalan [27] have been able to improve the precision for similarity based content queries by removing template content by 5-10%. They stated further that removing noisy links can also improve link based analysis methods like HITS [15] or PageRank [21]. Yi et al. [26] 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 computational complexity. Anderson [1] states that the key applications and services for Web2.0 are: Blogs, Wikis, Tagging and social bookmarking, Multimedia Sharing, Audio blogging and 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 *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 [1].

### 1.3. Crawling for Ontology Evolution

According to Stojanovic [24] ontology evolution is defined as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artefacts. In difference to other work we focus on the quantitative evolution process. We do not aim to model the evolution in the ontology but we aim to evaluate the differences within independent ontology snapshots at different time points. The Web as main external resource is continually evolving and therefore a perfect resource for ontology evolution. Because of the enormerouse amount on news sites, discussion forums and wikis we believe that the Web is one of the best resources for such an ontology evolution framework. From this we can pinpoint the two main demands for a crawling system for ontology evolution:(1) 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. (2) Repeated high performance crawling to ensure consistency according to certain time constraints. If we take into account the upcoming technolo-

gies with Web2.0 another important point has to be added to those two: (3) Extraction of user visible content to ensure to process the full data a standard user would be able to see.

## 2. Web 2.0 Crawling

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 three, together with Sitescripts 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 those two approaches in more details and our persistence model.

### 2.1. Browser Rendering and Script Execution

Our local crawlers follow the principal of a crawling loop as shown in Fig.1. We have implemented an asynchron crawler in Java whereby every block works completely asynchron with a configurable number of threads. The communication inbetween the blocks is implemented according to the pipes and filters design pattern [13]. As main improvement to the standard crawling loop we extended the document fetching block with standard browser rendering. The intend here is to ensure that the content persisted is equivalent to the document object model (DOM) tree as a browser would show it to a user. For this we utilise the OpenQA-Selenium Test Tool [18] which allows one to progammatically control standard Web browsers like Internet Explorer, Mozilla and Firefox.

The embedded browser, currently we use Firefox, fetches according to the user profile the page content without multimedia objects. After this the browser runs all the Javascripts which are defined to be executed at page load and downloads all necessary subcontent. The resulting DOM tree is then persisted.

The Selenium Tool is able to inject arbitrary external Javascript what allows us to add additional functionality to Web sites. For repeated crawling of a predefined set of Web sites we define Sitescripts, Javascripts specific to a site, which manipulate the DOM tree of all pages on the site in a way to be able to separate the navigational elements from content and to suppress most advertisement elements.

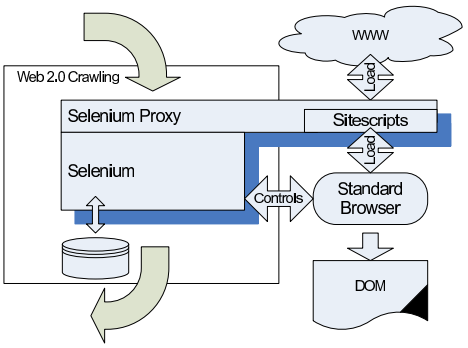


**Figure 2. Selenium Integration and Browser Control**

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.

### 2.2. 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 [10]. 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 [3]. Note that Nutch is based on Hadoop [2], another open source project which implements the MapReduce programming paradigm.

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 [8] is not avoidable because the independent mapping tasks are not informed about the redirection or mirroring of other map-

ping tasks. (2) Static non influencable 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 more details see section 2.3. 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 Narjok and Haydon [17] 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.

### 2.3. Content and Link Storage

The most important requirement for a crawling system storage is the possibility to handle a fast amount of data. Our two building blocks for the storage backend are distinguished by data type: Firstly, the content storage which handles the raw and processed data of pages. The result of the document parse and data extraction phase within the crawling loop is also persisted to avoid duplicate processing of the same data.

Secondly, the link storage is optimised for URL and IP address storage. We do store the IP addresses of each resolved domain to reduce the strain on the address resolution unit in a later crawl iteration. Furthermore we have achieved a reduction on the amount of stored data by storing only relative links and a reference to the according domain. The caching and duplicate detection module is responsible to reduce the number of database queries and to inform the local crawlers of so called mirrors. For URL duplicate detection we use two different hash values, one for the domain name or server string (including port specification) and one for the relative path on the server (including the query). Note that most of the duplicates are already eliminated by the local crawlers based on the second hash value. The remaining cross links are then checked by the duplicate detection module.

Ontology Evolution introduces a time dependency as addition to the above demands. The time when a specific content has been crawled is crucial for the evolution process. For the moment we track the following date and time information: Crawled, discovered for the first time, last changes and last time discovered. Whereby the last three state the most important timepoints in the live cycle of a page and the first is necessary to relativate the others.

## 3. Ontology Evolution Framework

As mentioned in section 1.3 we focus on the quantitative evolution process. Therefore we evaluate the differences within independent ontology snapshots at different time points. The most important step is semi automatic ontology extension at a certain time. Our semiautomatic ontology extension loop is shown in Fig.3: Starting from a time consistent crawl we extract a semantic network based on the extracted concepts and relationships. Dependent on the relationship type, we calculate an activation field with spreading activation on weighted graphs similar as in Liu and Weichselbraun [16]. This activation field is then used to rank the concepts. The concepts of highest rank are then positioned in the ontology and shown to the domain experts for validation. Thereby we allow the experts to select inbetween two modes: Firstly, one iteration one validation and secondly the $N$ iteration and one validation mode. This overall process is repeated until the domain experts judges the ontology as good according to their personal domain knowledge.

The crawling system is responsible to allocate the necessary amount of high quality data. We differ thereby two crawling phases:(1) Initial content development and domain specific content retrieval (2) Content update and identification of added and deleted content elements. In the initial phase we start from a predefined set of seed urls and perform full site crawls. To extend the seed set we further crawl

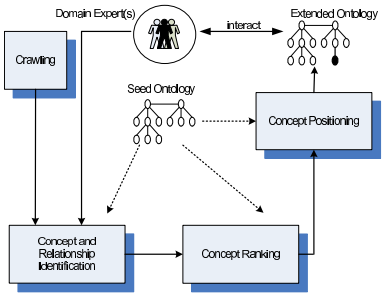


**Figure 3. Ontology Extension Loop**

the most often linked sites which have not been in the seed set. After this we use the content part plus meta information of each page as text input for the ontology evolution framework. In the update phase, we recrawl the sites developed in the initial phase, on predefined time intervals and validate their actuality. New, changed and deleted pages are then stored with the new timestamp in the database and the earlier described ontology extension step is repeated with exclusevily the data valid at that timepoint. The resulting ontologies represent the conceptual knowledge at certain time points and we are currently working on metrics to compare, match and evaluate these ontologies.

## 4. 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.

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

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

[2] Apache Lucene - Hadoop. Hadoop: Open source MapReduce Framework http://lucene.apache.org/hadoop/, 2007.

[3] Apache Lucene - Nutch. Nutch: Open source Web Search Software http://lucene.apache.org/nutch/, 2007.

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

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

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

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

[8] J. Cho and H. G.-M. . Parallel crawlers. In *Proc. of the 11th WWW Conference,*, 2002.

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

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

[11] DMOZ OpenDirectory. Human Edited Directory of the Web: http://www.dmoz.org/.

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

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

[14] A. Heydon and M. Najork. A scalable extensible web crawler. In *Proc. of WWW Conference*, 1999.

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

[16] W. Liu and A. Weichselbraun. Semi-automatic ontology extension using spreading activation. *Journal of Universal Knowledge Management*, 2005.

[17] M. Najork and A. Heydon. High-performance web crawling. Technical report, Compaq System Research Center, 2001.

[18] OpenQA Selenium. Browser Remote Control: Test Tool for WebApplications http://www.openqa.org/selenium/, 2007.

[19] T. O'Reilly. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, 2000.

[20] T. O'Reilly. What is web 2.0. O'Reilly Media, Inc., 2005.

[21] L. Page, S. Brin, R. Motwani, and T.Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[22] B. Pinkerton. Webcrawler. In *Proc. of the 2nd WWW Conference*, 1994.

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

[24] L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe, 2004.

[25] Yahoo! Yahoo! Directory: http://dir.yahoo.com/.

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

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

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