Volume 2, Issue 4, April 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Design of a Parallel Migrating Web Crawler

Abhinna Agarwal, Durgesh Singh, Anubhav Kedia
Akash Pandey, Vikas Goel
CSE,AKGEC,Gzb
India.
abkedia@live.com

Abstract: **As the size of the Web grows exponentially, crawling the web using parallel crawlers poses certain drawbacks such as generation of large amount of redundant data and wastage of network bandwidth due to transmission of such useless data. Thus to overcome these inherent bottlenecks with traditional crawling techniques we have proposed the design of a parallel migrating web crawler. We first present detailed requirements followed by the architecture of a crawler.**

Keywords: **web crawler, parallel, migration, web database**

I.Introduction

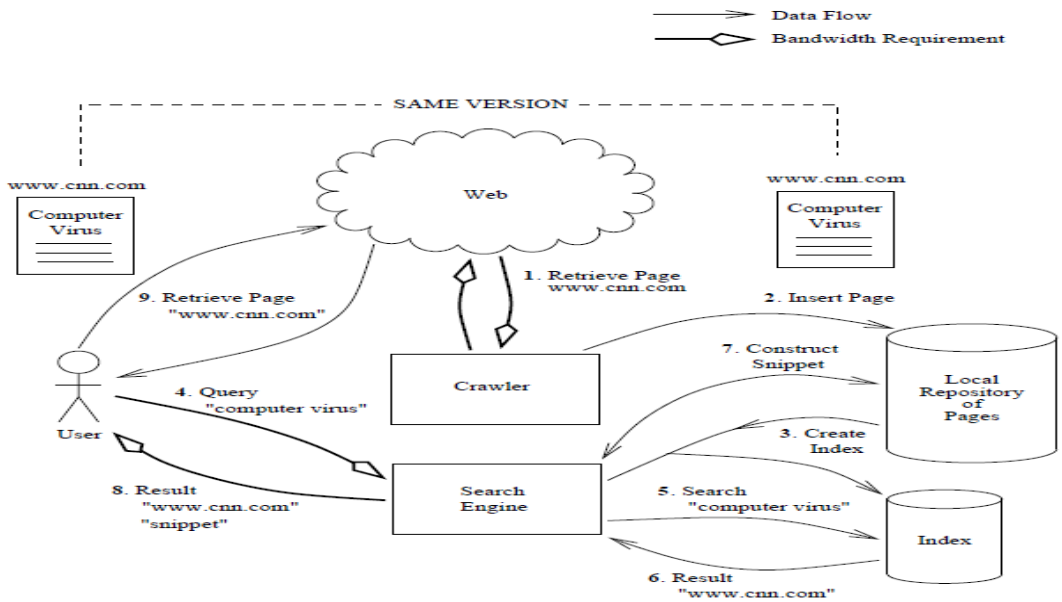
1.1 Introduction to Crawling

The Web contains large volumes of documents and resources that are linked together. In the early days of the Web, manually locating relevant information was reasonably easy due to the limited amount of information that was available. Typically, users found relevant information with the aid of manually maintained web rings, links pages, and directories, such as Yahoo! [2008] and later DMOZ [2008], which were organised by topic. However, as the size of the Web grew, these approaches were augmented or replaced by automated systems using web crawlers and search engines.

Search engines typically support "bag of word" querying techniques, where users enter query terms and the search engine ranks web documents by their likelihood of relevance to the query terms. This approach, while effective, requires an index of documents on the Web. This index is created by retrieving a copy of every document to be indexed, from the Web, a task that is undertaken by a web crawler. Web crawlers exploit the link structure of web documents and traverse the Web by retrieving documents, extracting the embedded URLs, and following them to new documents. Retrieved documents are placed in a central repository so they can be indexed. Once indexed, a document is ranked in response to user queries and its URL is returned to searchers as part of a ranked list.

The user then follows the link to the live Web copy of the document. We highlight this process in Figure 1[1] and the following example.

1. The crawler retrieves a document about a computer virus from the CNN home-page.
2. The crawler inserts the document into the local repository.
3. The search engine indexes the documents in the local



repository.
4. A user poses the query "computer virus".
5. The search engine examines the index for relevant documents.
6. The search engine locates the CNN home-page and retrieves its URL.
7. The search engine creates a short snippet or summary from the cached document.
8. The search engine returns the CNN home-page URL and snippet to the user.
9. The user clicks on the URL and is presented with the CNN home-page containing the computer virus article.

However, the Web is a volatile environment where documents are frequently created, modified, and removed, which means that crawlers must revisit documents periodically to update the local repository. Index inconsistency occurs when crawlers fail to recrawl documents that have changed.

1.3 Pseudo code

An informal description of the remotely executed crawling algorithm could look like the following pseudocode [3] :

```
/**
* Pseudocode for a simple subject specific
* mobile crawler.
*/
        migrate to web server;
put server url in url_list;
for all url ∈ url_list do begin
// *** local data access
load page;
// *** page analysis
        extract page keywords;
        store page in page_list if relevant;
        // *** recursive crawling
        extract page links;
        for all link ∈ page do begin
        if link is local then
        add link to url_list;
        else
        add link to external_url_list;
        end
end
```

II.Web Crawler Requirements

Figure 2[1] shows the structure of a generic crawling process . We now discuss the requirements for a good crawler, and approaches for achieving them. [7]

2.1 Flexibility:

We would like to be able to use the system in a variety of scenarios, with as few modifications as possible.

2.2 Low Cost and High Performance:

The system should scale to at least several hundred pages per second and hundreds of millions of pages per run, and should run on low-cost hardware. Note that efficient use of disk access is crucial to maintain a high speed after the main data structures, such as the *"URL seen"* structure and crawl frontier, become too large for main memory. This will only happen after downloading several million pages.

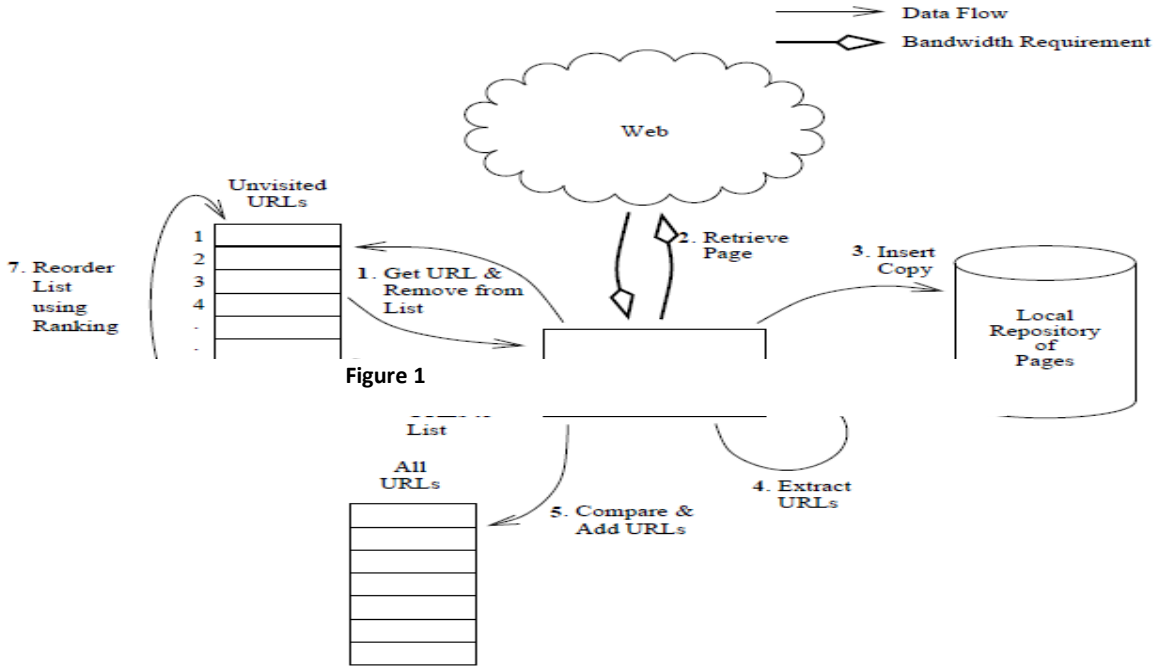


Figure 1

Figure 2 : The crawling process

2.3 Robustness:

There are several aspects here. First, since the system will interact with millions of servers, it has to tolerate bad HTML, strange server behaviour and configurations, and many other odd issues. Our goal here is to err on the side of caution, and if necessary ignore pages and even entire servers with odd behaviour, since in many applications we can only download a subset of the pages anyway. Secondly, since a crawl may take weeks or months, the system needs to be able to tolerate crashes and network interruptions without losing (too much of) the data. Thus, the state of the system needs to be kept on disk. We note that we do not really require strict ACID properties. Instead, we decided to periodically synchronize the main structures to disk, and to recrawl a limited number of pages after a crash.

2.4 Etiquette and Speed Control:

It is extremely important to follow the standard conventions for robot exclusion (robots.txt and robots meta tags), to supply a contact URL for the crawler, and to supervise the crawl. In addition, we need to be able to control access speed in several different ways. We have to avoid putting too much load on a single server; we do this by contacting each site only once every 30 seconds unless specified otherwise. It is also desirable to throttle the speed on a domain level, in order not to overload small domains, and for other reasons to be explained later. Finally, since we are in a campus environment where our connection is shared with many other users, we also need to control the total download rate of our crawler.

2.5 Manageability & Reconfigurability:

An appropriate interface is needed to monitor the crawl, including the speed of the crawler, statistics about hosts and pages, and the sizes of the main data sets. The administrator should be able to adjust the speed, add and remove components, shut down the system, force a checkpoint, or add hosts and domains to a "blacklist" of places that the crawler should avoid. After a crash or shutdown, the software of the system may be modified to fix problems, and we may want to continue the crawl using a different machine configuration.

2.6 Localized Data Access:

The main task of stationary crawlers in traditional search engines is the retrieval of Web pages on behalf of the search engine. In the context of traditional search engines one or more stationary crawlers attempt to recursively download all documents managed the existing Web servers. Due to the HTTP request/response paradigm, downloading the contents from a Web server involves significant overhead due to request messages which have to be sent for each Web page separately. Using a mobile crawler we reduce the HTTP overhead by transferring the crawler to the source of the data.

2.7 Remote Page Selection:

By using mobile crawlers we can distribute the crawling logic (i.e. the crawling algorithm) within a system of distributed data sources such as the Web. This allows us to elevate Web crawlers from simple data retrieval tools to more intelligent components which can exploit information about the data they are supposed to retrieve. Crawler mobility allows us to move the decision whether or not certain pages are relevant to the data source itself. Once a mobile crawler has been transferred to a Web server, it can analyze each Web page before sending it back which would require network resources. By looking at this so-called remote page selection from a more abstract point of view, it compares favourably with classical approaches in database systems.

2.8 Remote Page Filtering:

Remote page filtering extends the concept of remote page selection to the contents of a Web page. The idea behind remote page filtering is to allow the crawler to control the granularity of the data it retrieves. With stationary crawlers, the granularity of retrieved data is the Web page itself since HTTP allows page-level access only. For this reason, stationary crawlers always have to retrieve a whole page before they can extract the relevant page portion. Depending on the ratio of relevant to irrelevant information, significant portions of network bandwidth are wasted by transmitting useless data.

III.System Architecture

We now discuss the architecture of a generic parallel migrating web crawler.

3.1 Crawler Manager

We have to initialize crawler objects with some initial facts to begin crawler execution. As an example, consider a crawler which tries to examine a certain portion of the Web. This particular kind of crawler will need initial fact seeds containing URL addresses as starting points for the crawling process. The structure and the content of initial facts depends on the particular crawler specification used.

Initialized crawler objects are transferred to a location which provides a crawler runtime environment. Such a location is either the local host (which always has a runtime environment installed) or a remote system which explicitly allows crawler execution through an installed crawler runtime environment. The crawler manager is responsible for the transfer of the crawler to the execution location. The migration of crawler objects and their execution at remote locations implies that crawlers have to return to their home systems once their execution is finished. Thus, the crawler manager has to wait for returning crawlers and has to indicate their arrival to other

components (e.g., query engine) interested in the results carried home by the crawler.

To fulfil the tasks specified above, the crawler manager uses an inbox/outbox structure similar to an email application. Newly created crawlers are stored in the outbox prior to their transmission to remote locations. The inbox buffers crawlers which have returned from remote locations together with the results they contain. Other system components such as the query engine can access the crawlers stored in the inbox through the crawler manager interface. Figure 3[3] summarizes the architecture of the crawler manager and also demonstrates its tight cooperation with the communication subsystem.

3.2 Query Engine

The query engine basically establishes an interface between the application framework and the application specific part of the system. From a more abstract point of view, the query engine establishes a SQL like query interface for the Web by allowing users to issue queries to crawlers containing portions of the Web. Since retrieved

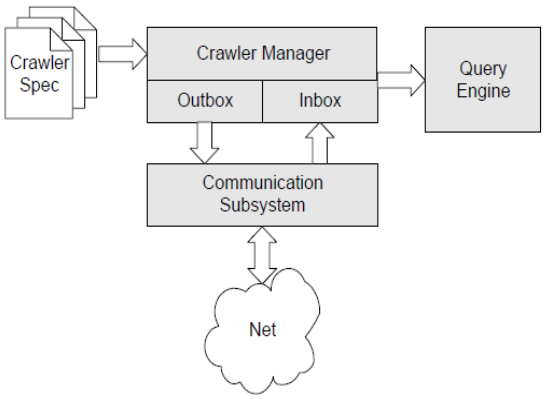


Figure 3

Web pages are represented as facts within the crawler memory, the combination of mobile crawlers and the query engine provides a translation of Web pages into a format that is queriable by SQL.

3.3 Database Drivers

3.3.1 Database Connection Manager : Since our framework is based on Java we have decided to use the JDBC (Java Database Connectivity) interface to implement the necessary database mechanisms. JDBC provides a standard SQL interface to a wide range of relational database management systems by defining Java classes which represent database connections, SQL statements, result sets, database metadata, etc. The JDBC API allows us to issue SQL statements to a database and process the results that the database returns. The JDBC implementation is based on a driver manager that can support multiple drivers to allow connections to different databases. These JDBC drivers can either be written in Java entirely or they can be implemented using native methods to bridge existing database access libraries. The JDBC configuration used for our framework as depicted in Figure 5[3] uses a client side JDBC driver to access relational databases. The JDBC API uses a connection paradigm to access the actual databases. Once a database is identified by the user, JDBC creates a connection object which handles all further communication with the database.

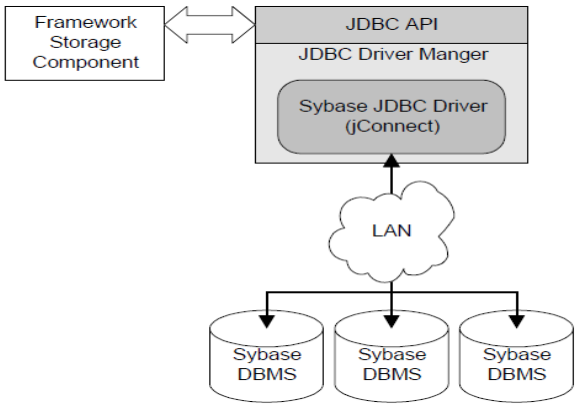


Figure 4

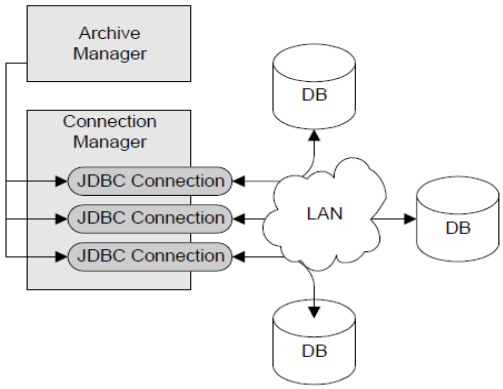


Figure 5

3.3.2 Database Command Manager : The connection manager allows the archive manager to connect to multiple, distributed databases within the local network. By establishing a database connection through the connection manager, the archive manager can interact with the database by issuing SQL commands. Such an interaction requires the archive manager to have knowledge of the structure and the semantics of each database it works with as shown in Figure 4[3] . We do not intend to impose any particular data model and storage structures upon the user of our framework. Thus, the organization of data with the database cannot be known to the archive manager because it is defined in the context of the user application which utilizes our framework.

3.4 Robots Exclusion protocol

The Web crawler tries to comply with the Robots Exclusion protocol and not crawl Web sites if rules in the server's robots.txt file disallow crawling. A successful download is when the crawler can retrieve the robots.txt file from a Web server or confirm that a robots.txt file does not exist. The download is considered a failure when the crawler cannot obtain the rules or cannot confirm that a robots.txt file exists. A successful download does not mean that the crawler has permission to crawl because rules in the robots.txt file can disallow crawling. A download failure temporarily prohibits crawling because the crawler cannot determine what the rules are.

These are the steps that the crawler takes when attempting to download the robots.txt file:

When the crawler discovers a new site, it tries to obtain the server's IP address. If this attempt fails, crawling is not possible. When at least one IP address is available, the crawler tries to download the robots.txt file by using HTTP (or HTTPS) GET. If the socket connection times out, is broken, or another low-level error occurs (such as an SSL certificate problem), the crawler logs the problem, and repeats the attempt on every IP address known for the target server. If no connection is made after the crawler tries all addresses, the crawler waits two seconds, then tries all the addresses one more time. If a connection is made, and HTTP headers are exchanged, the return status is examined. If the status code is 500 or higher, the crawler interprets this as a bad connection and continues trying other IP addresses. For any other status, the crawler stops trying alternative IP addresses and proceeds according to the status code. After the crawler receives an HTTP status code below 500, or after the crawler tries all IP addresses twice, the crawler proceeds as follows:

If no HTTP status below 500 was received, the site is disqualified for the time being.
If an HTTP status of 400, 404 or 410 was received, the site is qualified for crawling with no rules.
If an HTTP status of 200 through 299 was received, the following conditions direct the next action:
If the content was truncated, the site is disqualified for the time being.
If the content parsed without errors, the site is qualified for crawling with the rules that were found.
If the content parsed with errors, the site is qualified for crawling with no rules.
If any other HTTP status was returned, the site is disqualified for the time being.

When the crawler attempts to download the robots.txt file for a site, it updates a persistent timestamp for that site called the robots date. If a site is disqualified because the robots.txt information is not available, the persistent robots failure count is incremented.

When the retry interval is reached, the crawler tries again to retrieve robots.txt information for the failed site. If the number of successive failures reaches the maximum number of failures allowed, the crawler stops trying to retrieve the robots.txt file for the site and disqualifies the site for crawling. After a site is qualified for crawling (the check for robots.txt file rules succeeds), the failure count is set to zero. The crawler uses the results of the download until the interval for checking rules elapses. At that time, the site must be qualified again.

3.5 Parallel Implementation

The same crawler can be run in parallel mode on various machines which can then store the retrieved information in a central database. Access to the central database can be synchronized using locks.

3.5.1 URL Handling: The hyperlinks parsed from the files, after normalization of relative links, are then checked against the "URL seen" structure that contains all URLs that have been downloaded or encountered as hyperlinks thus far. A parsing speed of 300 pages per second results in more than 2000 URLs per second that need to be checked and possibly inserted. Each URL has an average length of more than 50 bytes, and thus a naive representation of the URLs would quickly grow beyond memory size.

Several solutions have been proposed for this problem. The crawler of the Internet Archive uses a Bloom filter stored in memory; this results in a very compact representation, but also gives false positives, i.e., some pages are never downloaded since they collide with other pages in the Bloom filter. Lossless compression can reduce URL size to below 10 bytes though this is still too high for large crawls. In both cases main memory will eventually become a bottleneck, although partitioning the application will also partition the data structures over several machines. A more scalable solution uses a disk-resident structure, as for example done in Mercator . Here, the challenge is to avoid a separate disk access for each lookup and insertion. This is done in Mercator by caching recently seen and frequently encountered URLs, resulting in a cache hit rate of almost 85% . Nonetheless, their system used several fast disks for an average crawl speed of 112 pages per second.

3.5.2 Downloaders and DNS Resolvers The downloader component, implemented in Python, fetches files from the web by opening up to 1000 connections to different servers, and polling these connections for arriving data. Data is then marshaled into files located in a directory determined by the application and accessible via NFS. Since a downloader often receives more than a hundred pages per second, a large number of pages have to be written out in one disk operation.

We note that the way pages are assigned to these data files is unrelated to the structure of the request files sent by the application to the manager. Thus, it is up to the application to keep track of which of its URL requests have been completed. The manager can adjust the speed of a downloader by changing the number of concurrent connections that are used.

The DNS resolver, implemented in C++, is also fairly simple. It uses the GNU adns asynchronous DNS client library to access a DNS server usually collocated on the same machine. While DNS resolution used to be a significant bottleneck in crawler design due to the synchronous nature of many DNS interfaces, we did not observe any significant performance impacts on our system while using the above library. However, DNS lookups generate a significant number of additional frames of network traffic, which may restrict crawling speeds due to limited router capacity.

3.6 Crawler Migration

To demonstrate the advantages of mobile crawling, we present the following example. Consider a special purpose search engine which tries to provide high quality searches in the area of health care. The ultimate goal of this search engine is to create an index of the part of the Web which is relevant to health care issues. The establishment of such a specialized index using the traditional crawling approach is highly inefficient. This inefficiency is because traditional crawlers would have to download the whole Web page by page in order to be able to decide whether a page contains health care specific information. Thus, the majority of downloaded pages would not be indexed.

In contrast, a mobile crawler allows the search engine programmer to send a representative of the search engine (the mobile crawler) to the data source in order to filter it for relevant material before transmitting it back to the search engine. In our example, the programmer would instruct the crawler to migrate to a Web server in order to execute the crawling algorithm at the data source.

The important difference is that our crawler gets executed right at the data source by the mobile crawler. The crawler analyzes the retrieved pages by extracting keywords. The decision, whether a certain page contains relevant health care information can be made by comparing the keywords found on the page with a set of predefined health care specific keyword known to the crawler. Based on this decision, the mobile crawler only keeps pages which are relevant with respect to the subject area.

As soon as the crawler finishes crawling the whole server, there will be a possibly empty set of pages in its memory. Please note that the crawler is not restricted to only collecting and storing Web pages. Any data which might be important in the context of the search engine (e.g., page metadata, Web server link structure) can be represented in the crawler memory. In all cases, the mobile crawler is compression to significantly reduce the data to be transmitted. After compression, the mobile crawler returns to the search engine and is decompressed. All pages retrieved by the crawler are then stored in the Web index. Please note, that there are no irrelevant pages since they have been discarded before transmission by the mobile crawler. The crawler can also report links which were external with respect to the Web server crawled. The host part of these external addresses can be used as migration destination for future crawls by other mobile crawlers.

By looking at the example discussed above, the reader might get an idea about the potential savings of this approach. In case a mobile crawler does not find any useful information on a particular server, nothing beside the crawler code would be transmitted over the network. If every single page of a Web server is relevant, a significant part of the network resources can be saved by compressing the pages prior to transmission. In both of these extreme cases, the traditional approach will produce much higher network loads.

IV. Conclusions and Future Work

We have described the architecture and implementation details of our crawling system, and presented some preliminary experiments. There are obviously many improvements to the system that can be made. A major open issue for future work is a detailed study of the scalability of the system and the behaviour of its components. This could probably be best done by setting up a simulation testbed, consisting of several workstations, that simulates the web using either artificially generated pages or a stored partial snapshot of the web. We are currently considering this, and are also looking at testbeds for other high-performance networked systems (e.g., large proxy caches).

Our main interest is in using the crawler in our research group to look at other challenges in web search technology, and several students are using the system and acquired data in different ways.

References

[1] Halil Ali. Effective web crawlers, Mar 2008.
[2] R. Baeza-Yates and B. Rebeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.
[3] Jan Fiedler and Joachim Hammer. Using Mobile Crawlers to Search the Web Efficiently, 2000.
[4] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. In *7th Int. World Wide Web Conference*, May 1998.
[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World-Wide Web Conference*, 1998.

[6] M. Burner. Crawling towards eternity: Building an archive of the world wide web. *Webtechniques*, 1997.
[7] Joo Yong Lee, Sang Ho Lee, Yanggon Kim SCRAWLER: A Seed -By-Seed Parallel Web Crawler
[8] S. Chakrabarti, M. van den Berg, and B. Dom. Distributed hypertext resource discovery through examples. In *Proc. Of 25th Int. Conf. on Very Large Data Bases*, pages 375–386, September 1999.
[9] Vladislav Shkapenyuk, Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler
[10] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, pages 117–128, Sept. 2000.
[11] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 117–128, May 2000.
[12] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *7th Int.World Wide Web Conference*, May 1998.
[13] M. Diligenti, F. Coetzee, S. Lawrence, C. Giles, and M. Gori. Focused crawling using context graphs. In *Proc. of 26th Int.Conf. on Very Large Data Bases*, September 2000.
[14] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the web. In *Proc. of the 8th Int. World Wide Web Conference (WWW8)*, pages 213–225, 1999.

